Revision 1.1, June 1998

Written by Christopher Dobrian
Assistant Professor and Director of the Electronic Music Studio
Music Department, School of the Arts
University of California, Irvine

MSP © 1997 David Zicarelli—All rights reserved
based on Pd by Miller Puckette
© 1997 The Regents of the University of California
MSP and Pd are based on ideas in Max/FTS,
an advanced DSP platform © IRCAM. Used by permission.

Cycling '74
1186 Folsom Street
San Francisco, CA 94103 USA
(415) 621-5743
fax (415) 621-6563
info@cycling74.com
http://www.cycling74.com

Table of Contents

Introduction.....4
PowerPC signal processing in Max.....4
How the MSP documentation is organized.....6
Reading the manual online.....6

Digital Audio.....8
Sound.....8
Digital representation of sound.....15
Limitations of digital audio.....17
Advantages of digital audio.....21

How MSP Works.....23
Max patches and the MSP “signal network”.....23
Audio rate and control rate.....23
The link between Max and MSP.....24
Limitations of MSP.....26
Advantages of MSP.....27

Tutorial.....28
Introduction.....28
Fundamentals.....30
1. Test tone.....30
2. Adjustable oscillator.....34
3. Wavetable oscillator.....38
4. Routing signals.....43
5. Turning signals on and off.....51
6. Review.....58
Synthesis.....63
7. Additive synthesis.....63
8. Tremolo and ring modulation.....67
9. Amplitude modulation.....71
10. Vibrato and FM.....74
11. Freqeuency modulation.....76
12. Waveshaping.....80
Sampling.....84
13. Recording and playback.....84
14. Playback with loops.....88
15. Variable-length wavetable.....90
16. Record and play sound files.....95
17. Review.....99
MIDI control.....103
18. Mapping MIDI to MSP.....103
19. Synthesizer.....108
20. Sampler.....114
21. Panning.....120

Analysis 126
22. Viewing signal data 126
23. Oscilloscope 132
24. Using the FFT 135
Processing 141
25. Delay lines 141
26. Delay lines with feedback 144
27. Flange 147
28. Chorus 151
29. Comb filter 154

Audio Input and Output 159
Using the Sound Manager with MSP 163
The volume (the Sound Out level) 164
The sampling rate 164
The Sound Input source 164
The Sound Output destination 165
Adjusting input-output delay 166
Using audio interface cards 167
The audiodrivers folder 167
INITs for audio cards 169
Changing audio settings 169
Using more than two audio channels 169
Notes on specific audio cards 170
Digidesign Audiomedia II 170
Digidesign Audiomedia III, ProTools, and d24 170
Lucid PCI24 171
Sonorus StudI/O 172
Korg 1212I/O 174

Objects 177

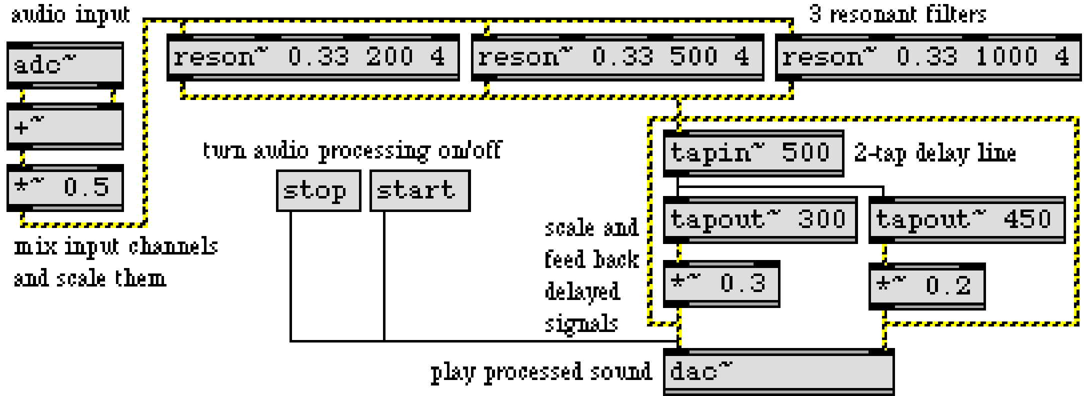
Messages to dsp 290

MSP Object Thesaurus 291

Index 294

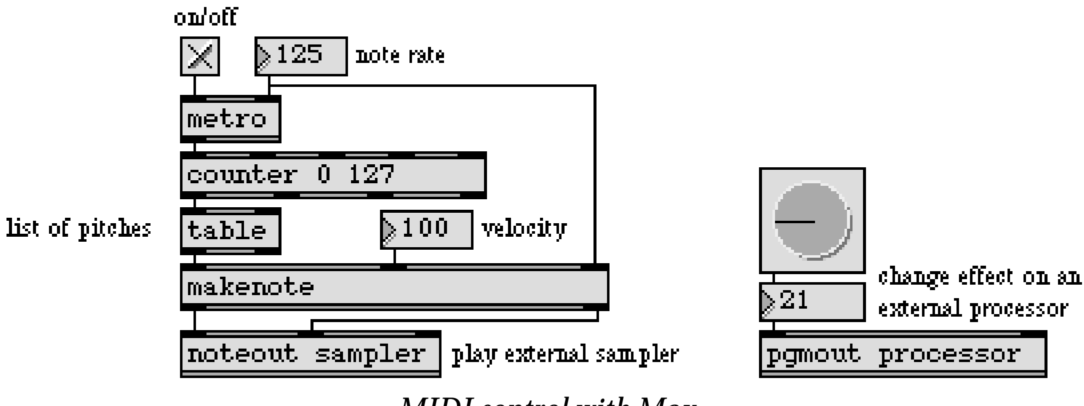
PowerPC signal processing in Max

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



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.

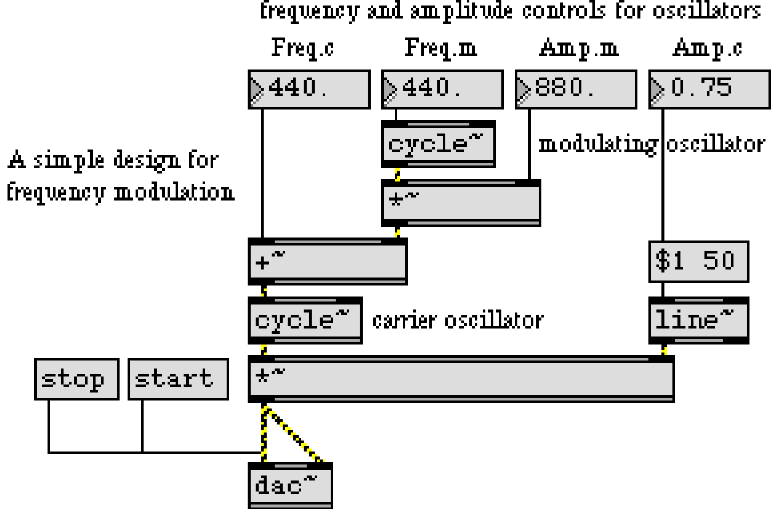


MIDI control with Max

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 the Power PC (using the Sound Manager in the Mac OS) or through an installed sound card such as the Digidesign Audiomedia III.



Signal network for an FM instrument

How the MSP documentation is organized

The organization of the MSP manual is similar to that of the Max manual. It consists of introductory and background material, a substantial number of tutorials, and a reference section.

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 a lot of 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. Unless you're familiar with Max/FTS from IRCAM, this chapter will provide essential information.

The *MSP Tutorial* accompanies the Max patches found in the MSP Tutorial folder. You'll read each chapter as you try out the examples in the patches. We recommend that everyone work through the tutorials, regardless of background. MSP is a deep and complex system and requires some thought and effort to learn. Let the tutorials help you in this process. The other reference materials in the manual provide the facts about MSP, but they do not provide the necessary context for understanding the program in the way that the tutorial does.

Audio Input and Output describes MSP's support for the Macintosh Sound Manager and audio-interface cards. There is a lot of detail about audio interface cards that you can skip if you don't have one. You'll also want to read this chapter to learn how you can tweak MSP's performance.

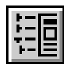
MSP Objects includes a page or two describing the capabilities of each object, along with a Max patch or excerpt that illustrates one use of the object. Some of the more obscure objects are not covered in the tutorials; you'll need to read about them here.

Messages to dsp lists the messages you can send to the dsp object to access low-level features of MSP. This section will be of interest only to advanced users.

The *MSP Object Thesaurus* matches the subject and typical uses of MSP objects with the object name. For example, if you are interested in sample playback but do not know which MSP object does it, you could look up "Sample playback" to find the six(!) objects that deal with this topic.

Reading the manual online

The MSP manual takes advantage of a couple of features in Adobe Acrobat Reader. The table of contents is bookmarked, so you can view the bookmarks and jump to any topic listed by clicking on its names. To view the bookmarks, click on the icon that looks like this:



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 msp help folder provide interactive examples of the use of each MSP object.

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

The Support page at www.cycling74.com provides new, updated, and unsupported MSP objects. In addition you'll find updates to audiodrivers and miscellaneous other files. MSP users can contribute objects, examples, and information here.

The Max internet mailing list is the best source of assistance, news, and information about Max and all its applications, including signal processing. For information on how to subscribe, see the Support page at *http://www.cycling74.com/support*.

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 send questions of a more conceptual nature ("how do I…?") to the Max internet mailing list, so that the entire community can provide input and benefit from the discussion.

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 the most extensive bibliography on the subject.

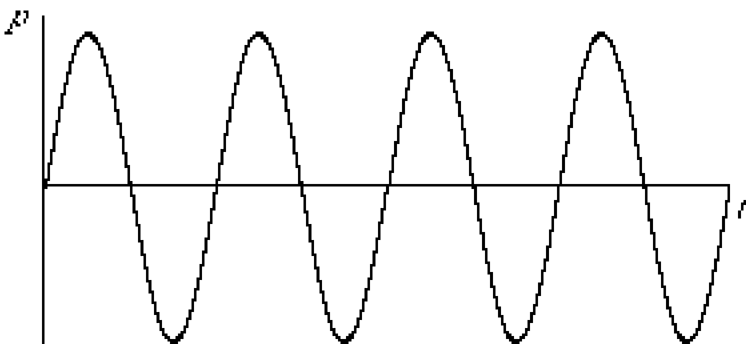
Sound

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:

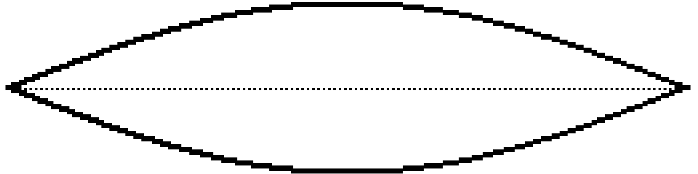
- The fluctuations must be substantial enough to affect our timpanic membrane (eardrum), yet not so substantial as to hurt us. In practice, the intensity of the changes in air pressure must be greater than about 10^{-9} times atmospheric pressure, but not greater than about 10^{-3} times atmospheric 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

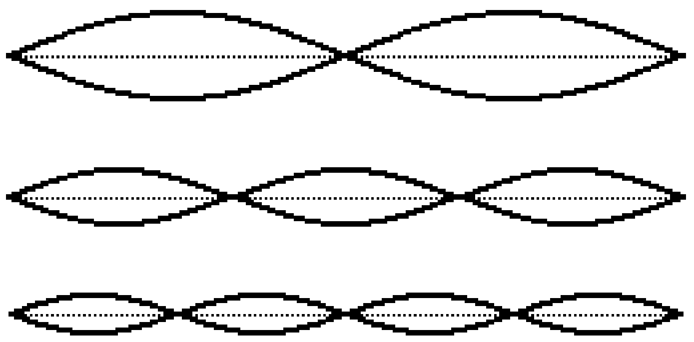
An object that vibrates in simple harmonic motion is said to have a *resonant mode* of vibration—a frequency at which it will naturally tend to vibrate when set in motion. However, most real-world objects have *several* resonant modes of vibration, and thus vibrate at many frequencies at once. Any sound 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.



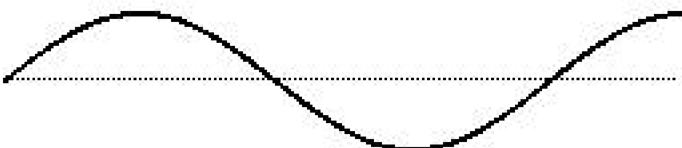
A plucked string vibrating in its fundamental resonant mode

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.



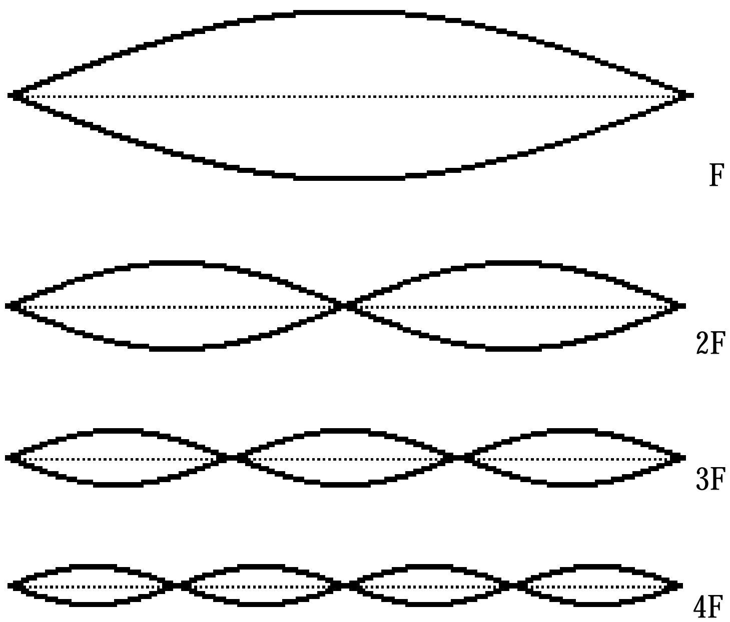
Some other resonant modes of a stretched string

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

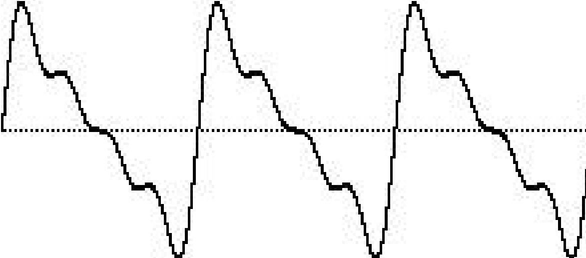
In fact, a plucked string will vibrate in all of these possible resonant modes simultaneously, creating energy at all of the corresponding frequencies. Of course, each mode of vibration (and thus each frequency) will have a different amplitude. (In the example of the guitar string, the longer segments 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.)

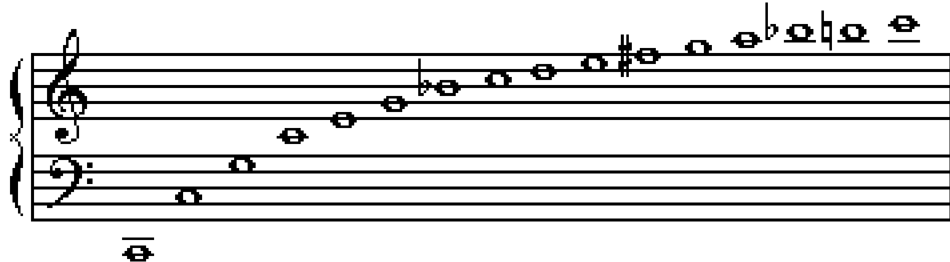
When the partials (component frequencies) in a complex tone are all integer multiples of the same fundamental frequency, as in our example of a guitar string, the sound is said to have a *harmonic spectrum*. Each component of a harmonic spectrum is called a *harmonic partial*, or simply a *harmonic*. 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

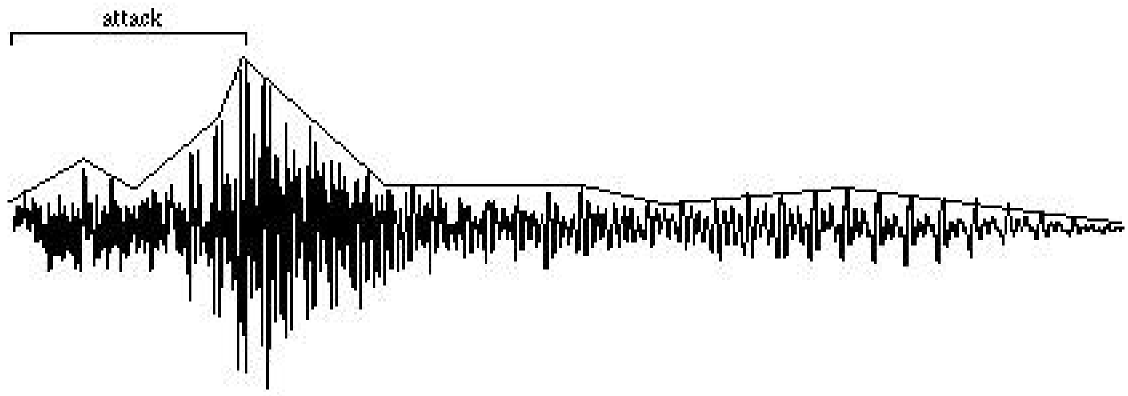
Some objects—such as a bell, for instance—vibrate in even more complex ways, with many different modes of vibrations which may not produce a harmonically related set of partials. If the frequencies present in a tone are not integer multiples of a single fundamental frequency, the wave does not repeat 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 A_0, then the relative amplitude of another sound in decibels can be calculated with the equation:

$$\text{level in decibels} = 20 \log_{10} (A/A_0)$$

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

$$20 \log_{10} (0.5/1) \quad 20 (-0.3) = -6 \text{ dB}$$

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

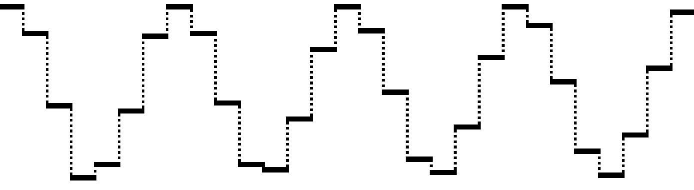
A theoretical understanding of sine waves, harmonic tones, inharmonic complex tones, and noise, as discussed here, is useful to understanding the nature of sound. However, most sounds are actually complicated combinations of these theoretical descriptions, changing from one instant to another. For example, 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

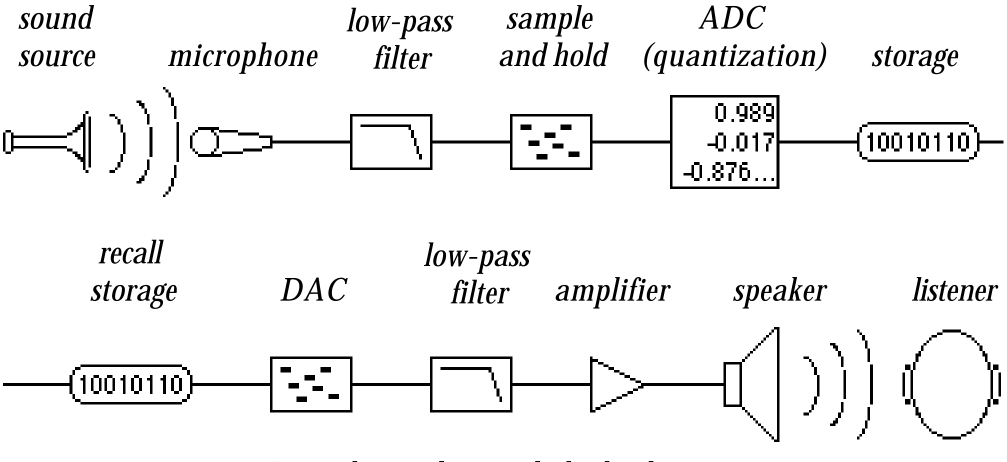
A device known as an *analog-to-digital converter* (ADC) receives the discrete voltages from the sample and hold device, and ascribes a numerical value to each amplitude. This process of converting voltages to numbers is known as *quantization.* Those numbers are expressed in the computer as a string 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 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

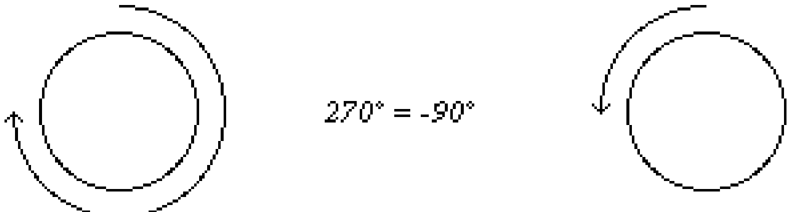
Limitations of digital audio

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.

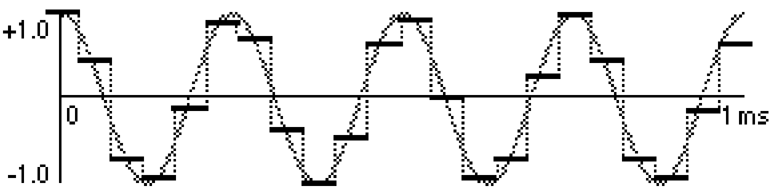
To understand why this aliasing phenomenon occurs, think back to the example of a film camera, which shoots 24 frames per second. If we're shooting a movie of a car, and the car wheel spins at a rate greater than 12 revolutions per second, it's exceeding half the "sampling rate" of the camera. The wheel 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 $-6/24$

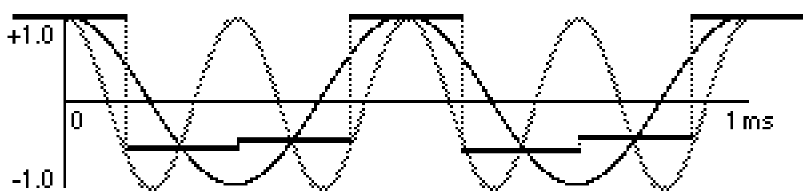
For audio sampling, the phenomenon is practically identical. Any frequency that exceeds the Nyquist rate is indistinguishable from a *negative* frequency the same amount *less than* the Nyquist rate. (And we do not distinguish perceptually between positive and negative frequencies.) To the extent that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

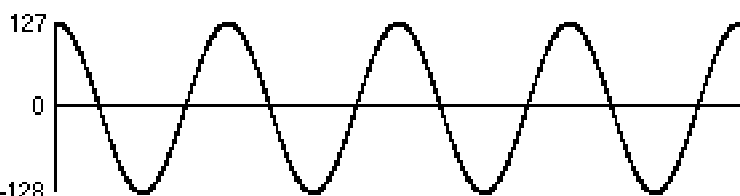
The simple lesson to be learned from the Nyquist theorem is that digital audio cannot accurately represent any frequency greater than half the sampling rate. Any such frequency will be misrepresented by being folded over into the range below half the sampling rate.

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.

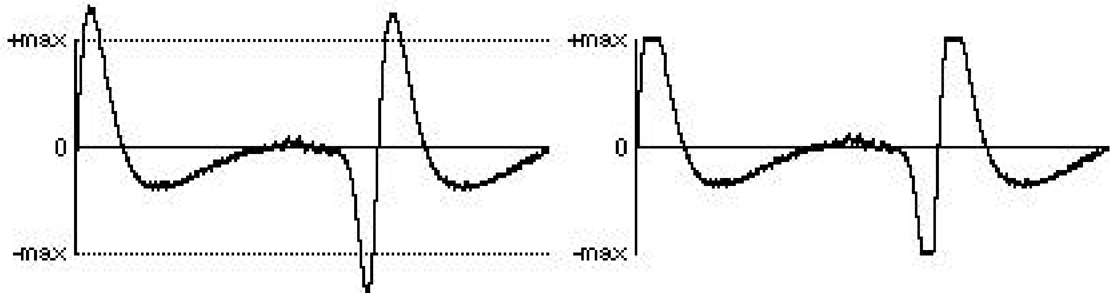
Memory and storage

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:

$$y = A\ sin(2\quad fn/R+ø)$$

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*:

$$y = A_1\ sin(2\quad f_1n/R+ø_1) + A_2\ sin(2\quad f_2n/R+ø_2) + \ldots$$

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.

$$y = x_n + A\, y_{n-1000}$$

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.

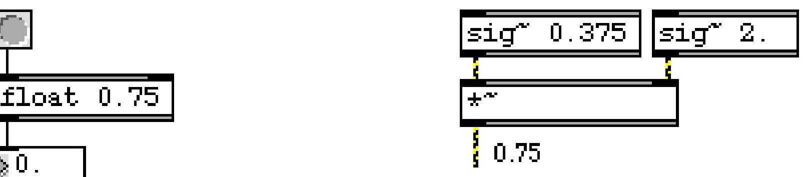
MSP provides a toolkit for exploring this range of possibilities. It integrates digital audio recording, synthesis, and processing with the MIDI control and object-based programming of Max.

Max patches and the MSP "signal network"

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

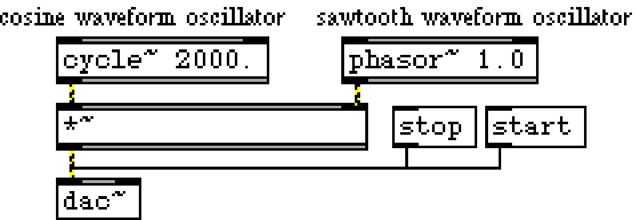
In the Max example on the left, the **number box** doesn't know about the number 0.75 stored in the **float** object. When the user clicks on the **button**, the **float** object sends out its stored value. Only then does the **number box** receive, display, and send out the number 0.75. In the MSP example 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—and smallest—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.



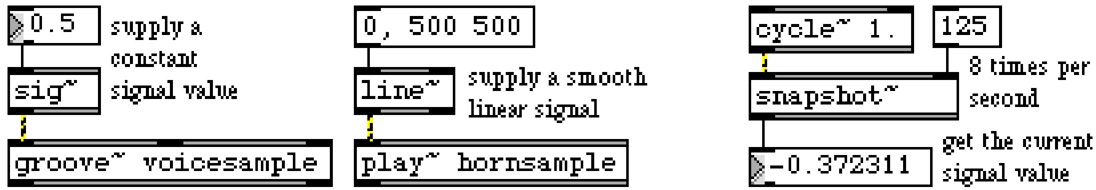
*An oscillator (**cycle~**), and an amplifier (*~) controlled by another oscillator (**phasor~**)*

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 millisecond scheduler, and the faster audio *sample* rate.

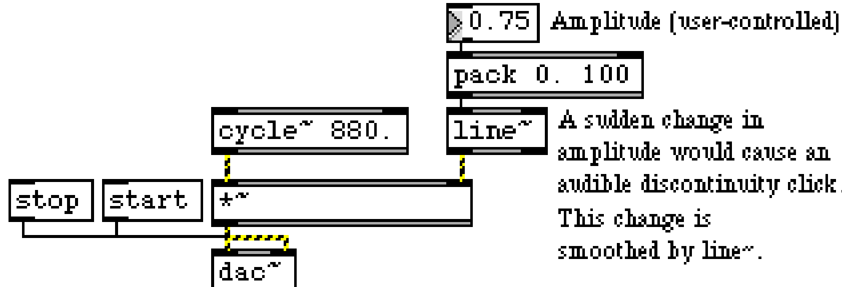
The link between Max and MSP

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.



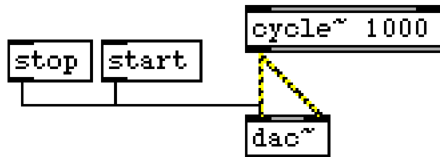
Supply a Max message to the signal network, or get a Max message from a signal

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



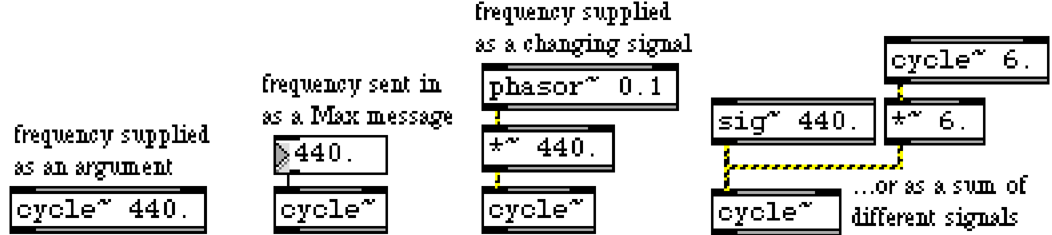
User interface control over the signal's amplitude

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.



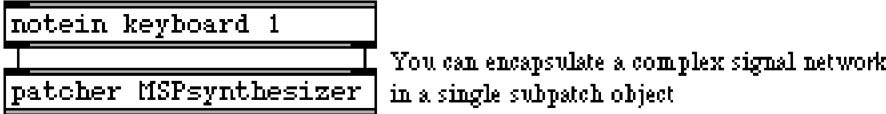
Some MSP objects can receive audio signals and Max messages in the same inlet

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).



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. The PowerBook 5300 series is particularly ill-suited to run MSP, and is not recommended.

- Allocating more RAM to the Max application will increase the available buffer memory for MSP, which will allow it to handle more audio data.

- A fast hard drive and a fast SCSI connection will improve input/output of audio files.

- Turning off AppleTalk and other similar processes that may be making demands on the processor's time will improve MSP's performance. Using File Sharing or other network-intensive AppleTalk processes may freeze MSP when using the Sound Manager, and at best, will cause clicks in the output. The freezing problem is resolved in Max version 3.5.9, and the output click problem is resolved in the version of System 8 that comes with Power Macintosh G3 computers, and in System 8.1.

- 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*.

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 Tutorial*.

Organization of the Tutorial

As with the *Max Tutorial*, the *MSP Tutorial* is organized as a progressive series of lessons. It begins with extremely simple programs and gradually introduces more objects and more ideas for signal processing. The intention is to teach you a bit about different techniques for synthesizing and processing digital audio, and at the same time teach you how to implement those ideas with MSP. This tutorial assumes that you already know how to program with Max.

Each chapter of this Tutorial corresponds to an example patch in the *MSP Tutorial* folder. It is suggested that you refer to the example patch and try it out as you read each lesson.

The chapters progress from simple to more complex, and they are also organized into six subjects:

1) ***fundamentals***—the basics of how to do digital audio with MSP

2) ***synthesis***—generating sounds from scratch (within the computer)

3) ***sampling***—recording and playing back sounds (that originate outside the computer)

4) ***MIDI control***—using MIDI to change the signal processing in real time

5) ***analysis***—getting information about sounds.

6) ***processing***—altering sounds

Preparing for the Tutorial

To get the most out of this tutorial you will need

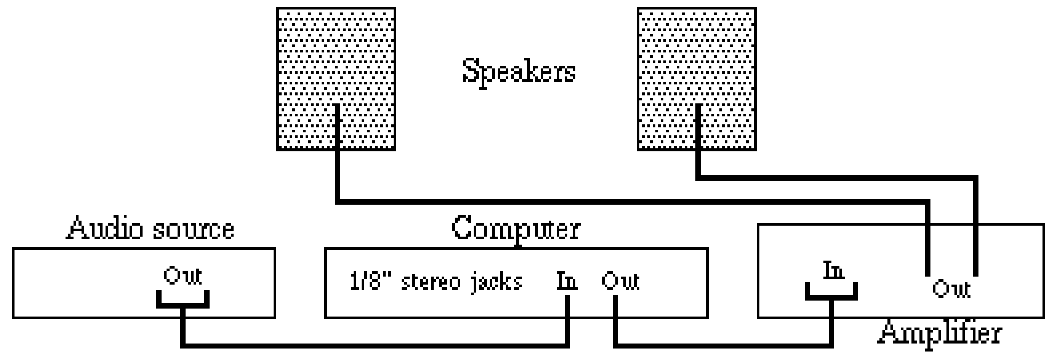
a) a stereo amplifier and speakers (or a pair of amplified multimedia speakers) for the sound output of your computer

b) a line level audio source to play *into* your computer, such as a tape player, and

c) a MIDI keyboard controller with an interface to the computer's modem port.

Audio

If you're using the Sound Manager, you can hear the sounds you make in MSP by playing them through the speakers built into your computer (or your monitor). To hear the best fidelity of sound possible, however, it's preferable to connect the audio output jack of your computer (or the output jacks of your sound card if you have a special audio card installed) to an amplifier and a good pair of speakers.



Audio connections

Important: Please note that the very first example patch plays a 1 kHz test tone at full volume. This can be useful to you for setting the proper level of amplification, but you should begin with your amplification turned way down in order to avoid possible damage to your speakers and your ears. A good way to proceed would be to begin with the amplification turned to zero, start the test tone playing, then increase the volume to the level that you consider to be the appropriate maximum.

MIDI

Many of the example patches use MIDI to play notes in MSP or to control some parameters of an MSP signal network. For this tutorial we make certain assumptions about the MIDI controller you are using. (The assumptions are the same as for the Max Tutorial.) They are:

- Your keyboard controller has at least 61 velocity-sensitive keys, a pitchbend wheel, and a modulation wheel. (If you have any doubt about how to connect the keyboard to your computer, see “Connecting MIDI Equipment” on page 6 of the volume *Getting Started With Max*.)

- Max is receiving MIDI data from the keyboard via OMS on port *a*. For information on how to make sure that this is the case, see “Configuring MIDI (OMS)” on pp. 7-8 of *Getting Started With Max*, and “Using the MIDI Setup Dialog in OMS” on pp. 44-45 of the *Max Reference* manual.

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.

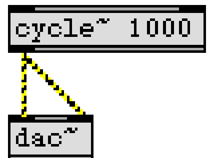
Here's another helpful way to think of the difference. Think of a patch of MSP objects *not* as a program in which events occur at specific instants (as in a standard Max patch), but rather as a description of an instrument design—a synthesizer, sampler, or effect processor. It's like a mathematical 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.

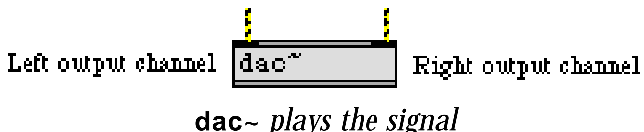


MSP objects are connected by striped patch cords

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 the Sound Manager to play sounds directly from your Power PC'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.)



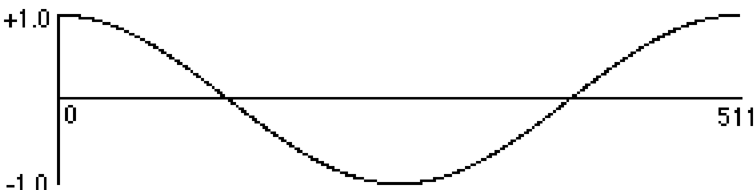
dac~ *plays the signal*

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

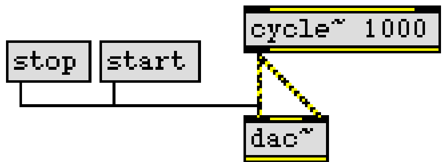
$$I=fL/R$$

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.

- 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, check all your audio connections, and check to ensure that you have all the proper files in your Max folder. Check the Monitors & Sound (or Sound) control panel to ensure that the Sound Out Level is not muted and (if you are listening through your computer's built-in speakers) that the Computer Speaker Volume is not muted. If you are using an audio card, check the Max window to see if MSP has reported any errors in accessing the card.

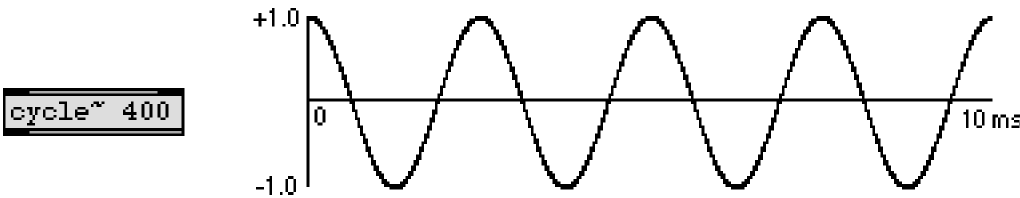
Summary

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.

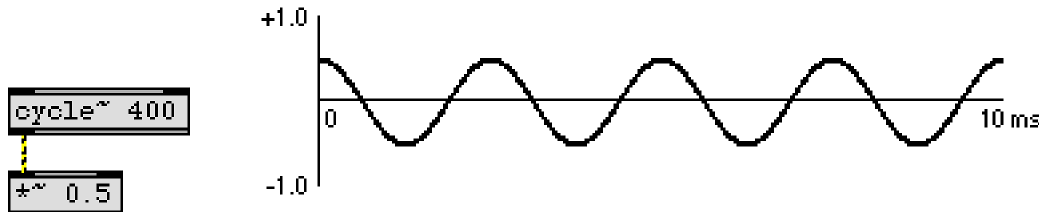
Amplifier: *~

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.



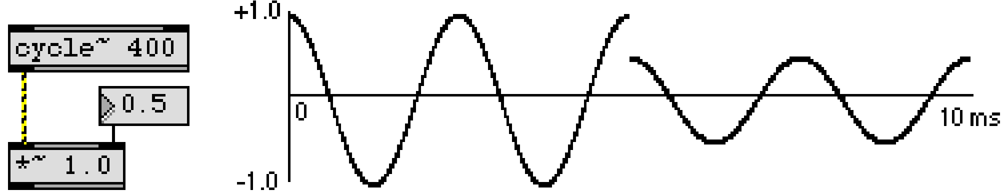
The default output of **cycle~** *has amplitude of 1*

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.

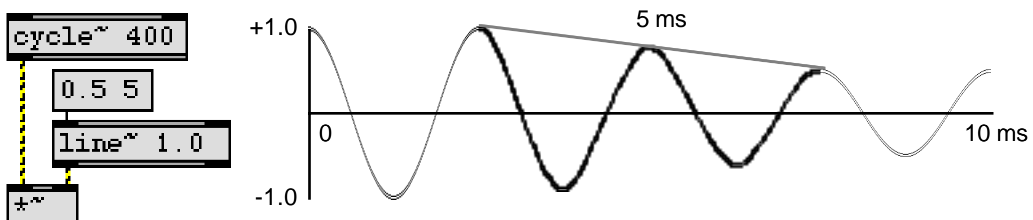


Instantaneous change of amplitude causes a noisy distortion of the signal

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~

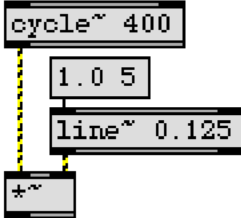
If, instead of an instantaneous change of amplitude (which can cause an objectionable distortion of the signal), we supply a signal in the right inlet of *~ that changes from 1.0 to 0.5 over the course of 5 milliseconds, we interpolate between the starting amplitude and the target amplitude with each 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. One important difference between **line~** and **line** is that **line~** can accept float input for target values and transition times.

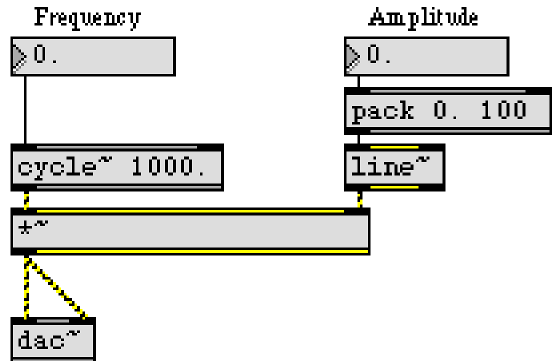
> ***Technical detail:*** Any change in the over-all amplitude of a signal introduces some amount of distortion during the time when the amplitude is changing. (The shape of the waveform is actually changed during that time, compared with the original signal.) Whether this distortion is objectionable 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.



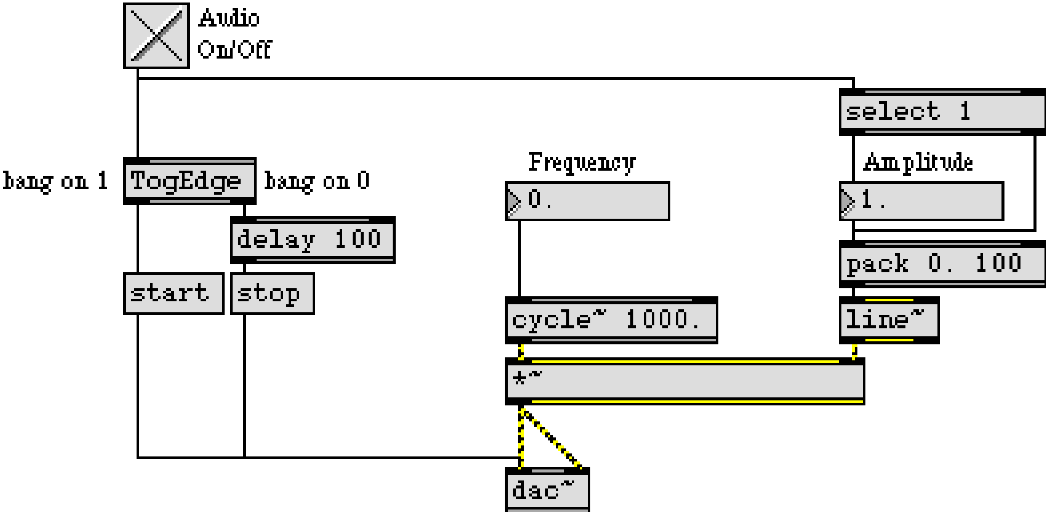
Oscillator with adjustable frequency and amplitude

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**es 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.

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.

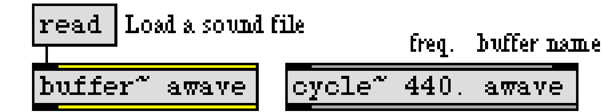
ezdac~ *is an on/off button for audio, available in the object palette*

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~

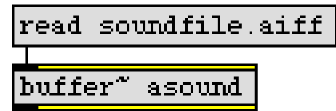
In the previous examples, the **cycle~** object was used to read repeatedly through 512 values describing a cycle of a cosine wave. In fact, though, **cycle~** can read through any 512 values, treating them as a single cycle of a waveform. These 512 numbers must be stored in an object called **buffer~**. (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 a standard file dialog box, allowing you to select an AIFF or Sound Designer II 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 sound 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 a sound file that contains exactly 512 samples.

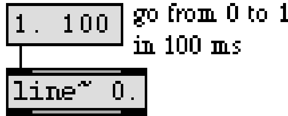
> ***Technical detail:*** In fact, **cycle~** uses 513 samples. The 513th sample is used only for interpolation from the 512th sample. When **cycle~** is being used to create a periodic waveform, as in this example patch, the 513th sample should be the same as the 1st sample. If the **buffer~** contains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

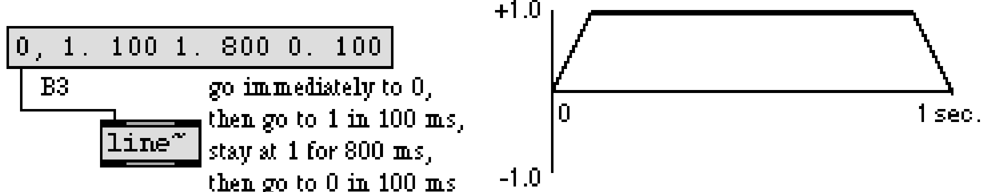
Create a breakpoint line segment function with line~

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.

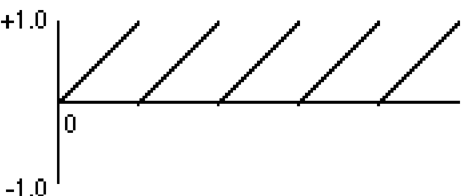


A function made up of line segments

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.

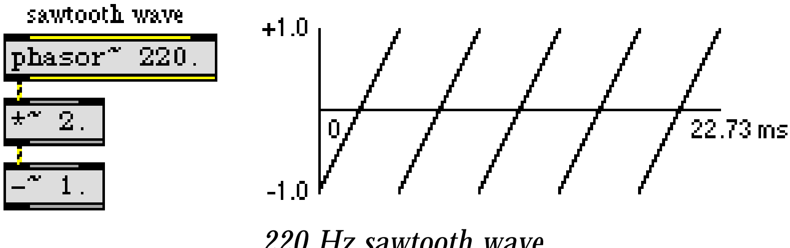
Other signal generators: phasor~ and noise~

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

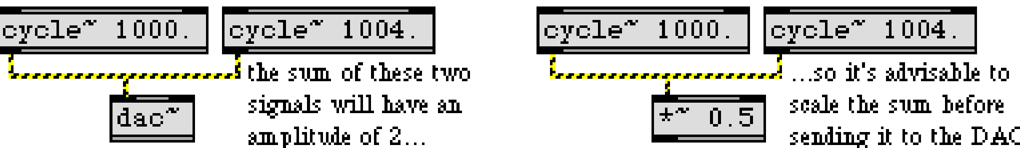
> ***Technical detail:*** A sawtooth waveform produces a harmonic spectrum, with the amplitude of each harmonic inversely proportional to the harmonic number. Thus, if the waveform has amplitude *A*, the fundamental (first harmonic) has amplitude *A*, the second harmonic has amplitude *A/2*, the third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

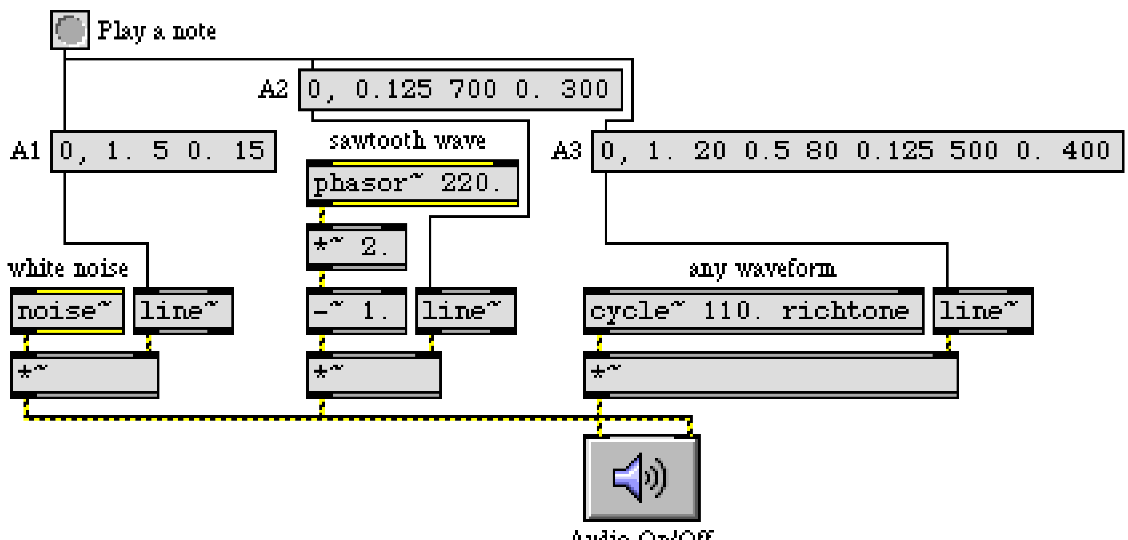
Any time two or more signals are connected to the same signal inlet, those signals are added together and their sum is used by the receiving object.



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 sounds 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 a sound file into **buffer~** with a read message, which opens a standard file 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.

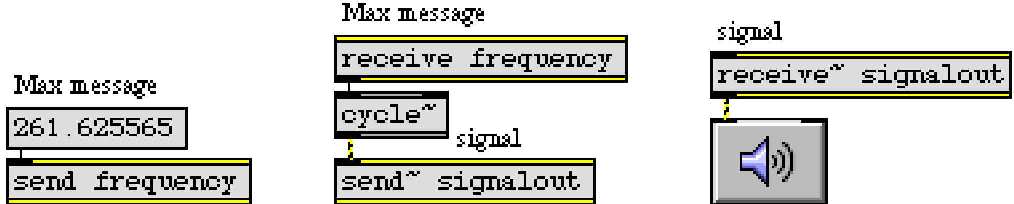
Whenever you connect more than one signal to a given signal inlet, the receiving object adds those signals together and uses the sum as its input in that inlet. Exercise care when mixing (adding) audio signals, to avoid distortion caused by sending a signal with amplitude greater than 1 to the DAC; signals 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.

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.

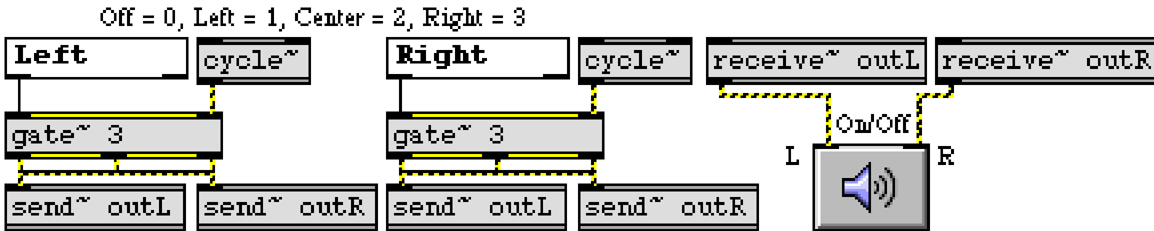
4. Changing the name of a **receive~** object with a set message is a useful way of dynamically redirecting audio signals. Changing the name of **receive**, however, does not redirect the signal until you turn audio off and back on again.

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** objects.



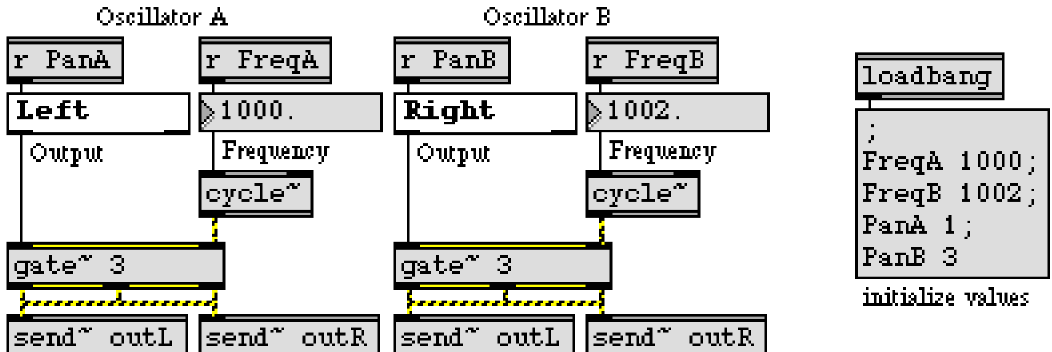
gate~ *sends a signal to a chosen location*

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 **gate~**'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*.

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.



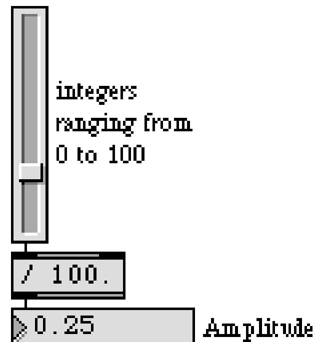
The two waves interfere at a rate of 2 Hz

- 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.

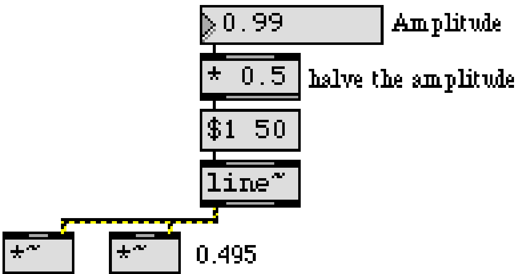
Amplitude and relative amplitude

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

$$dB = 20(\log_{10}(A/A_{ref}))$$

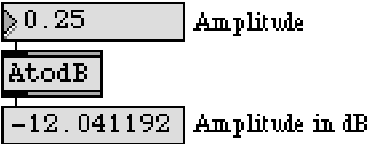
where A is the amplitude being measured and A_{ref} is a fixed reference amplitude.

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

Convert a decimal amplitude to amplitude in decibels. 0dB = 1. (full amplitude)
amplitude expressed as a decimal;
1. = full amplitude
expr 20.*log10($f1)
amplitude expressed in decibels;
0dB = full amplitude

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.



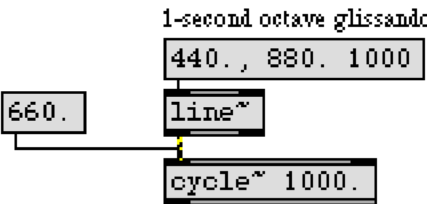
AtodB *reports amplitude in dB, relative to a reference amplitude of 1*

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

Constant signal value: sig~

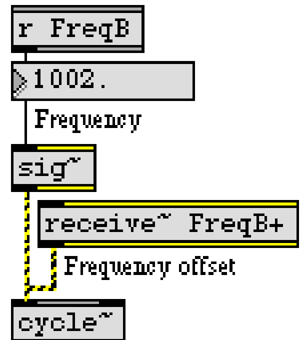
Most signal networks require some changing values (such as an amplitude envelope to vary the amplitude over time) and some constant values (such as a frequency value to keep an oscillator at a steady pitch). In general, one provides a constant value to an MSP object in the form of float message, as we 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.

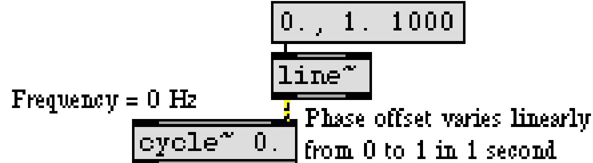
Changing the phase of a waveform

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

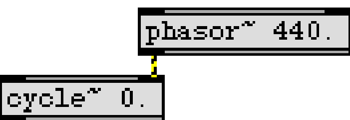
However, there are some very useful reasons to control the phase offset of a wave. For example, by leaving the frequency of **cycle~** at 0, and continuously increasing its phase offset, you can change its instantaneous value (just as if it had a positive frequency). The phase offset of a sinusoid is 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.



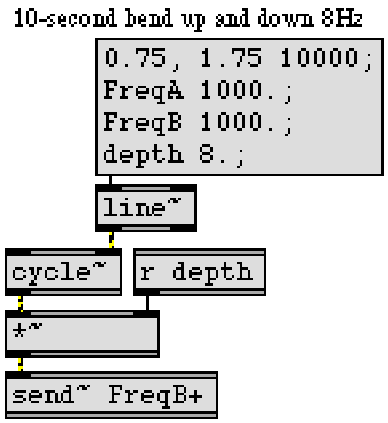
The resulting output is a cosine wave with a frequency of 1 Hz

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~**'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~**'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.

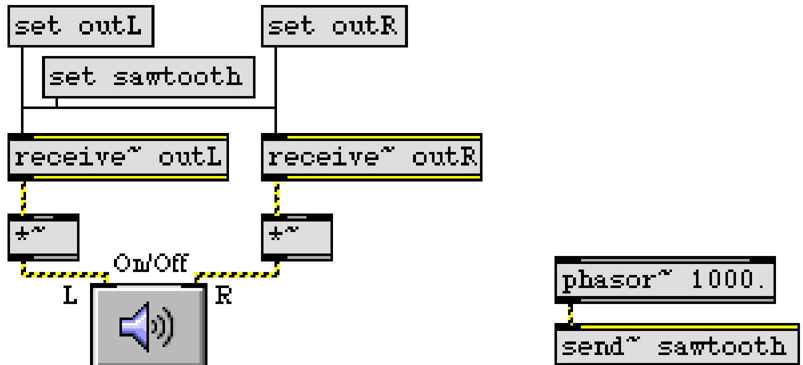


The result is one cycle of a sine wave. The sine wave is multiplied by a "depth" factor to scale its amplitude up to 8. This sub-audio sine wave, varying slowly from 0 up to 8, down to -8 and back to 0, is added to the frequency of Oscillator B. This causes the frequency of Oscillator B to fluctuate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

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.

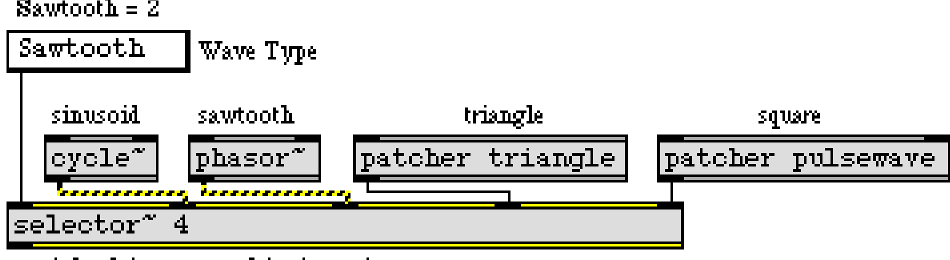
Any of those methods of starting MSP will turn the audio on in *all* open Patcher windows and their subpatches. There is also a way to turn audio processing on and off in a single Patcher.

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

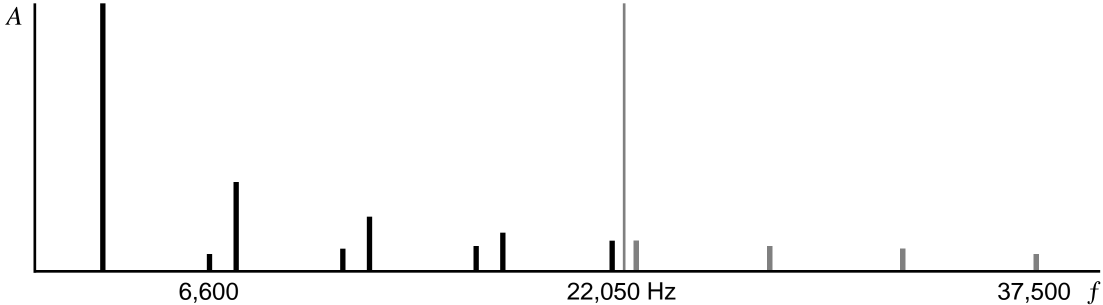
As with **gate~**, switching signals with **selector~** can cause a very abrupt change in the signal being sent out, resulting in unwanted clicks. So if you want to avoid such clicks it's best to change the open inlet of **selector~** only when audio is off or when all of its input signal levels are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



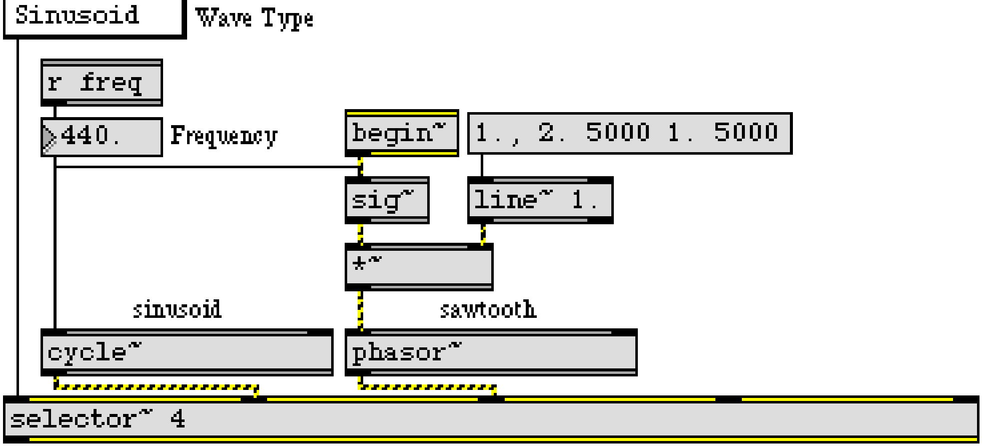
Partials that exceed the Nyquist frequency are folded over

- 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.



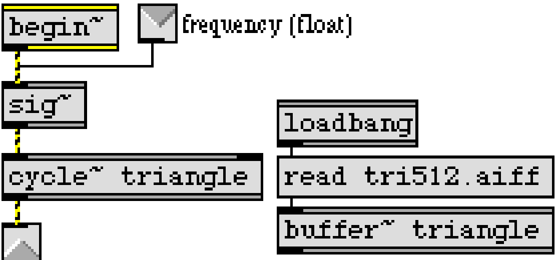
When the sinusoid is chosen, processing for the sawtooth is turned off entirely

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.



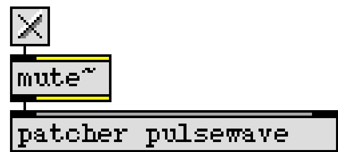
Contents of the **patcher** triangle *object*

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

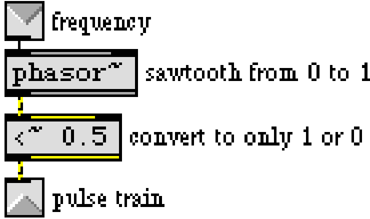
To mute a subpatch, connect a **mute~** object to the inlet of the subpatch, as shown. When **mute~** receives a non-zero int in its inlet, it stops audio processing for all MSP objects in the subpatch. Sending 0 to **mute~**'s inlet unmutes the subpatch.

- 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.

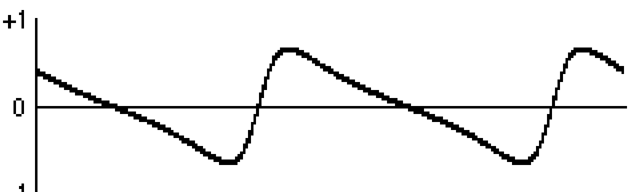
Another way to disable the MSP objects in a subpatch is with the **pcontrol** object. Sending the message enable 0 to a **pcontrol** object connected to a subpatch disables all MSP objects—and all MIDI objects!—in that subpatch. It's the same thing as clicking on the MIDI icon in the title bar of the subpatch's Patcher window. 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.)

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.

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~**, 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 sound 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:

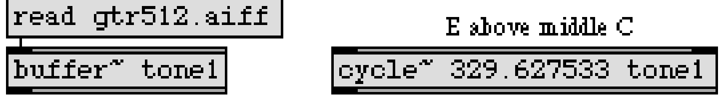
1) This will require yet another amplitude scaling factor (presumably another *~ object) to reduce the amplitude gradually by a factor of .03125.

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 sound file *gtr512.aiff* (or some similar waveform) into a **buffer~**, and then refer to that **buffer~** with a **cycle~**. (See *Tutorial 3*.)

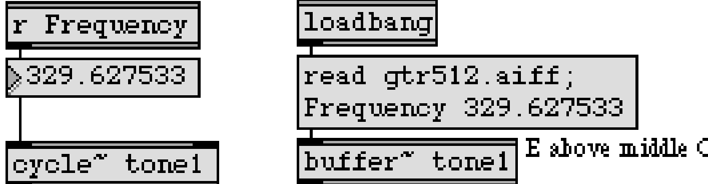


cycle~ *traverses the* **buffer~** *329.627533 times per second*

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.

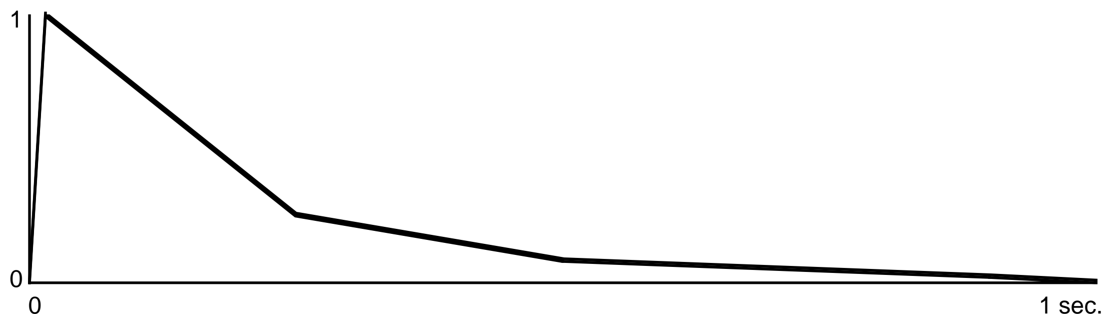
If you want the sound file to be read into the **buffer~** immediately when the patch is loaded, you can type the filename in as a second argument in the **buffer~** object, or you can use

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**.



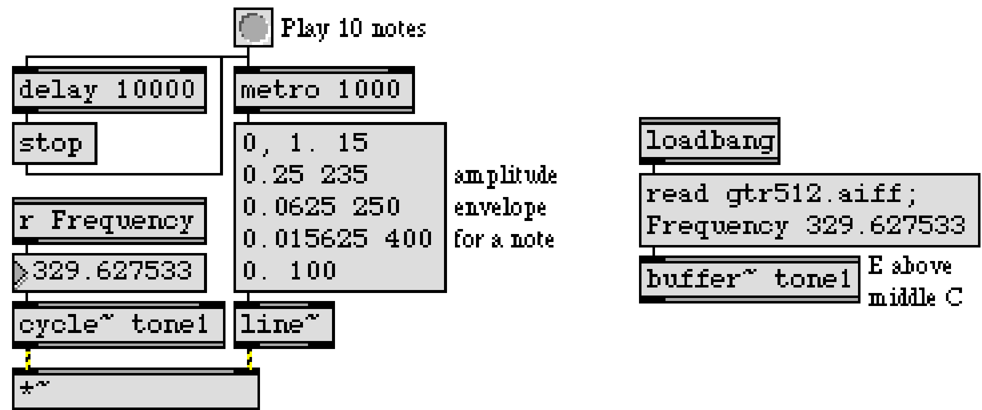
loadbang *is used to initialize the contents of* **buffer~** *and the frequency of* **cycle~**

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

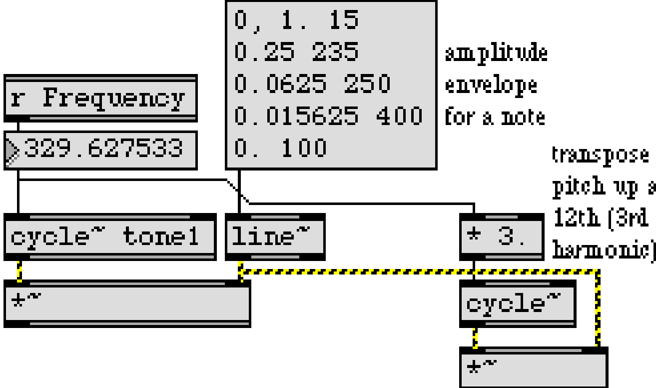
This amplitude envelope is imposed on the output of **cycle~** with a combination of **line~** and *~. A **metro** is used to trigger the envelope once per second, and the **metro** gets turned off after a 10-second delay.



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

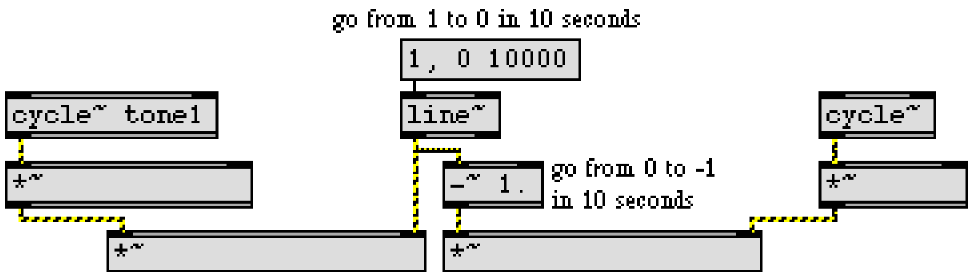
Solution to Exercise 2

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.



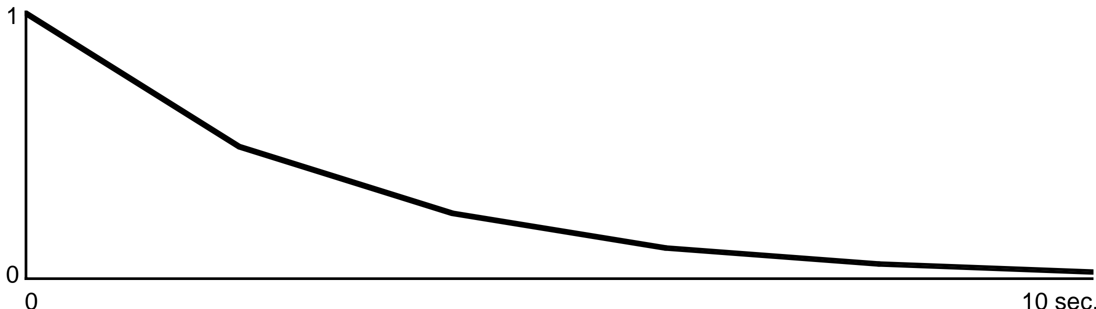
Linear crossfade of two tones

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.

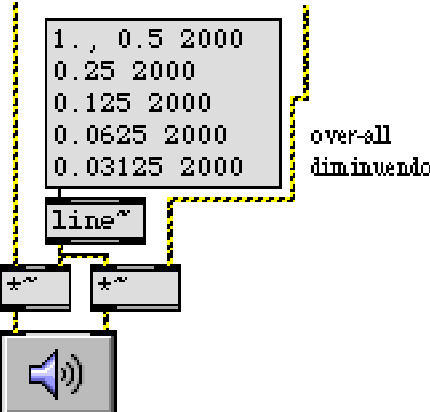
Solution to Exercise 3

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.

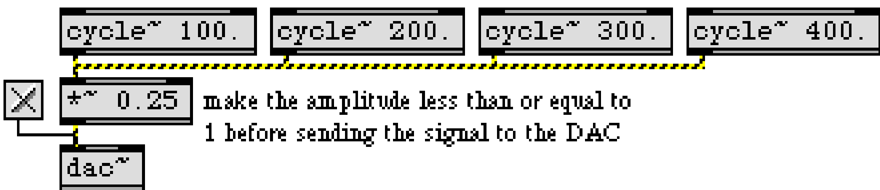


Both tones are scaled by the same global envelope

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

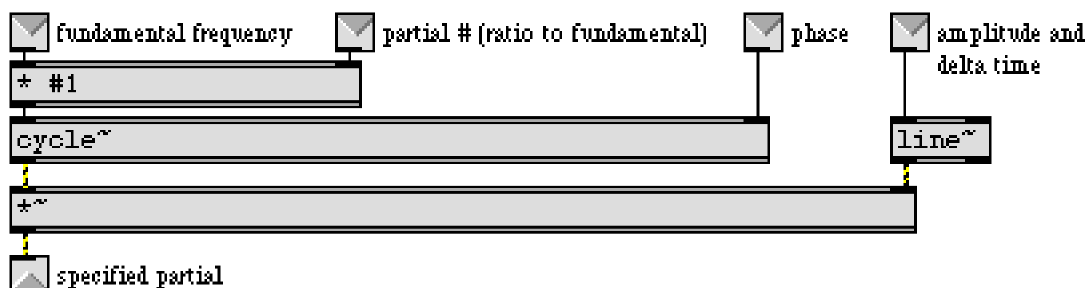
A sine wave contains energy at a single frequency. Since complex tones, by definition, are composed of energy at several (or many) different frequencies, one obvious way to synthesize complex tones is to use multiple sine wave oscillators and add them together.



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~**

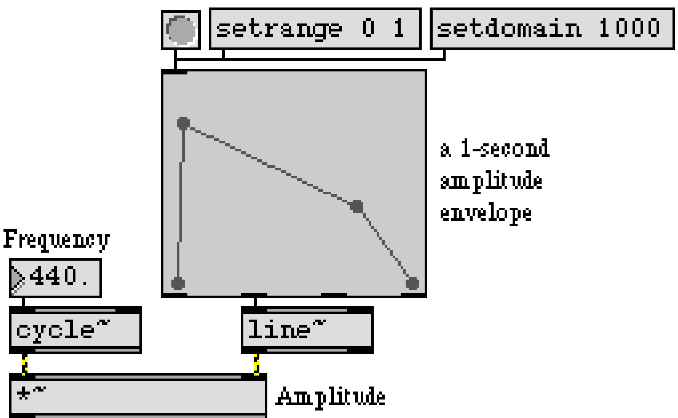
For example, if we want a partial to have a frequency twice that of the fundamental we just type in 2.0 as an argument (or send it in the second inlet). This way, if several **partial~** objects are receiving their fundamental frequency value (in the left inlet) from the same source, their relative frequencies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

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.

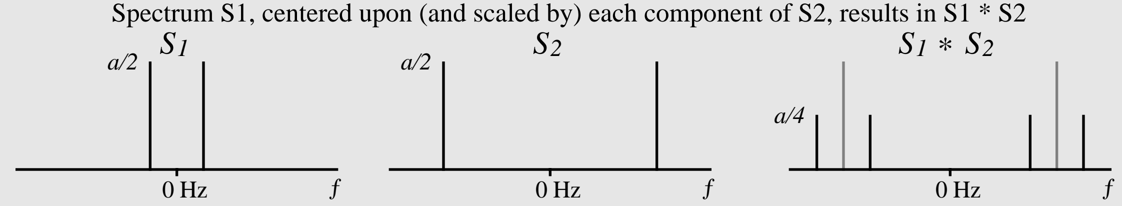
In order for a complex tone to have an interesting timbre, the amplitude of the partials must change with a certain degree of independence. The **function** object allows you to draw control shapes such as amplitude envelopes, and when **function** receives a bang it describes that shape to a **line~**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

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 S_1 and S_2, each of which contains many different frequencies all at different amplitudes, make a copy of S_1 at the location of every frequency in S_2, with each copy scaled by the amplitude of that particular frequency of S_2.

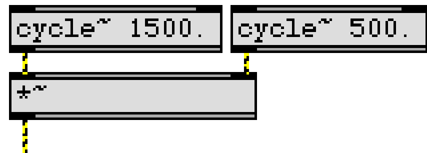
Since a cosine wave has equal amplitude at both positive and negative frequencies, its spectrum contains energy (equally divided) at both *f* and *-f*. When convolved with another cosine wave, then, a scaled copy of (both the positive and negative frequency components of) the one wave is centered around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



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

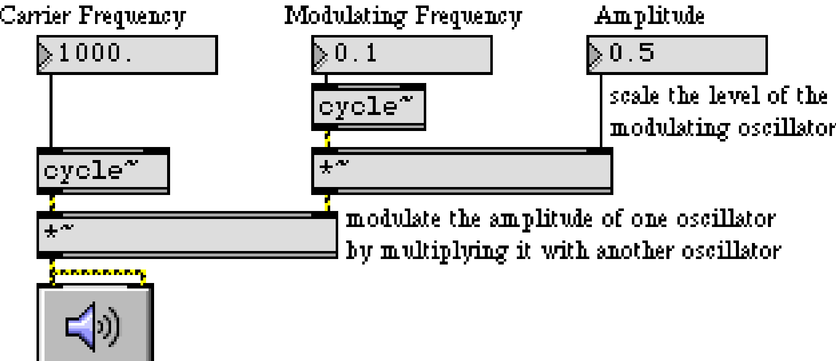
So, in this simple case where each signal is a cosine wave, and thus each spectrum consists of energy at only one frequency (positive and negative), the resulting spectrum consists of the sum and the difference of the frequency of the two waves.

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** boxes 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**es. When you have finished, click on **ezdac~** again to turn audio off.

Summary

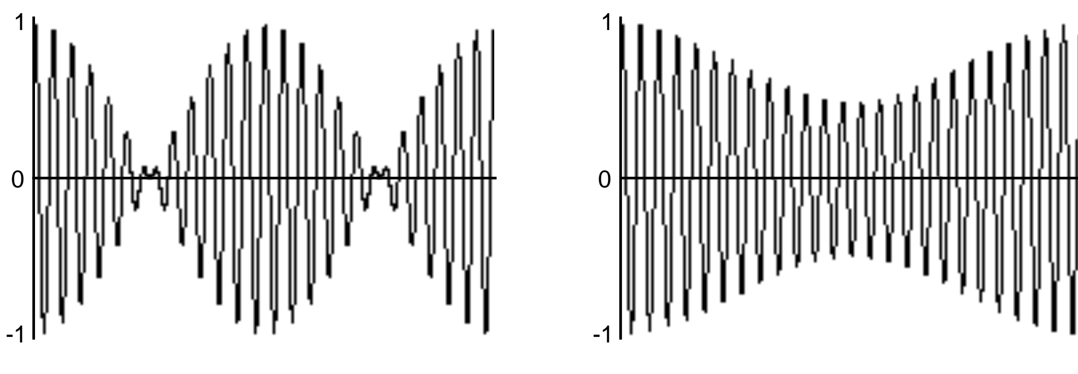
Multiplication of two digital signals is comparable to the analog audio technique known as *ring modulation*. Ring modulation is a type of *amplitude modulation*—changing the amplitude of one tone (termed the *carrier*) with the amplitude of another tone (called the

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.

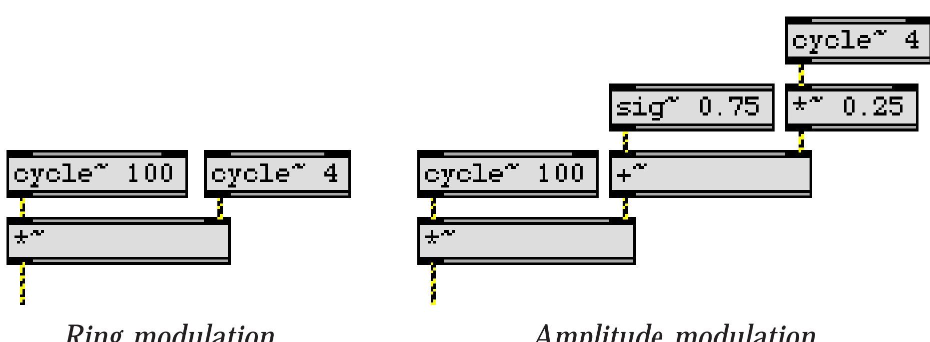
Ring modulation and amplitude modulation

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.



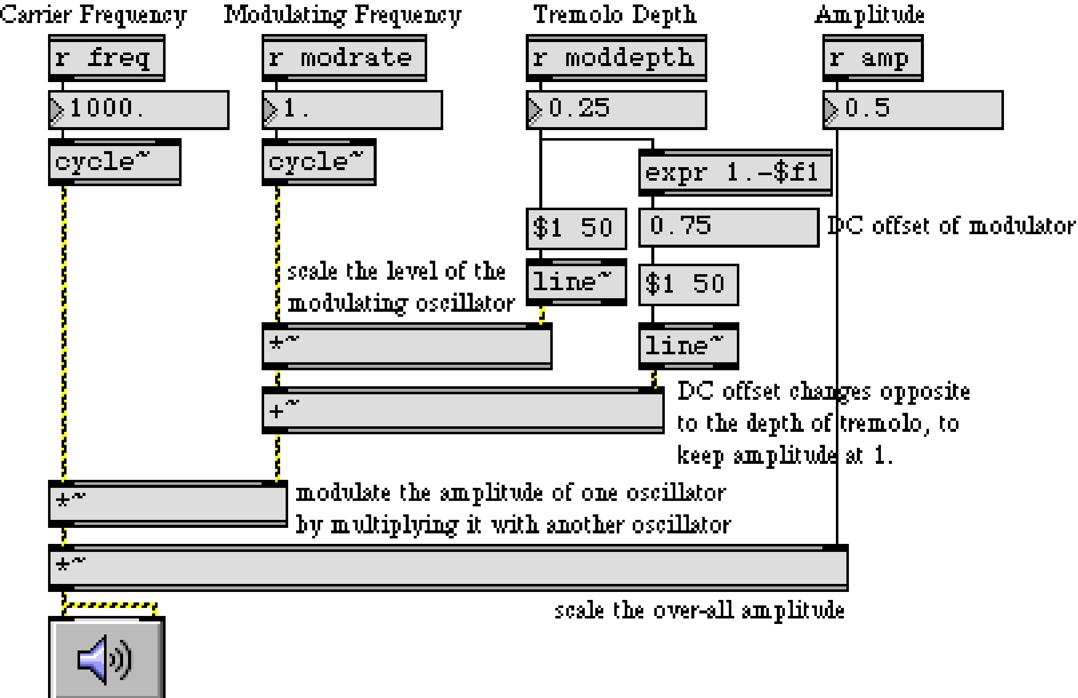
Ring modulation *Amplitude modulation*

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**es. When you have finished, click on the **ezdac~** to turn audio off.

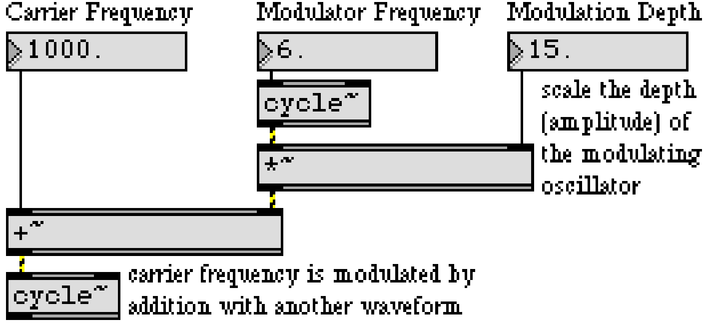
It is worth noting that *any* audio signals can be used as the carrier and modulator tones, and in fact many interesting results can be obtained by amplitude modulation with complex tones. (*Tutorial 22* allows you to perform amplitude modulation on the sound coming into the computer.)

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.

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 (F_c) and the modulator frequency (F_m); the relative strength of the different sidebands (which affects the timbre) is determined by the relationship between the modulator amplitude (A_m) and the modulator frequency (F_m).

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 F_m/F_c; 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 A_m/F_m; this value affects the "brightness" of the timbre by affecting the relative strength of the partials.

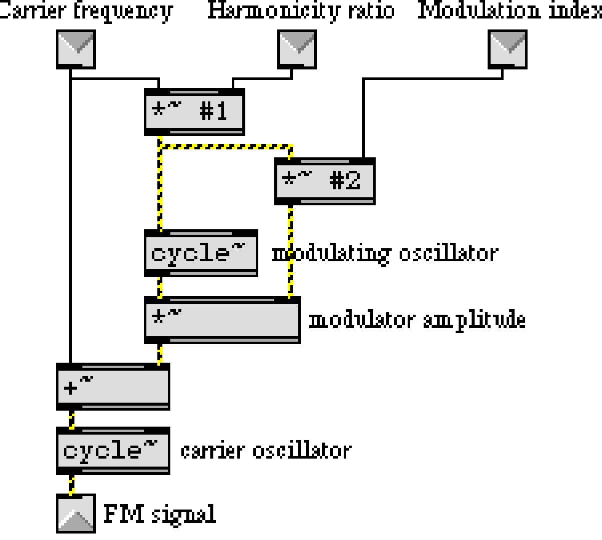
> ***Technical detail***: In John Chowning's article "Synthesis of Complex Audio Spectra by Means of Frequency Modulation" and in Curtis Roads' *Computer Music Tutorial*, they write about the ratio F_c/F_m. However, in F.R. Moore's *Elements of Computer Music* he defines the term *harmonicity ratio* as F_m/F_c. 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 F_c as 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 F_c, F_c+F_m, F_c-F_m, F_c+2F_m, F_c-2F_m, F_c+3F_m, F_c-3F_m, 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.



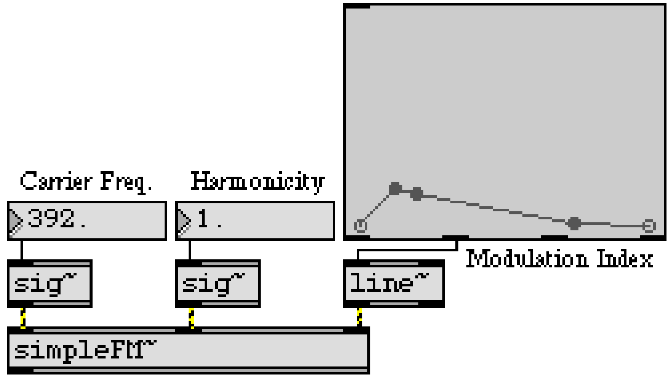
The **simpleFM~** *subpatch*

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

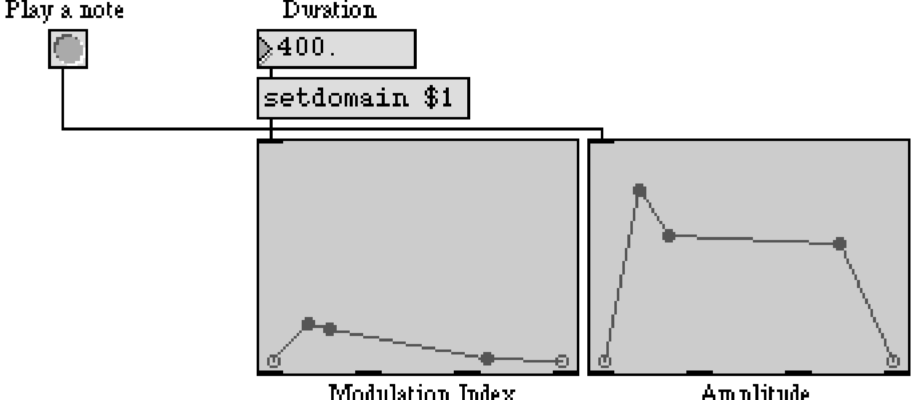
In the main patch, the carrier frequency and harmonicity ratio are provided to **simpleFM~** as constant values, and the modulation index is provided as a time-varying signal generated by the envelope in the **function** object.



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 slowish 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.

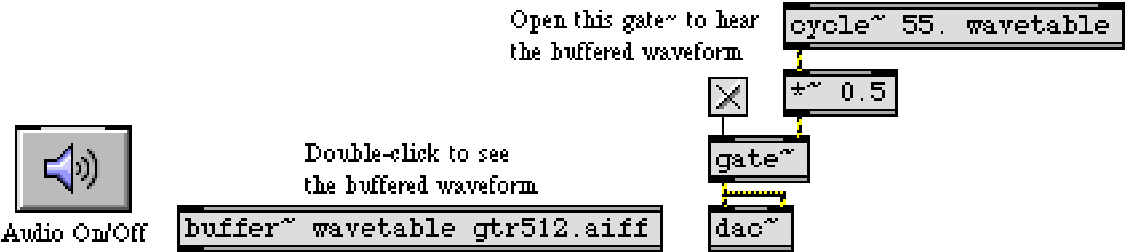
The frequencies present in an FM tone are equal to the carrier frequency plus and minus integer multiples of the modulator frequency. Therefore, the harmonicity of the tone can be described by a single number—the ratio of the modulator and carrier frequencies—sometimes called the *harmonicity ratio*. 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~***

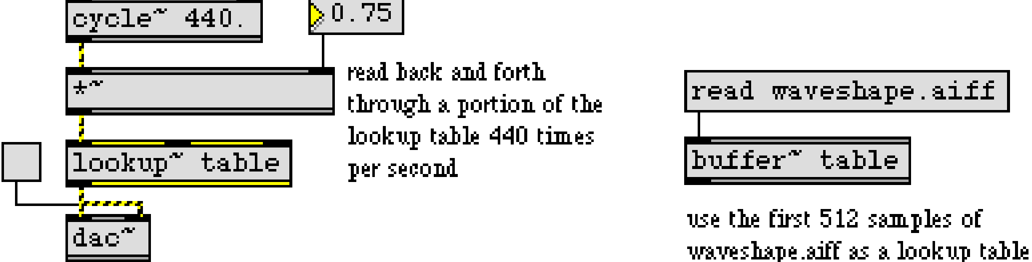
- Double-click on the **buffer~** object to see its contents. The file *gtr512.aiff* contains one cycle of a recorded electric guitar note. Click on the **ezdac~** speaker icon to turn audio on. Click on the **toggle** to open the **gate~**, allowing the output of **cycle~** to reach the **dac~**. Click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

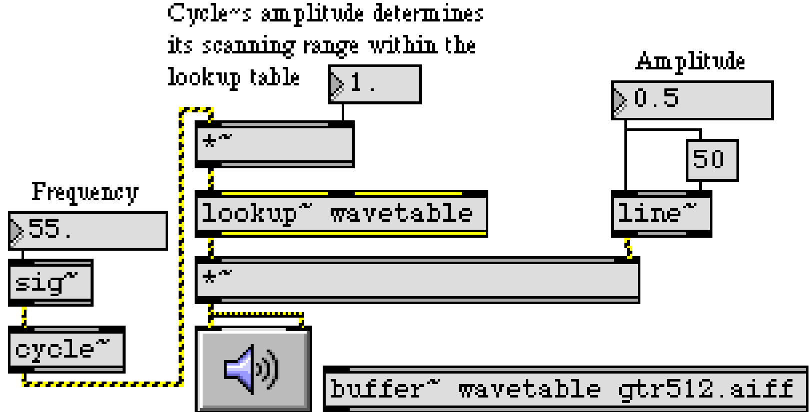
The most commonly used input signal for indexing the lookup table is a sine wave—it's a reasonable choice because it reads smoothly back and forth through the table—but any audio signal can be used as input to **lookup~**.

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

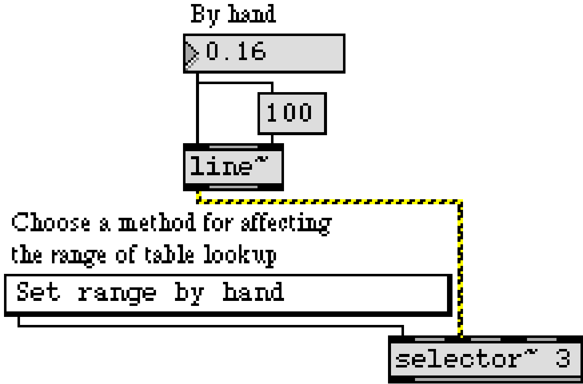
The waveshaping part of the tutorial patch is in the lower left portion of the Patcher window. It's very similar to the example shown above. The lookup table consists of the 512 samples in the **buffer~**, and it is read by a cosine wave from a **cycle~** object.



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.



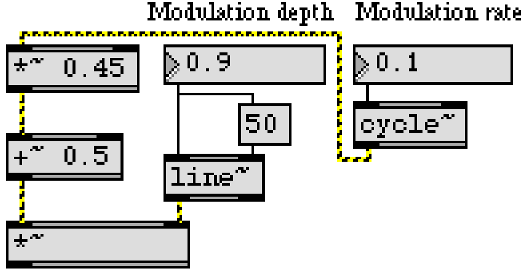
Set the amplitude of the input signal to change the timbre of the output

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.

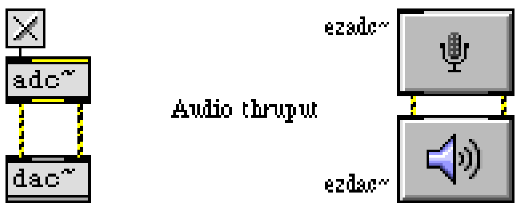
If you’re designing an instrument for musical purposes, you might use some combination of these three ways to vary the timbre, and you almost certainly would have an independent amplitude envelope to scale the amplitude of the output sound. (Remember that the amplitude of the signal coming out of **lookup~** 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.

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~**.

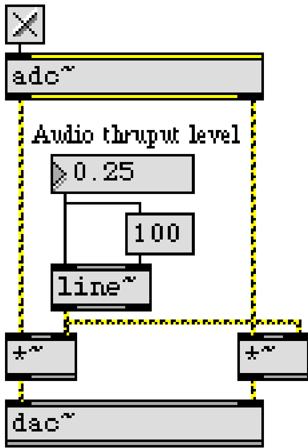


***adc~** and **ezadc~** get sound from the audio input jacks and send it out as a signal*

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—either a Macintosh microphone, or a standard microphone via a preamplifier.

- Double click on the **adc~** object to open the DSP Status window. Make sure that the *Input Source* popup menu displays the input source you want. In most cases you'll have the choice of Internal CD or Microphone (meaning whatever is coming in the input jacks). If you're using an audio card and one of the MSP audio drivers, you will not have a choice of input source. Close the DSP Status window.

- 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*.

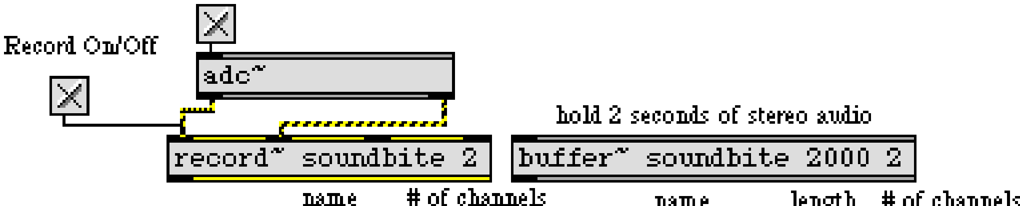


Adjust the audio thruput to a comfortable listening level

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~

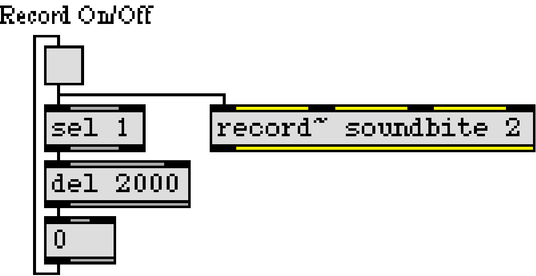
To record a sample of the incoming sound (or any signal), you first need to designate a buffer in which the sound will be stored. Your patch should therefore include at least one **buffer~** object. You also need a **record~** object with the same name as the **buffer~**. The sound that you want to record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

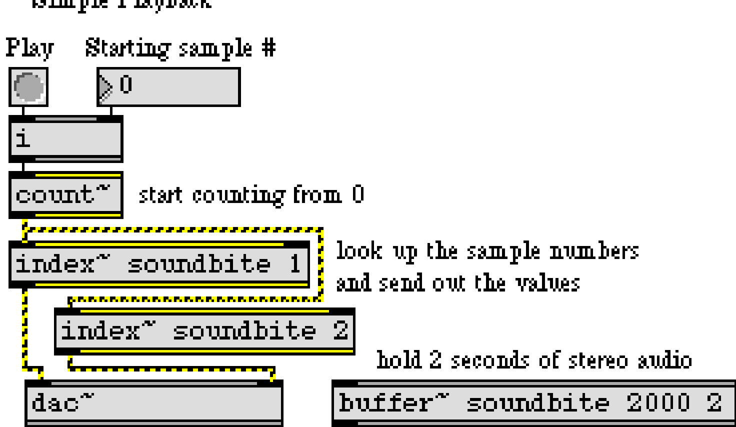
- Make sure that you have sound coming into the computer, then click on the **toggle** to record two seconds of the incoming sound. If you want to, you can double-click on the **buffer~** afterward to see the recorded signal.

Reading through a buffer~: index~

So far you have seen two ways to get sound into a **buffer~**. You can read in an existing AIFF or Sound Designer II 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 a sound 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~**.

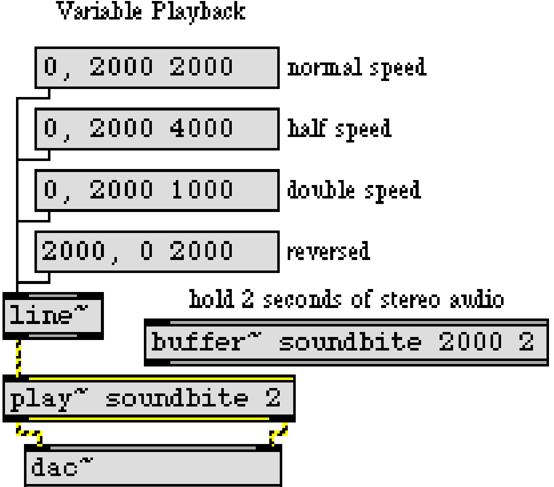
This combination of **count~** and **index~** lets you specify a precise sample number in the **buffer~** where you want to start playback. However, if you want to specify starting and ending points in the **buffer~** in terms of milliseconds, and/or you want to play the sound back at a different speed—or 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**es to hear the sound played in various speed/direction combinations. Turn audio off when you have finished.

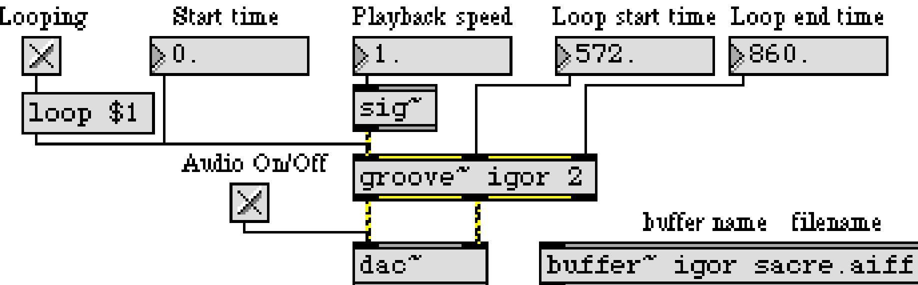
Although not demonstrated in this tutorial patch, it's worth noting that you could use other signals as input to **play~** in order to achieve accelerations and decelerations, such as an exponential curve from a **curve~** object or even an appropriately scaled sinusoid from a **cycle~** object.

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.

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.

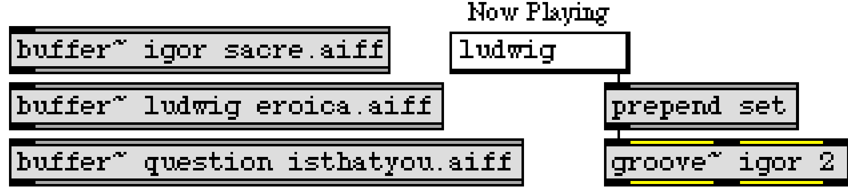


A standard configuration for the use of **groove~**

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.

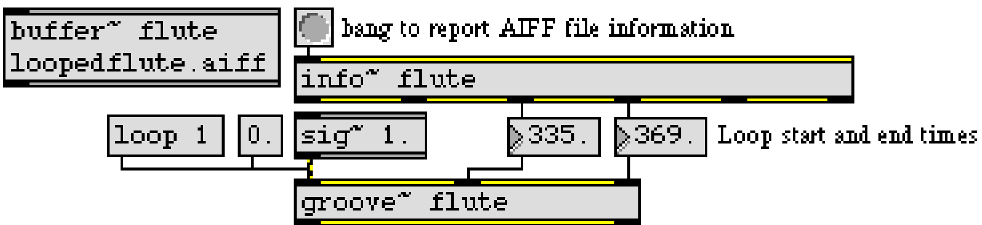
The first preset just functions as an “Off” button. The next three presets play the three **buffer~**s at normal speed without looping. The rest of the presets demonstrate a variety of sound possibilities using different playback speeds on different excerpts of the buffered files, with or without looping.

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

Now Playing | Looping | Start time | Playback speed | Loop start time | Loop end time | Amplitude
question | | 0. | 4. | 487. | 586. | 0.67

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

Smooth undetectable loops are difficult to achieve with **groove~**. Most commercial synthesizers perform a crossfade 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. The looping mode of **groove~** does not perform a crossfade, so it is better suited for the rhythmic repetitive effects demonstrated here. However, 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*

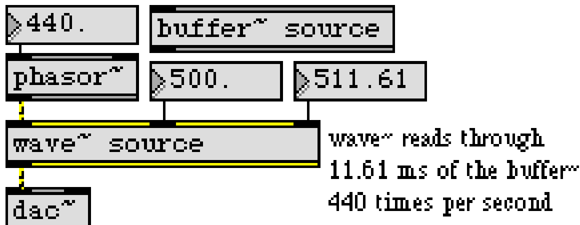
Summary

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.

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~**.



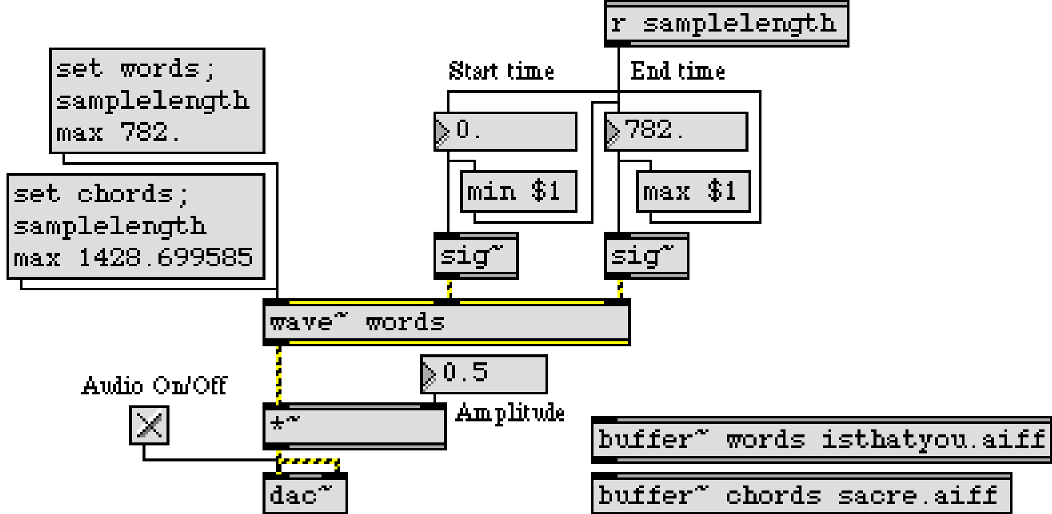
phasor~ *drives* **wave~** *through the section of the* **buffer~** *specified as the wavetable*

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.

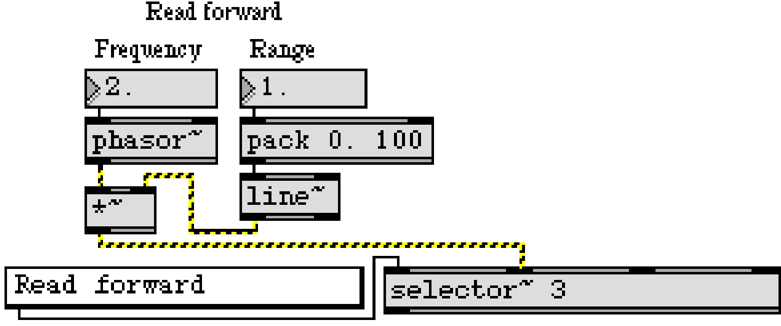


wave~ *can use an excerpt of any length from either* **buffer~** *as its wavetable*

- 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.

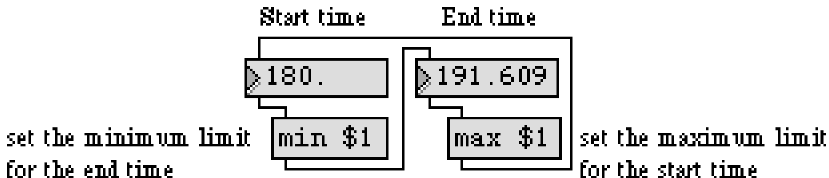
- Try a few different sub-audio frequency values for the **phasor~**, to read through the **buffer~** at different speeds. You can change the portion of the **buffer~** being read,

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~**.



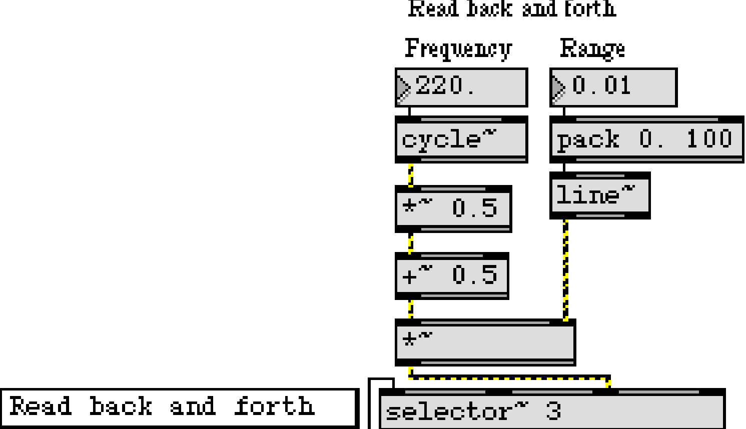
Each time the start or end time is changed, it revises the limits of the other **number box**

- 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 **cycle~**'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.

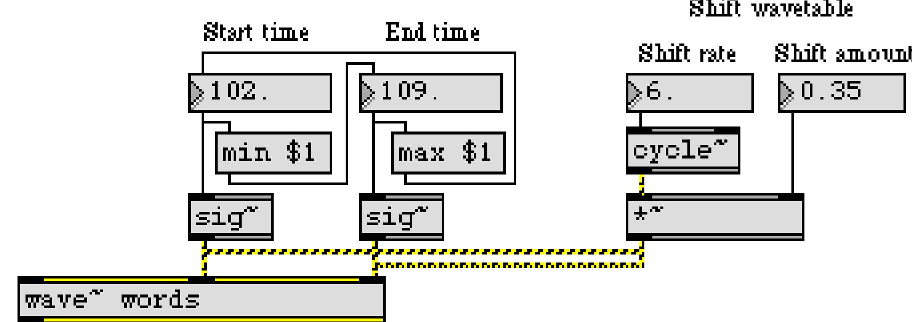
Play the segment as a note

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.

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 AIFF or Sound Designer II 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.

In MSP, playing sound files from disk is appropriate only for forward playback at normal speed, but you still have very flexible control over what file you play, what portion of the file you play, and when you play it.

Record sound files: sfrecord~

MSP has objects for recording directly into, and playing directly from, an AIFF file: **sfrecord~** and **sfplay~**. Recording a sound 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.

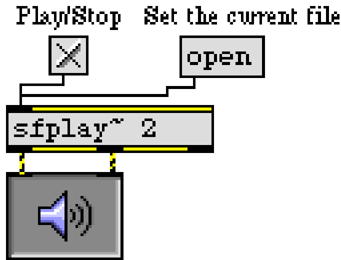
Recording audio into a sound file on disk

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 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 sound files: sfplay~

In the left part of the Patcher window there is a patch for playing sound 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 sound file playback

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

Play excerpts on cue

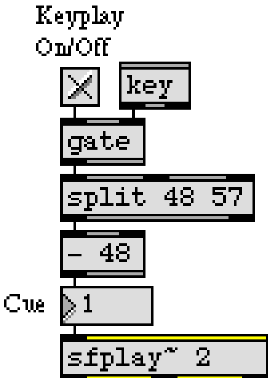
Because **sfplay~** does not need to load an entire sound file into memory, you can actually have many files open in the same **sfplay~** object, and play any of them (or any portion of them) on cue. The most recently opened file is considered by **sfplay~** to be the “current” file, and that is the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

- Click on the remaining open **message** boxes to open some other sound 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 Macintosh 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 sound file again (or any other sound 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 sound files to and from the hard disk. The **sfplay~** object can have many sound 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.

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 Macintosh keyboard turns audio on. Typing *a* again toggles audio off.
- Typing *r* on the Macintosh 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

Because you need to play the sample back at half-speed, **sfplay~** (which only plays at normal speed) is not the correct choice. 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*.)

The Macintosh 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.

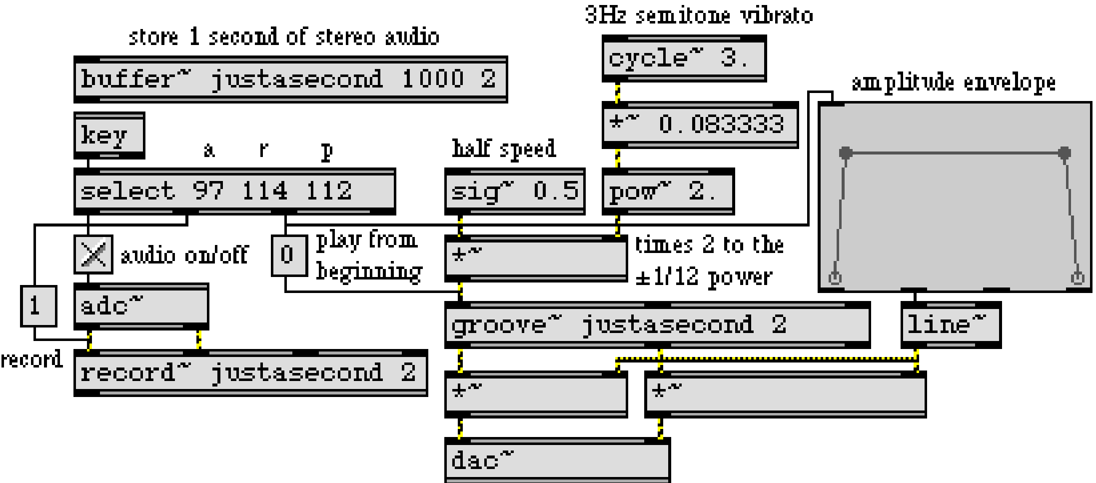
Use a sinusoidal wave from a **cycle~** object to apply vibrato to the sample. The frequency of the **cycle~** will determine the *rate* of the vibrato, and the amplitude of the sinusoid will determine the *depth* of vibrato. Therefore, you will need to scale **cycle~**'s amplitude with a ***~** object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

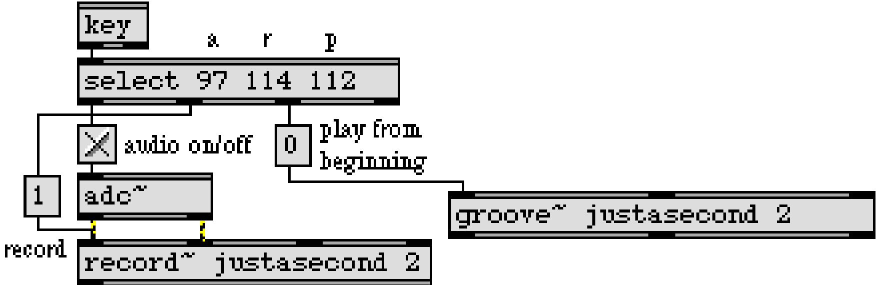
store 1 second of stereo audio

buffer~ justasecond 1000 2

Set name, length, and channels of the **buffer~**

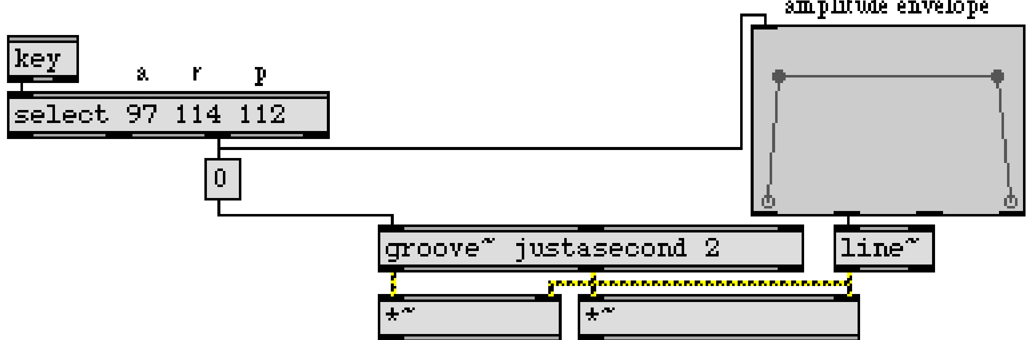
Since the memory allocated in the **buffer~** is limited to one second, there is no need to tell the **record~** object to stop when you record into the **buffer~**. It stops when it reaches the end of the **buffer~**.

The keystrokes from the Macintosh 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~**.



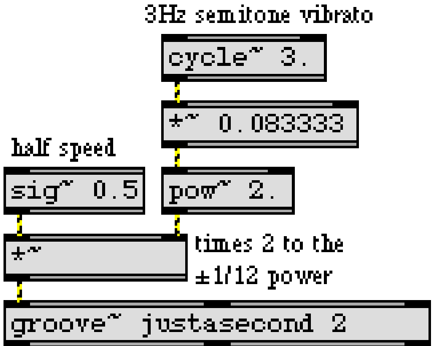
Keystrokes are detected and used to send messages to MSP objects

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

MIDI range vs. MSP range

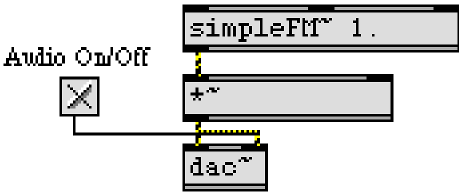
One of the greatest assets of MSP is the ease with which one can combine MIDI and digital signal processing. The great variety of available MIDI controllers means that you have many choices for the instrument you want to use to control sounds in MSP. Because Max is already a well developed environment 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 *Max Tutorial 29* , 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 *MSP 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

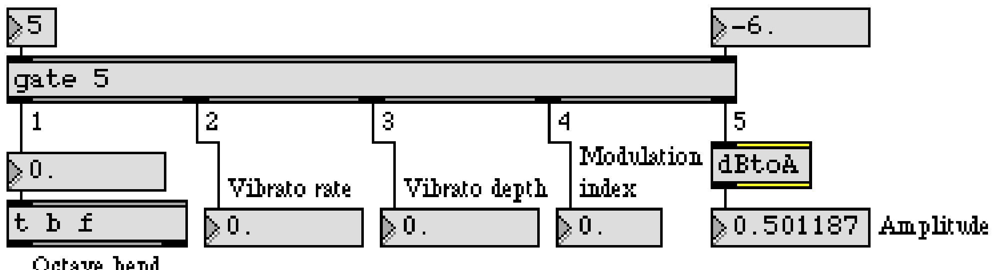
If we were designing a real performance instrument, we would probably control each of these parameters with a separate type of MIDI message—controller 7 for amplitude, controller 1 for vibrato depth, pitchbend for pitch bend, and so on. In this patch, however, we use the mod wheel controller for everything, 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 Macintosh keyboard can be used to choose an item in the pop-up **umenu** at the top of the window.

Use ASCII from the Macintosh 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:

$$y = ((x - xmin) * (ymax - ymin) \div (xmax - xmin)) + ymin$$

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*

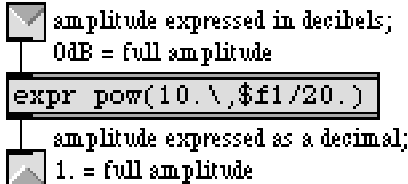
Once we have scaled the range of control values with **map**, some additional mapping may be necessary to suit various signal processing purposes, as you will see.

Mapping MIDI to amplitude

As noted in *MSP Tutorial 4*, we perceive relative amplitude on a multiplicative rather than an additive scale. For example we hear the same relationship between an amplitudes 0.5 and 0.25 (a factor of 1/2, but a difference of 0.25) as we do between amplitudes 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 *MSP Tutorial 17* and *MSP 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 2^0 to 2^1

$2^0 = 1$, 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.

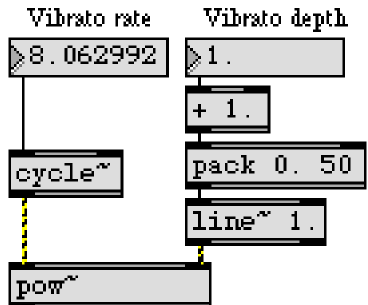
Mapping MIDI to modulation index

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.

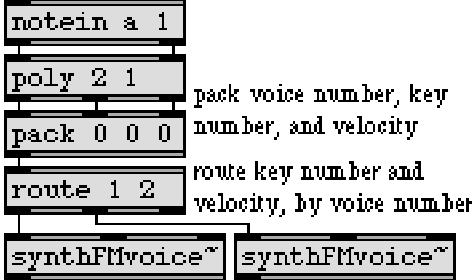
Implementing standard MIDI messages

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 first, converting a MIDI key number into the proper equivalent frequency; second, converting a MIDI pitch bend value into an appropriate frequency-scaling factor; and third, 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.



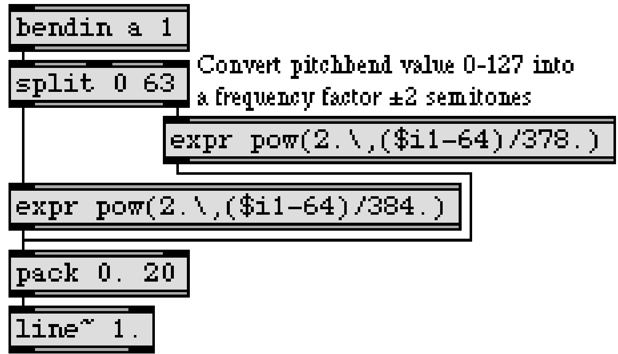
Assign a voice number with **poly** *to play polyphonic music*

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.)

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.

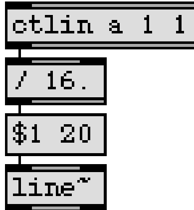


Downward bend is calculated slightly differently from upward bend

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

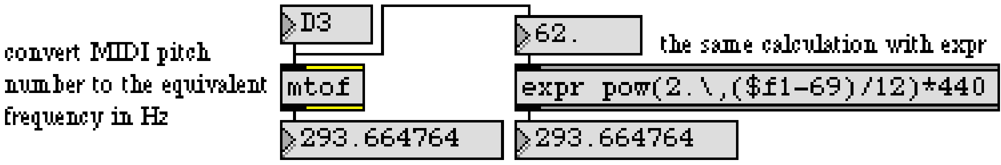
The FM synthesizer

- 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 *MSP 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.

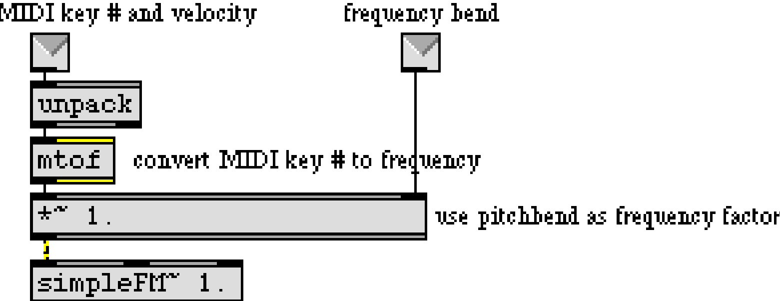
MIDI-to-frequency conversion

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.

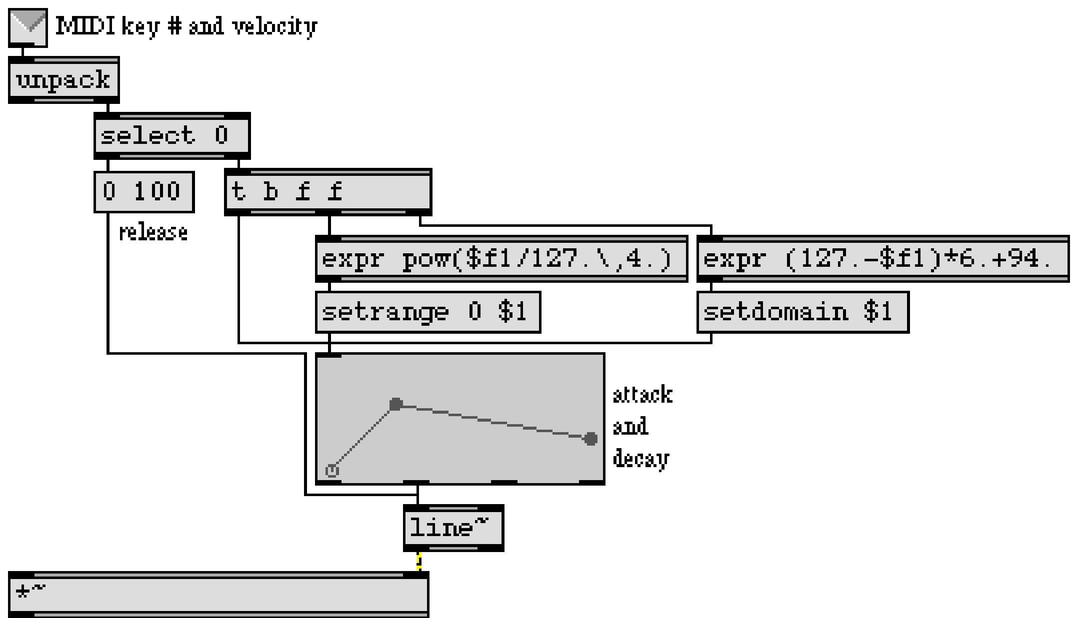


The frequency of the note calculated from key number and pitch bend data

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 *1*) separate note-on velocities from note-off velocities, *2*) 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), and in this case *3*) map note-on velocity to rate of attack and decay of the envelope.

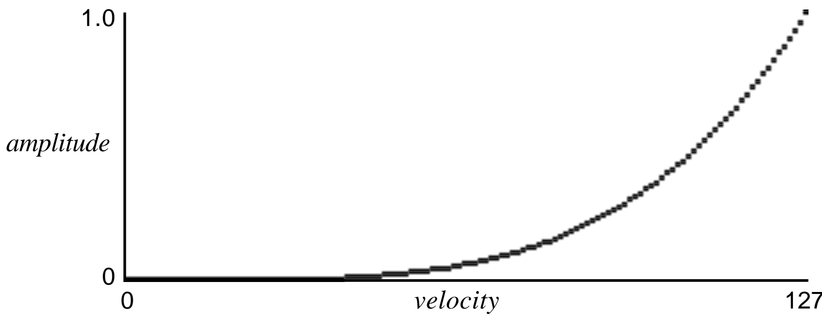
The first task is achieved easily with a **select** 0 object, so that note-on velocity triggers a **function** object to send the attack and decay shape, and note-off velocity returns the amplitude to 0, as shown in the following example.



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.

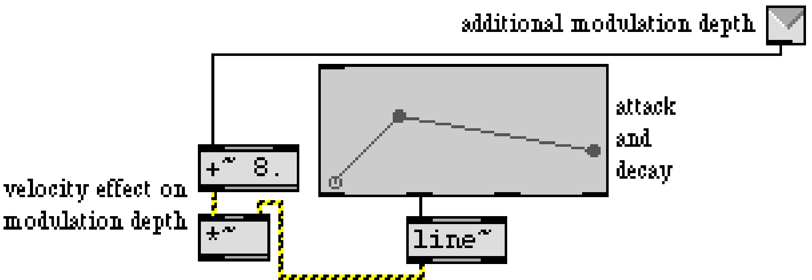


Velocity mapped to amplitude with an exponent of 4

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.

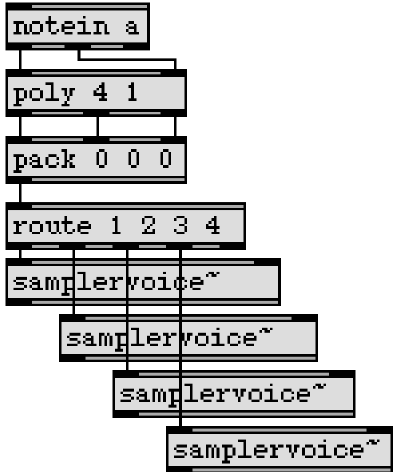
Note-on velocity is generally used to determine the amplitude of the note, and triggers the attack portion of the amplitude envelope. The note-off message triggers the release portion of the envelope. The velocity value can be used to alter the range of the envelope (or to provide a factor for scaling 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.

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 *MSP 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 *MSP 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.

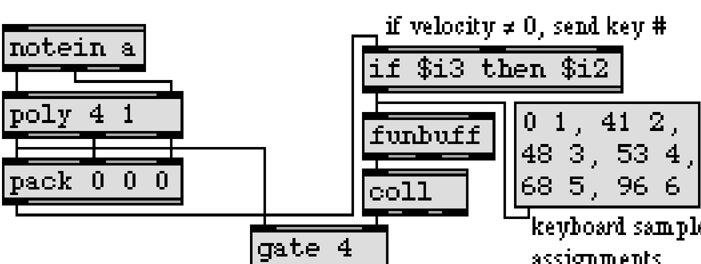
In this patch, the samples are all read into memory when the patch is first loaded.

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

You can see that six samples have been loaded into **buffer~**s 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~**s 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;
```

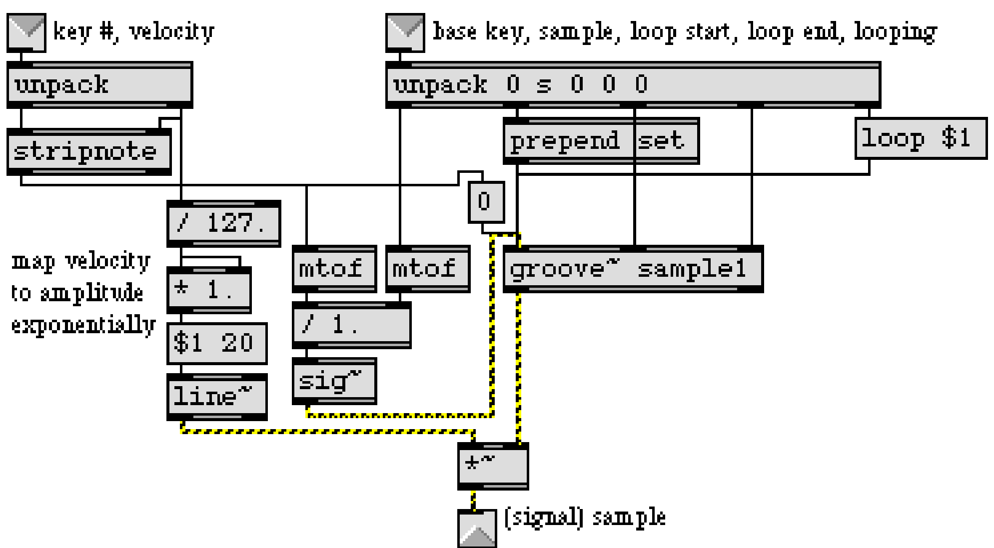
coll *contains sample information for each key region*

The key region number is used to index the information in **coll**. For example, whenever a key from 53 to 67 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. soundfile sample rate

- Close the subpatch window.

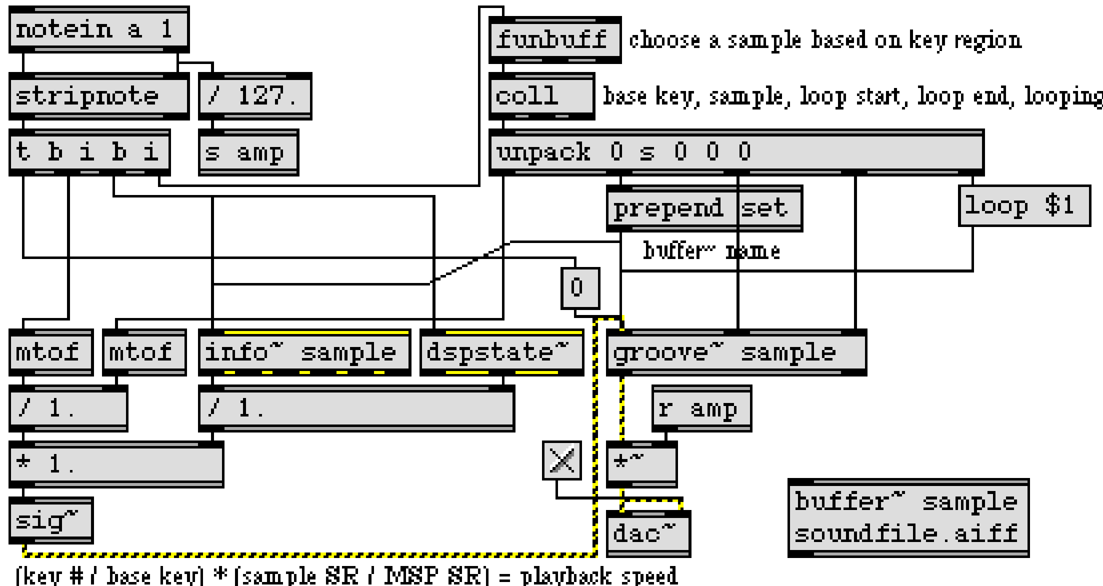
You're almost ready to begin playing samples, but there is one more detail to attend to first. To save storage space, the samples used in this patch are mono AIFF files with a sample rate of 22,050 Hz. To hear them play properly you should set the sample rate of MSP to that rate.

- 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.

The difference between the sample rate of a soundfile 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 soundfiles 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 soundfile 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 soundfile 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 soundfile 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 soundfile 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~**.

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**.

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*.

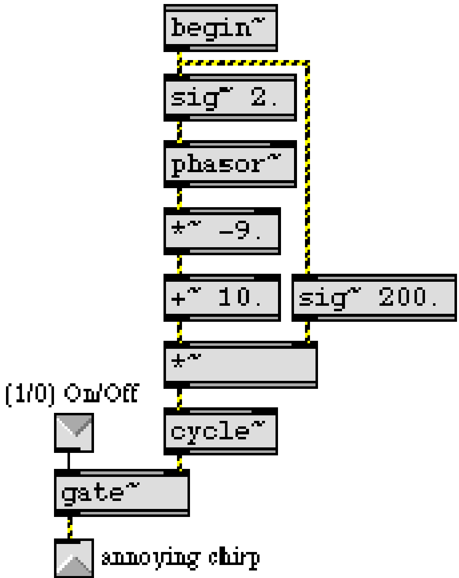
This tutorial demonstrates three ways of calculating panning, controllable by MIDI values 0 to 127. You can try out the three methods and decide which is most appropriate for a given situation in which you might want to control panning.

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.

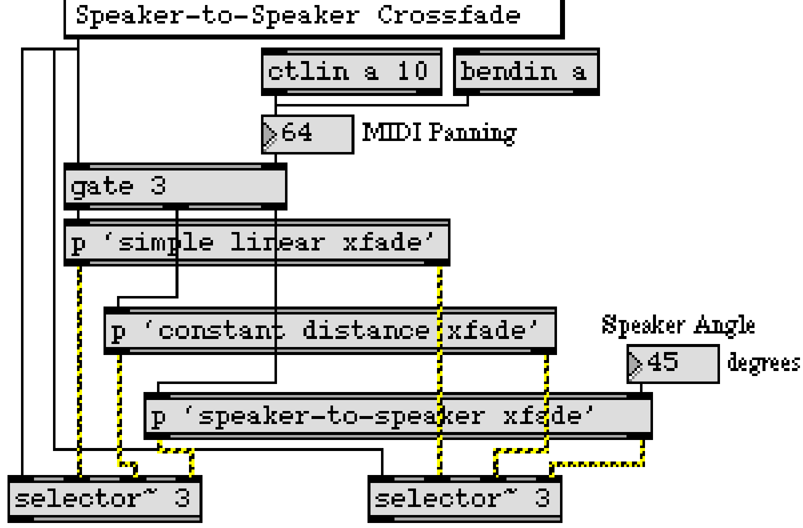
Because of the **gate~** and **begin~** objects, audio processing is off in this subpatch until a 1 is received in the inlet to open the **gate~**. At that time, the **phasor~** generates a linear frequency glissando going from 2000 to 200 two times per second.



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 an 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".)

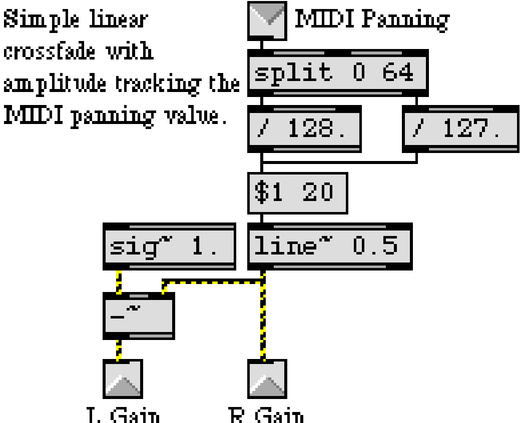


Selection from the **umenu** *opens input and output for one of the three panning subpatches*

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 ($^{64}/_{127} \neq 0.5$). 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

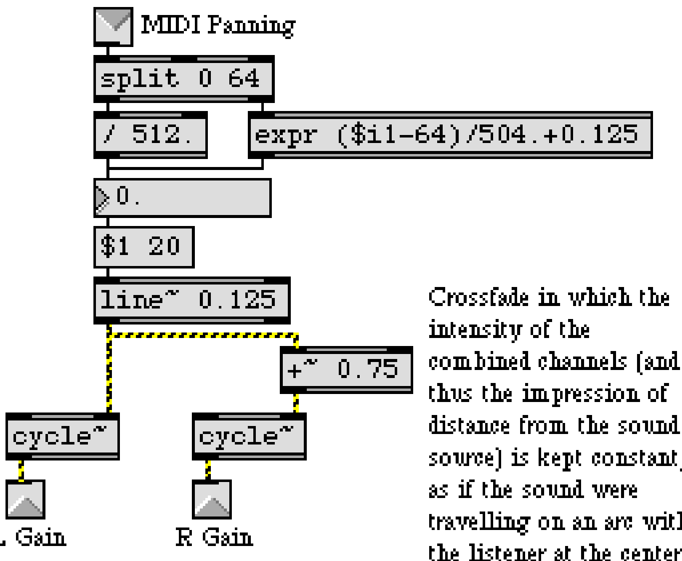
If we can maintain a constant intensity as we pan from one side to the other, this will give the impression that the sound source is maintaining a constant distance from the listener.

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

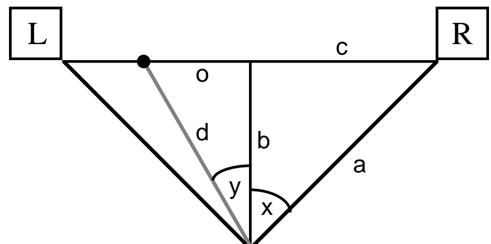
Once again we need to treat MIDI values greater than 64 differently from those less than or equal to 64, in order to retain 64 as the "center" of the range. Once the MIDI value is mapped into the range 0 to 0.25, the result is used as a phase angle two **cycle~** objects, one a cosine and the other (because 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. The 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.

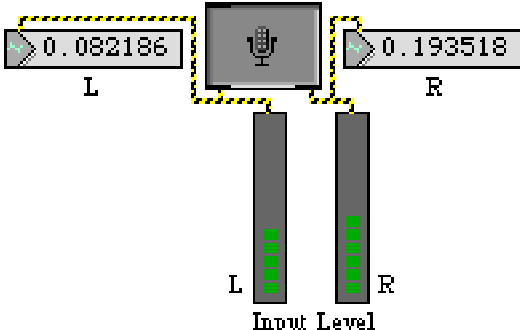
When a sound moves past the listener in a straight line, it is loudest when it passes directly in front of the listener. To emulate straight line movement, one can calculate the relative distance of the sound source as it travels, and modify the amplitude of each channel (and the overall intensity) accordingly.

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.

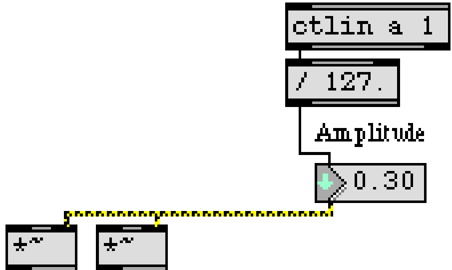


Current signal value is shown by **number~***; peak amplitude is shown by* **meter~**

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.

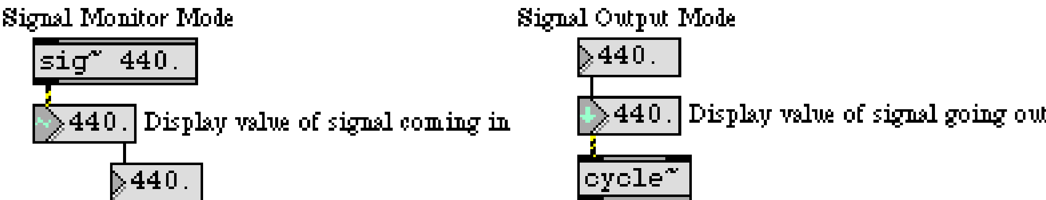
The **number~** object is actually like two objects in one. In addition to receiving signal values and sending them out the right outlet as a float, **number~** also functions as a floating point **number box** that sends a signal (instead of a float) out its left outlet.

- 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 **number~**'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~**.



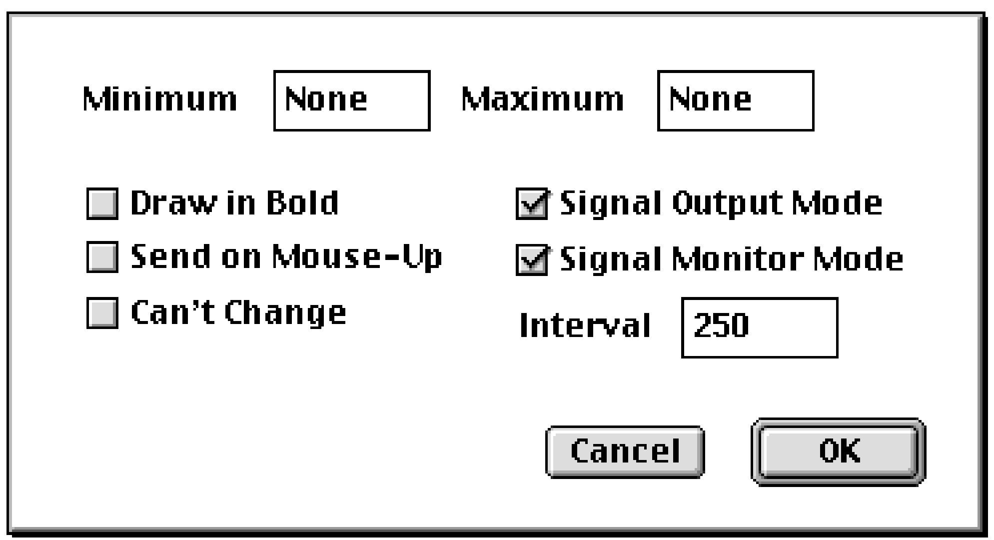
float *input to* **number~** *sets the value of the signal sent out the left outlet*

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 Max menu.



Allowed display modes can be chosen in the ***Get Info…*** *dialog*

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.

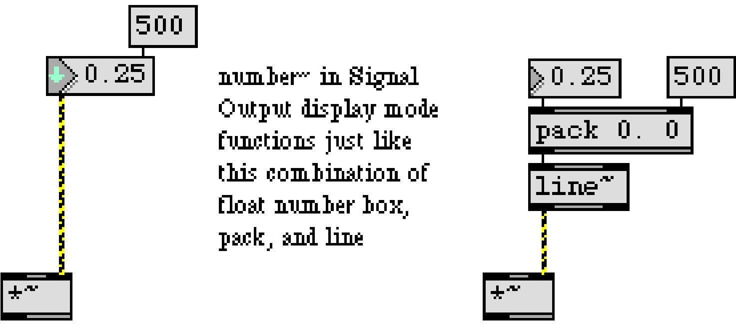
In the tutorial patch you can see the two display modes of **number~**. The **number~** objects at the top of the Patcher window are in *Signal Monitor Mode* because we are using them to show the value of the incoming signal. The "Amplitude" **number~** is in *Signal Output Mode* because we are using 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 **Get Info…** dialog.

- 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 **Get Info…**.

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~**.

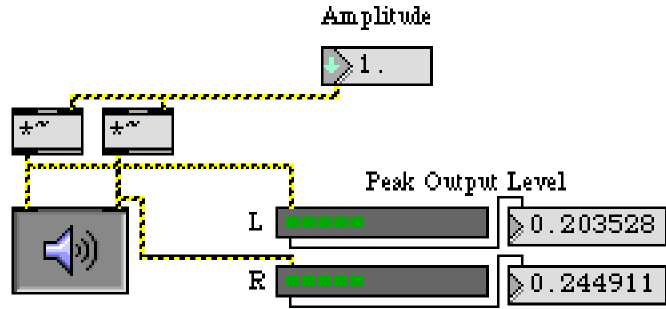


number~ *can send a linear ramp signal from its old value to a new value*

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 **Get Info…** dialog).

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**es 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), but there is a limit of 32,000 on how many characters can be displayed in a **capture~** text window. There are various arguments and messages for controlling exactly what will be stored by **capture~**. See its description in the *Objects* section 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.

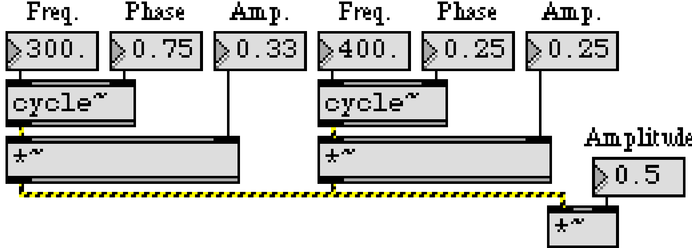
Graph of a signal over time

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 ints in the inlets of **scope~**. The sampling rate of MSP can be obtained with the **dspstate~** object.

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.

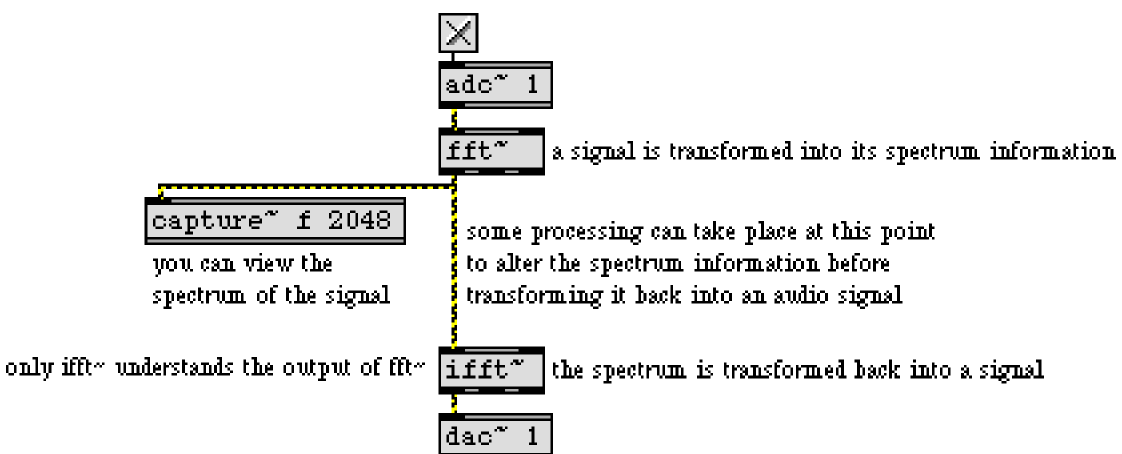
Typically the Fourier transform is used on a small "slice" of time, which ideally is equal to exactly one cycle of the wave being analyzed. To perform this operation on "real world" sounds—which are almost invariably *not* strictly periodic, and which may be of unknown frequency—one can perform the DFT 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.

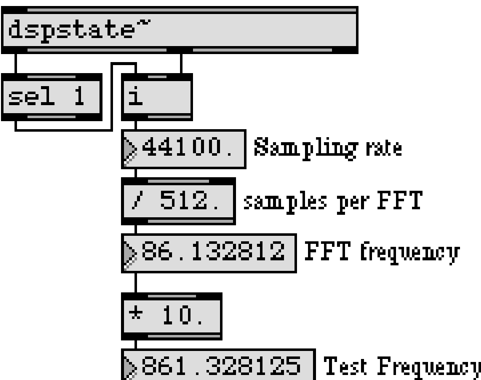


The signal coming out of **fft~** *is spectral information, not a time-domain signal*

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.



The test tone is at 10 times the base frequency of the FFT time slice

In 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 **fft~**'s analysis shows no signal.

- Scroll past the first 512 numbers. (The numbers in **capture~**'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.

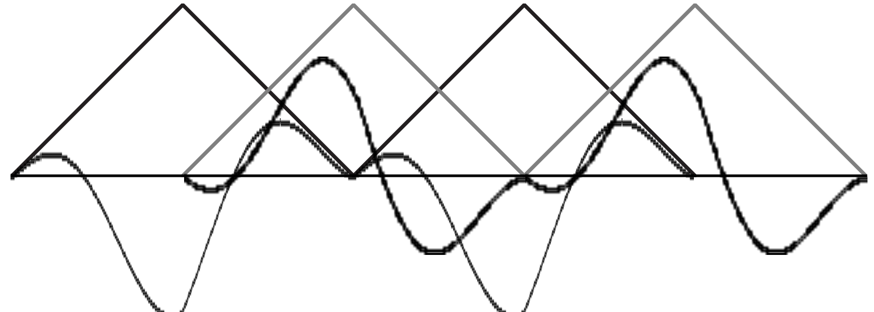
- Close the text window of **capture~**. With the audio still on, set the "Test Frequency" **number box** to 1000. This also triggers the clear message in the upper left corner of the patch to empty the **capture~** object of its prior contents. Double-click once again on **capture~**, and scroll ahead 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.

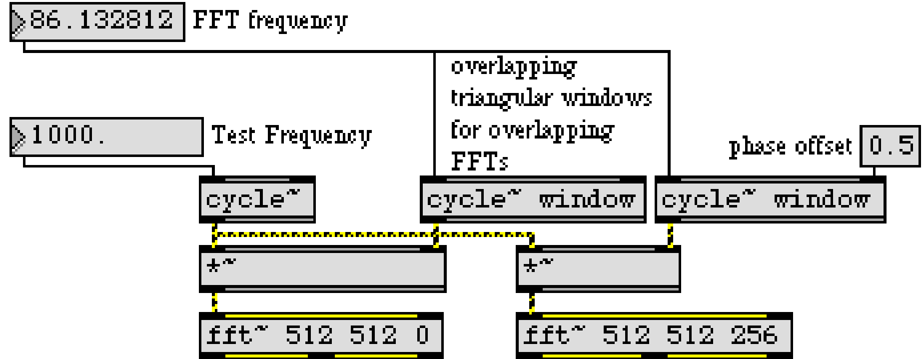
Overlapping FFTs

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).

Effects achieved with delayed signals

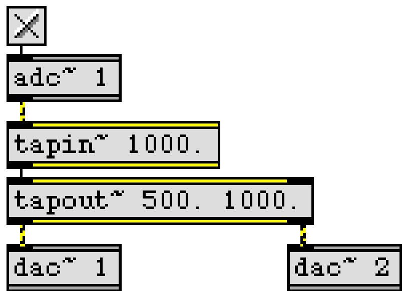
One of the most basic yet versatile techniques of audio processing is to delay a signal and mix the delayed version with the original signal. The delay time can range from a few milliseconds to several seconds, limited only by the amount of RAM you have available to store the delayed signal.

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

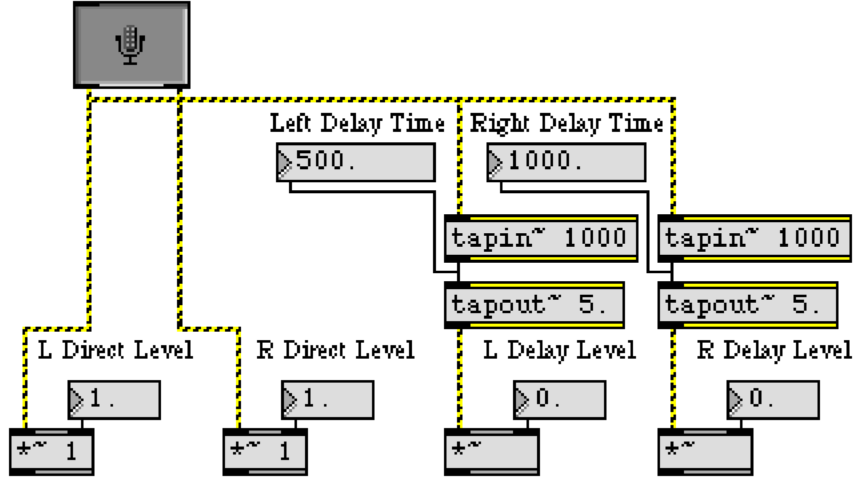
The only object to which the outlet of **tapin~** should be connected is a **tapout~** object. This connection links the **tapout~** object to the buffer stored by **tapin~**. The **tapout~** object “taps into” the delayed signal at certain points in the past. In the above example, **tapout~** gets the 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.

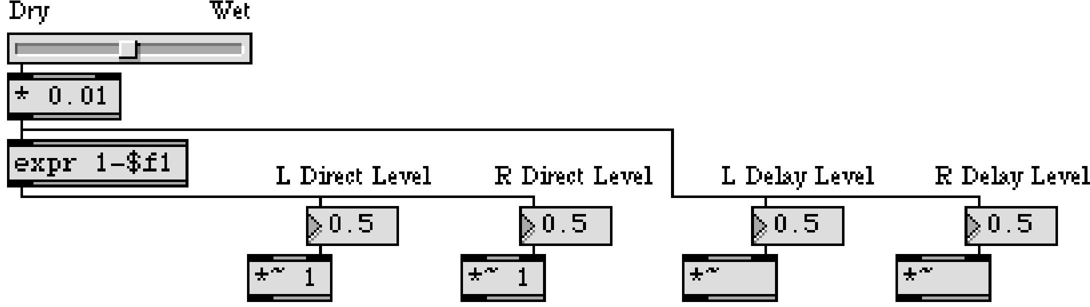
At this point you don't hear any delayed signal because the "Direct Level" for each channel is set at 1 and the "Delay Level" for each channel is set at 0. The signal is being delayed, but you simply don't hear it because its amplitude is scaled to 0.



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.

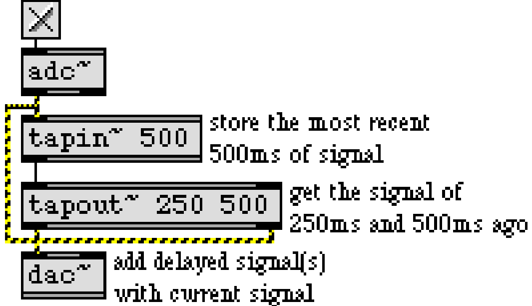
In future tutorial chapters, you will see how to create delay feedback, how to use continuously variable delay times for flanging and pitch effects, and other ways of altering sound using delays, filters, and other processing techniques.

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 **tapin~**'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.

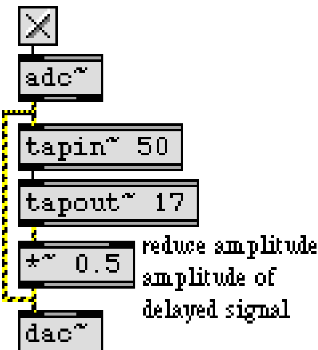
Delay emulates reflection

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.

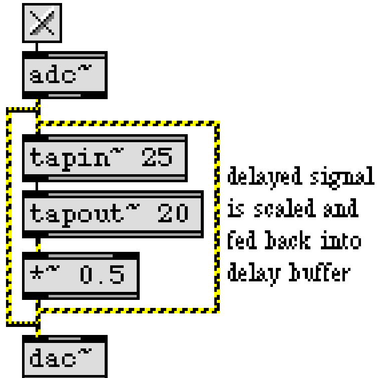


Scaling the amplitude of a delayed signal, to emulate absorption

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, but will be considered in a later Tutorial on reverberation.

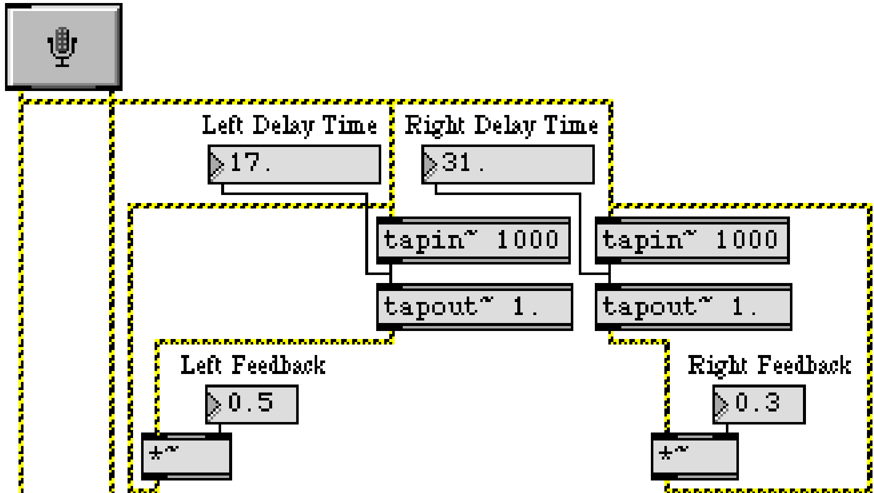
Delaying the delayed signal

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**es 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**.

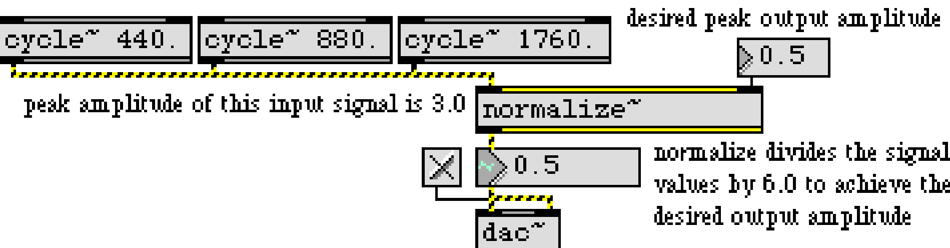
Note that any time you feed audio signal back into a system, you have a potential for overloading the system. That's why it's important to scale the signal by some factor less than 1.0 (with the *~ objects and the "Feedback" **number box**es) before feeding it back into the

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.

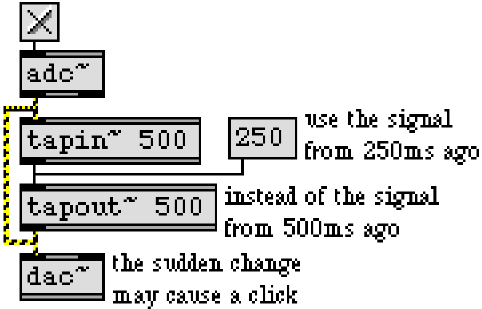
Summary

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.

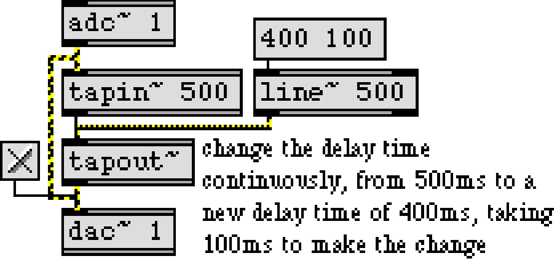
Variable delay time

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

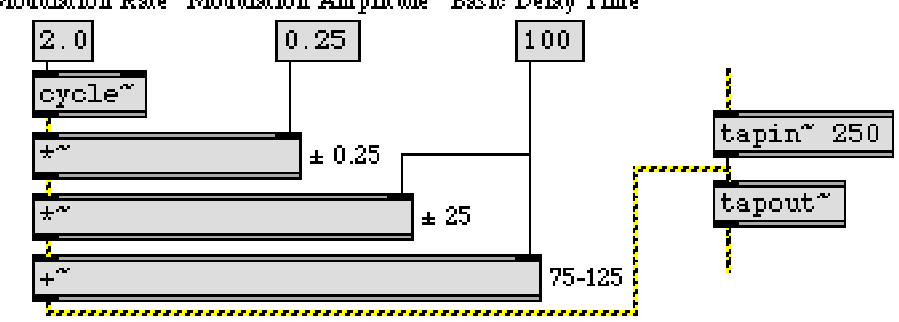
actually being produced by the source. Conversely, if the sound source is moving away from the listener, the wavefronts arrive at the listener slightly less frequently than they are actually being produced. The classic case of Doppler effect is the sound of an ambulance siren. As the ambulance passes 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 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.

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*.

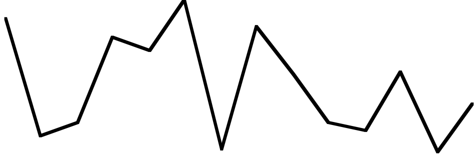
We've already seen in the preceding chapter how slight pitch shifts can be introduced by varying the delay time of a signal. When we mix this signal with its original undelayed version, we create interference between the two signals, resulting in a constantly varying filtering effect known as flanging. 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.

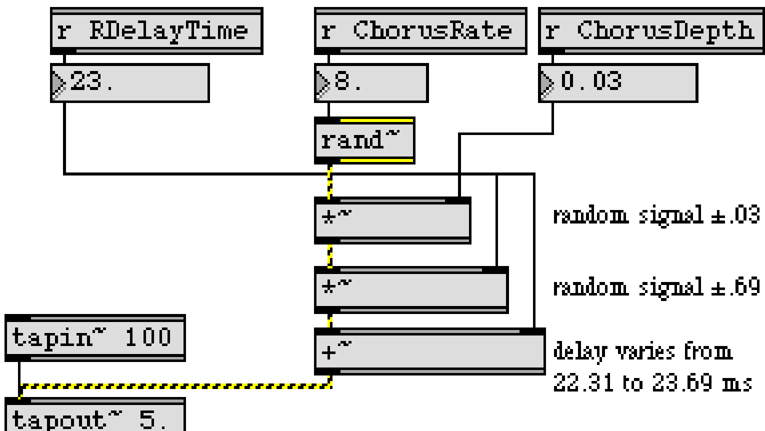
The **rand~** object chooses random numbers between -1 and 1, but does so less frequently than every sample. You can specify the frequency at which it chooses a new random value. In between those randomly chosen samples, **rand~** interpolates linearly from one value to the next to produce an unpredictable but more contiguous signal.

Random values chosen every sample



Random values chosen 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.

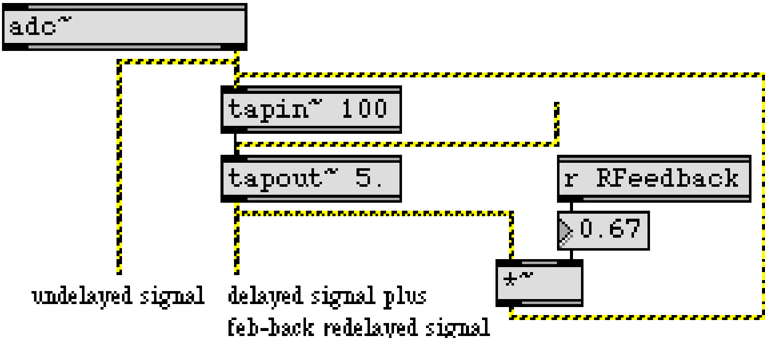


Unpredictable variations using **rand~**

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.

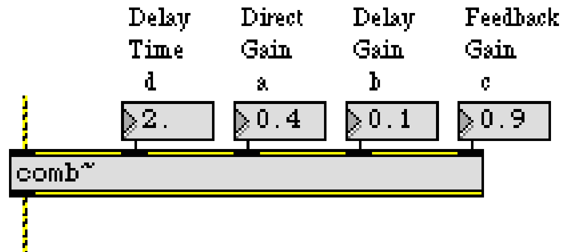
Comb filter: comb~

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

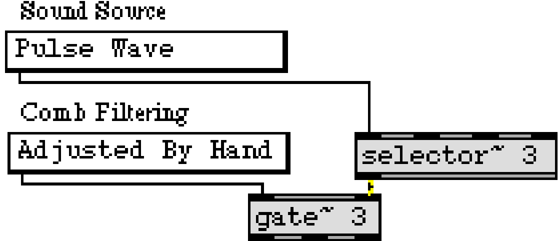
Technical detail: At any given moment in time (we’ll call that moment *t*), **comb~** uses the value of the input signal (x_t), to calculate the output y_t in the following manner.

$$y_t = ax_t + bx_{(t-d)} + cy_{(t-d)}$$

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.



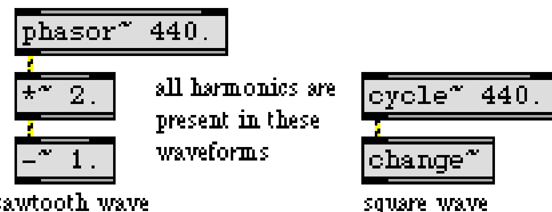
Choose a sound source and route it to the desired filtering using the pop-up menus

- 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

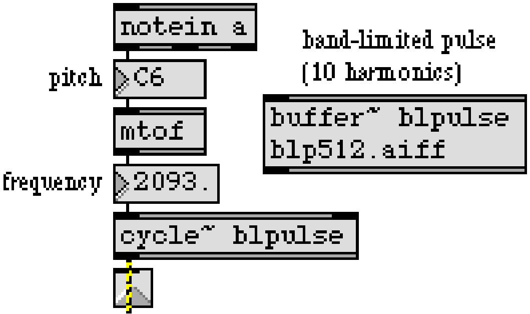
The effects of a filter are most noticeable when there are many different frequencies in the source sound, which can be altered by the filter. If we want to apply a comb filter to a pitched sound with a harmonic spectrum, it makes most sense to use a sound that has many partials such as a sawtooth wave 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.

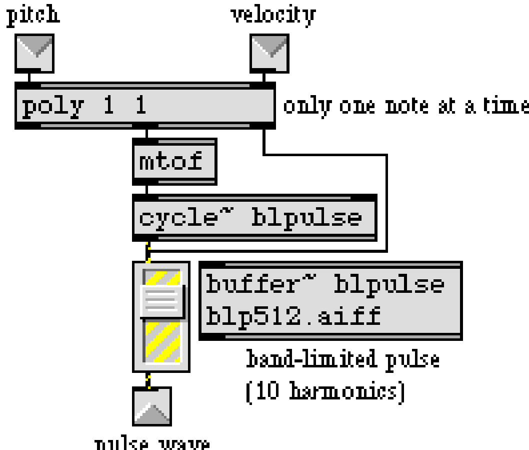


Playing a band-limited pulse wave with MIDI

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~

MIDI-to-amplitude conversionThe 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**es (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 (ie., 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.

Audio input and output with MSP

By default, MSP uses the Sound Manager for audio input and output. It starts up with the current settings of the Sound Manager as you've specified in the Monitors & Sound control panel.

You can also use MSP with audio interface cards. Some cards provide Sound Manager drivers for their cards; however, if MSP supports the audio card directly with an MSP audio driver, do NOT use the card's Sound Manager drivers.

Your choice of the Sound Manager or direct support for an audio interface card is determined by the presence of an Audiodriver file in the Max folder. Audiodriver files are found in a folder located inside your Max folder called audiodrivers. To use a particular interface card, move the Audiodriver file out of the audiodrivers folder and into the Max folder before launching Max.

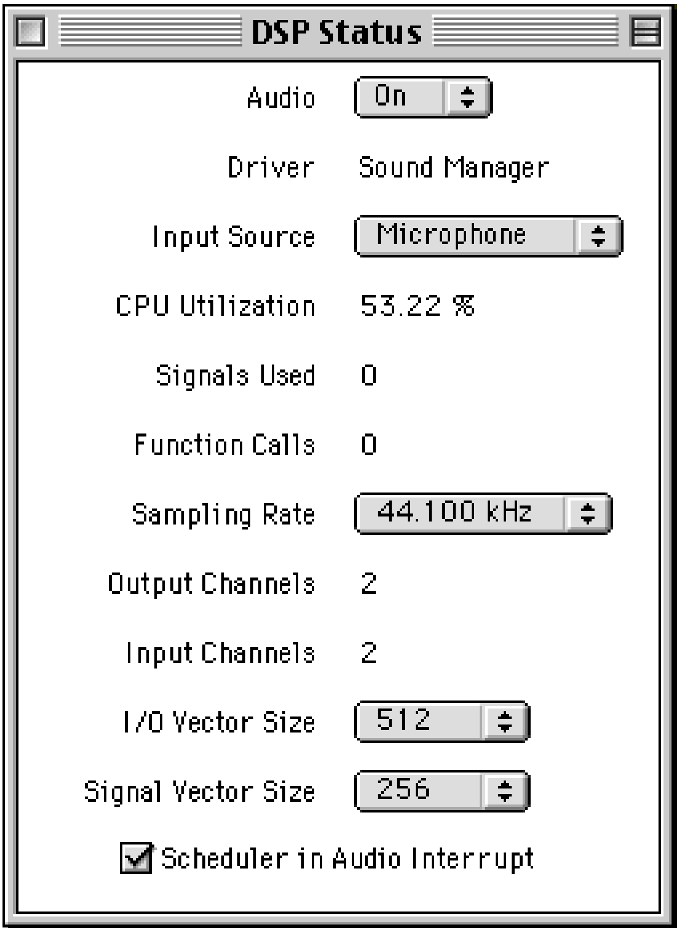
If MSP finds no Audiodrivers in the Max folder, it will use the Sound Manager. If it finds an Audiodriver but there is a problem initializing it—for instance, if you don't have the interface card installed in your computer, or you don't have a required INIT for the interface card installed in your Extensions folder—MSP will use the Sound Manager. If an Audiodriver file is present and it is able to initialize properly, MSP will bypass the Sound Manager completely and work directly with the specified audio interface card.

If you want to switch from one audio input/output system to another, you need to quit Max and change the Audiodriver file in your Max folder.

The rest of this chapter explains audio input and output in more detail. First, we'll give you a tour of the DSP Status window, where all global audio parameters can be changed—even while the music is playing—and you can get an indication of how hard the CPU is currently working to produce sound. Then we'll discuss the details of using the Sound Manager with MSP, followed by the details of using audio interface cards. Information specific to each of the currently supported audio interface cards is provided at the end of the chapter.

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.



Information about MSP is shown in the DSP Status window

The first item in the DSP Status window is a switch for turning MSP audio processing on and off. Audio can be turned on and off from within a Max patch, as well, so this switch also serves as an indicator of the current on/off status.

If you're using an unregistered copy of MSP, this item will indicate Disabled after you have turned the audio on and off once.

The second item shows what is controlling the audio input and output of MSP. By default, the audio is handled by the Macintosh Sound Manager, in which case there will be two available output channels and two input channels. If you have an audio interface card and you install an MSP driver for it, you will see the name of the driver instead of Sound Manager.

The next item determines the Sound Input source. If you are using the Sound Manager, you will see a pop-up menu showing the input sources available on your computer. Audio interface cards do not provide a choice of input sources. Rather, multiple inputs are treated as separate channels in the **adc~** object.

The next three items show how hard your computer is working to produce audio with MSP. *CPU Utilization* shows how much of your computer's total processing power is being devoted to

the audio processing, *Signals Used* shows the number of internal buffers that were needed by MSP to connect the signal objects used in the current signal network, and *Function Calls* gives an approximate idea of how many calculations are being required for each sample of audio.

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. Note that when using the Sound Manager, both the *Input Source* and the *Sampling Rate* can also be set in the Monitors & Sound (or Sound) control panel. However, MSP ignores a change in sample rate made with the Monitors & Sound control panel while Max is running.

The *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 the output device (the Sound Manager output or the audio interface card) at one time. The Signal Vector Size sets the number of samples that are calculated by all 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, for each I/O vector that needs to be calculated, MSP calculates two or more signal vectors in succession. With an I/O vector size of 256, and a sampling rate of 44.1 kHz, MSP calculates about 5.8 milliseconds of sound data at a time.

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. 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.

Note that some audio interface cards do not provide a choice of I/O Vector Sizes

Changing the vector size 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 *Scheduler in Audio Interrupt* feature is available with the Sound Manager when you have Overdrive enabled from Max's Options menu. It runs the Max control scheduler immediately after processing an I/O vector's worth of audio. The control scheduler does things like trigger the bang from a repeating **metro** object, as well as send out any recently received MIDI data. Checking the *Scheduler in Audio Interrupt* option can greatly improve the timing of audio events that are triggered from control processes or external MIDI input. However, the improvement in timing is directly related to your choice of I/O Vector Size, since this determines the interval at which the scheduler runs. For instance, with an I/O Vector Size of 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 I/O 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 what sounds acceptable to you.

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.

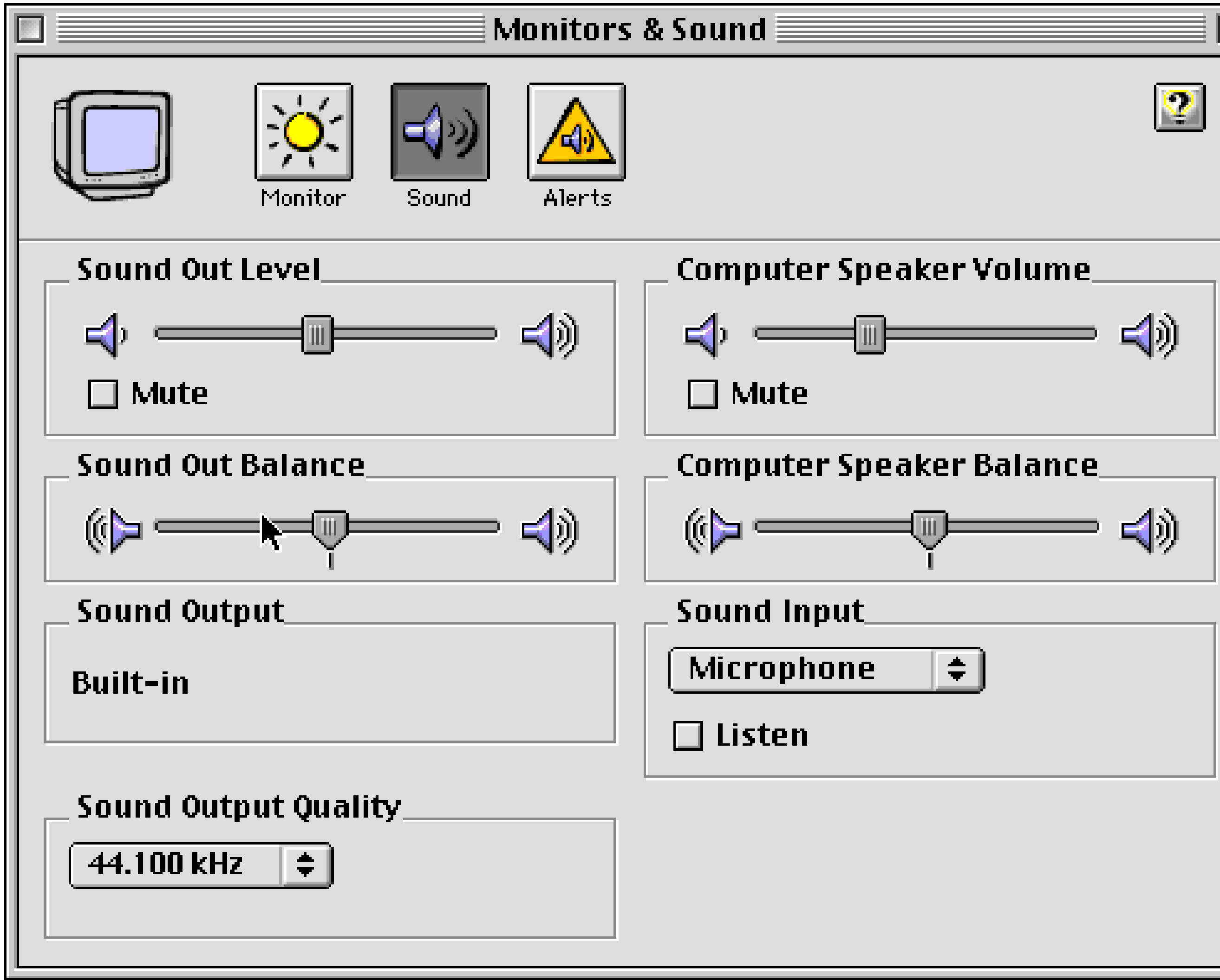
Note that the *Scheduler in Audio Interrupt* feature requires version 3.5.9 or later of Max. It will not appear in the DSP Status window if you are not using a driver that supports it or a version of Max that doesn't allow the feature to be used. Also, it will be disabled if Overdrive is not checked in the Options menu.

Using the Sound Manager with MSP

MSP uses the Sound Manager when there is no MSP Audiodriver file located in the Max folder, or when all Audiodriver files fail to initialize (for example, when they don't find the appropriate hardware or software).

The Monitors & Sound control panel only changes the settings used by the Sound Manager. It does not affect external audio interface cards, unless those cards are being used with the Sound Manager driver (which, just to repeat, we *don't* recommend unless there is no MSP support for the audio interface card).

The next figure shows the Sound part of the Monitors & Sound control panel.



Sound settings in the Monitors & Sound control panel

On most Mac OS computers and system versions, the following controls in Monitors & Sound will affect MSP:

The volume (the Sound Out level)

The Sound Out level affects the volume of the sound coming from MSP. Note that for computers equipped with RCA line output jacks (such as the Macintosh 8500 and 8600), the Sound Out level affects these connections as well as the headphone-style jack found on all computers.

The sampling rate (Sound Output quality)

This setting just stores a user preference for the desired sampling rate that is read by applications such as MSP when they start up. So, if you change this setting in the Monitors & Sound control while MSP is running, nothing will change in MSP.

However, you can change the sampling rate while MSP is running using the DSP Status window in MSP. Double-click on any **dac~** or **adc~** object in a locked Patcher window to see the DSP Status window, then choose the desired sampling rate from the pop-up menu.

Even though Monitors & Sound refers to the sampling rate as the Sound Output quality, MSP uses the same sampling rate for sound input and output. If it didn't, incoming sound would be distorted and transposed when you listened to it.

When using the Sound Manager, MSP always starts up using the sampling rate set in the Monitors & Sound control panel. However, when you choose a different sampling rate in MSP's DSP Status window, the setting in Monitors & Sound is not affected.

The Sound Input source

The Sound Input source sets the source of audio signals to the **adc~** or **ezadc~** input objects in MSP. The Monitors & Sound control panel will not let you change the Sound Input source while MSP (or any other application that is using Sound Input) is running. However, MSP will let you can change it in the DSP Status window. Unlike the sampling rate, if you change the Sound Input source by using the DSP Status window in MSP, that *will* change Monitors & Sound's setting (with one exception as noted below).

Different computer models have different Sound Input sources. A desktop machine with a CD-ROM will usually let you choose between Internal CD (or CD) and Microphone. The latter refers to the line-level mini stereo jack on the back of the computer. The fact that it's line-level (not mic-level) means you need to use either a line input source (such as a CD player), a microphone that puts out line level (such as the one that came with your computer), or the output of a mixer to which a microphone has been connected. If the computer has an *internal* microphone (for example, a PowerBook), Sound Input sources might be listed as Internal Microphone (the built-in mic), Expansion Bay (a CD-ROM drive), and External Audio Input—or Line In—(the mini stereo jack on the back of the computer). Machines with "AV" features also provide a choice of the AV Connector; this refers to the red-and-white RCA jacks

on the back of the AV-equipped computer. Note that while the AV Connector is an explicit audio *input* choice, the AV Connector *output* is always the same as the output from your computer's mini-stereo jack.

If you are using an audio interface card and have installed the Sound Manager driver for it in the Extensions folder, the name of the card might be present in the pop-up menu of Sound Input sources. For example, the Sound Manager driver for a Digidesign card would be listed as *Digidesign*. To select a Sound Manager driver in OS 8.0 and later, you may need to use the Sound control panel from an earlier system.

In Power Macintosh G3 models and System 8.1, the Monitors & Sound control panel behaves differently than in other computers. If you don't have a Sound Manager driver for an audio interface card installed, the Sound Input source is called the Sound Monitoring Source. If you do have a driver installed, pop-up menus for both the Sound Input source and the Sound Monitoring Source are present. However, you can't choose the audio interface card in either item's pop-up menu. One workaround is to use a Sound control panel from an older system. It appears that the Sound Monitoring Source will always be set to None when Max is launched. However, you can set it to something while Max is running, which changes what MSP uses as its audio input. After you do so, choosing a Sound Input source in MSP's DSP Status window has no effect until you turn the Sound Monitoring Source to None again. If the Sound Input source menu is present in Monitors & Sound, MSP will use the choice in that pop-up menu as its Sound Input source when it starts up. If only the Sound Monitoring Source pop-up menu is present, the selection is used for MSP's sound input only if it is not None. If it is None, then the previously selected choice (if any) that you made in MSP is used.

Another difference is that inserting an Audio CD into a G3 while MSP is running will set MSP's sound input to the CD, whether that's what you wanted or not. What's more, changing the input source with the pop-up menu in the DSP Status window has no effect. Sound Monitoring Source listed in the Monitors & Sound control panel changes to CD as well. Ejecting the CD returns the Sound Monitoring Source to None and MSP regains control over the selection of its sound input.

One difference that could be either useful or irritating is that the Sound Monitoring Source and the output of MSP are mixed together on a G3, something that is not true on other computers. Note that this is only true is you change Sound Monitoring Source or insert a CD while Max is running, since launching Max always sets the Sound Monitoring Source to none.

The Sound Output destination

If you have an audio interface card installed in your computer, and you're not using Mac OS 8, you will see a pop-up menu allowing you to pick either the Built-in audio output or the card's audio output. When you choose an audio card as a Sound Output destination, the list of sampling rates may change or disappear entirely.

The Monitors & Sound control panel will let you change the Sound Output destination while MSP is running, but it has no effect. There is no way to change the Sound Output destination from within MSP.

You can find additional information about Sound Manager drivers for specific cards in the next section.

Using audio interface cards

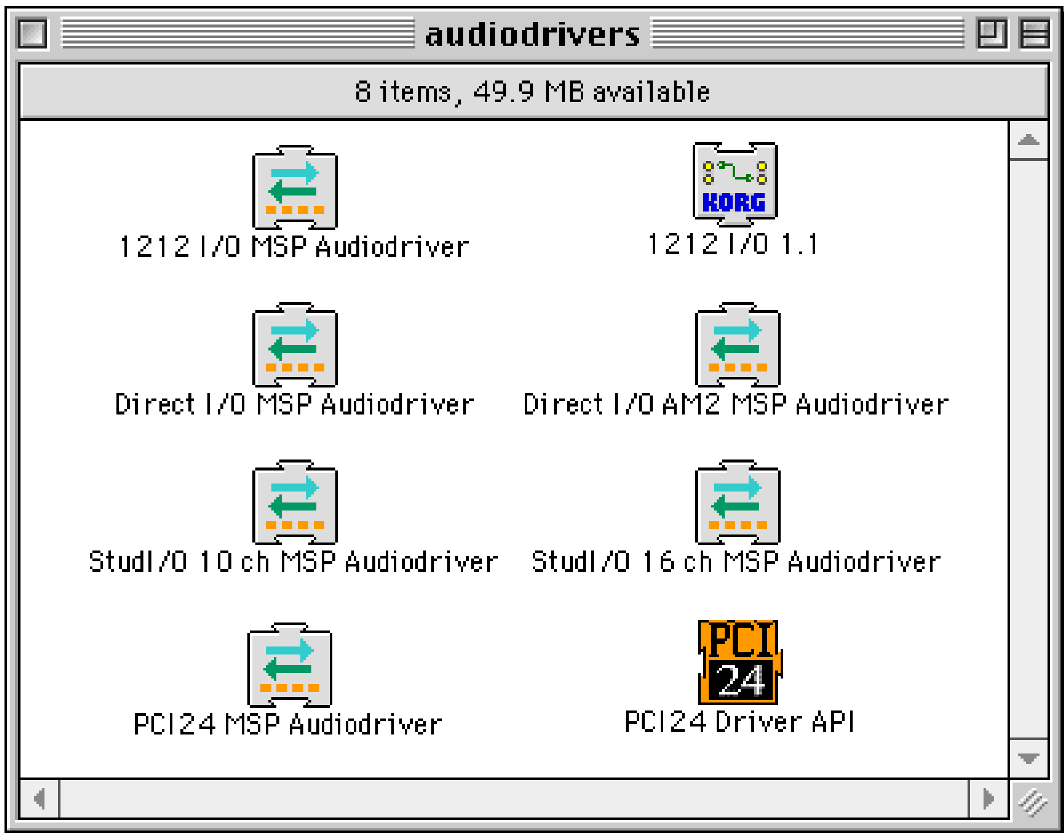
This section discusses how to use MSP with audio interface cards. Doing so completely bypasses the Sound Manager, the Monitors & Sound control panel, and all of the idiosyncrasies of the Mac OS support for sound. However, each card introduces its own idiosyncrasies and limitations; these are mentioned in the sections below that deal with each brand of audio hardware.

Direct MSP support for an audio card has nothing to do with the Sound Manager, so you do not need to choose your audio card for Sound Input or Output in the Monitors & Sound Control panel. Indeed, in some cases, you will need to disable the Sound Manager driver or INIT for the card before MSP can address it directly.

Using an audio card has a slight detrimental effect on the amount of DSP that can be done by MSP in comparison with the Sound Manager. There are two reasons for this. First, data must be transferred across a slow PCI bus (rather than just to a memory location as with the Sound Manager), and in many cases the computer ends up waiting for the completion of this task. So, there is less CPU time available for signal processing. Second, many cards use 24-bit input and output, so there is simply more memory being shoved around. For the CPU, moving large amounts of memory around is expensive in comparison with calculating signal values.

The audiodrivers folder

The MSP Installer places the plug-in drivers that let MSP use various audio interface cards in a folder called *audiodrivers* located inside your Max folder. All drivers are *inoperative* as long as they remain inside the audiodrivers folder.



The audiodrivers folder holds all the drivers, but hides them from MSP

To use an Audiodriver file, you must remove it from the audiodrivers folder and place it in the Max folder (the same folder that contains the Max Audio Library and the Max application). MSP will attempt to load the driver when it starts up. Normally, if one driver fails to initialize because it can't find the right hardware, the next one found in the Max folder (in alphabetical order) is tried. However, certain drivers will crash the computer under certain circumstances if they so much as search for their hardware (and find someone else's). For this reason, all drivers should be kept safely out of the way in the audiodrivers folder unless you're using them.

When MSP starts up, you will see messages such as the following in the Max window.

If MSP is successful in loading an audio driver:

```
Searching for audio drivers...
Trying Direct I/O MSP Audiodriver...
MSP: Using Direct I/O MSP Audiodriver
```

If MSP is not successful in loading an audio driver:

```
Searching for audio drivers...
Trying Direct I/O MSP Audiodriver...
MSP: Using Sound Manager
```

There may also be some kind of error message printed by the driver explaining that it couldn't find the hardware or initialize it properly.

INITs for audio cards

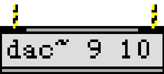
To use MSP with an audio interface card, you may need to install an INIT for that card in the Extensions folder in your System Folder. This INIT either was supplied with the card when you bought it or can be found in the MSP audiodrivers folder. The notes on each audio card below will tell you what is required. After placing the INIT in your Extensions folder and restarting the computer (if necessary), take the appropriate audio driver file out of the audiodrivers folder and place it in the Max folder, then open the Max application.

Changing audio settings

When you use an audio interface card, the only way to change its settings is with the DSP Status window in MSP. Double click on any **dac~** or **adc~** object in a locked Patcher window to open the DSP Status window. With most cards, you'll be able to change the sampling rate (affecting both output and input) and the I/O Vector Size. (See the discussion of vector size earlier in this chapter.) No card has the option of changing the Sound Input source; the best implementation is usually to provide all the card's inputs at once as different signal outputs from an **adc~** object.

Using more than two audio channels

Some audio interface cards support more than two channels of audio I/O. These channels can be accessed by specifying the channel numbers as typed-in arguments to the **adc~** and **dac~** objects. For instance, if you want to send signals to audio channels 9 and 10 on your card, you would make a **dac~** 9 10 object.



Play the received signals on channels 9 and 10 of the audio card

Similarly, an **adc~** 9 10 object receives input on channels 9 and 10.

Send out the signals coming in on channels 9 and 10 of the audio card

dac~ and **adc~** can accept up to eight channel number arguments. Since you can have as many **dac~** and **adc~** objects as you like, this doesn't restrict you from using all 16 channels if you have them available. The **ezdac~** and **ezadc~** objects are limited to channels 1 and 2.

Notes on specific audio cards

The sections below contain information specific to individual audio cards supported by MSP. If you do not find information about a supported audio card here, please check the MSP Support page at *http://www.cycling74.com/support.*

Digidesign Audiomedia II

The Audiomedia II is supported by the file called *Direct I/O AM2 MSP Audiodriver*. This will work with version 3.2 of the DigiSystem™ INIT, but version 3.3 or later will not support it.

The Audiomedia II is a Nubus card with two analog channels in and out. You may lose a bit of processor efficiency using this card versus the Sound Manager, but you'll gain an improvement in latency and audio quality. Note that Digidesign support software for this card that MSP uses is not actively maintained. You need to install DigiSystem™ INIT version 3.2 in order for MSP to work with the Audiomedia II. DAE is not required. Various versions of DigiSystem™ INIT are available at the Digidesign web site, *http://www.digidesign.com*.

Digidesign Sound Drivers is another INIT that depends on DigiSystem™ INIT and allows a Digidesign card to work with the Sound Manager. If you have the Digidesign Sound Drivers installed, it doesn't matter whether they're selected for use by the Sound Manager in the Monitors & Sound control panel, and loading them doesn't interfere with the operation of MSP.

One thing that could interfere with the operation of the Audiomedia II is if the *Direct I/O MSP Audiodriver* (which supports other Digidesign hardware) is in the Max folder and renamed so that it loads before the *Direct I/O AM2 MSP Audiodriver*. No error will be returned, the driver will seem to have loaded correctly, but no sound will be heard. Place only the MSP Digidesign audio driver appropriate for your hardware in the Max folder.

The Digidesign Sound Drivers that support the Sound Manager are incapable of handling audio input and output at the same time, yet you can choose Digidesign as both a Sound Input source and a Sound Output destination in the Monitors & Sound control panel. Since many users will want to do both audio input and output at the same time, this limitation renders the Digidesign Sound Drivers for the Sound Manager inappropriate for use with MSP. Use the support provided by the audio drivers instead.

In comparison to using the Sound Manager, two choices are not available with the Audiomedia II in the DSP Status window: you can't change the Input Source (obviously, since there are just the inputs on the card), and you can't change the I/O Vector Size. There is a problem with dropouts when MSP uses I/O Vector Sizes larger than 512 on Digidesign hardware. In any case, 512 samples is the minimum allowable vector size and larger sizes would only increase the audio input-output latency.

Digidesign Audiomedia III, ProTools, and d24

Digidesign hardware is supported by the MSP audio driver called *Direct I/O MSP Audiodriver*. (Place only the MSP audio driver for Digidesign hardware appropriate for your setup in the Max folder.)

The Audiomedia III is a Motorola 56002-based PCI-bus card with two analog and two S/PDIF digital channels in and out. When using the Audiomedia III with MSP, the analog and digital channels are separate, giving you four distinct input and output channels. You need to install DigiSystem™ INIT version 3.3 or later in order for MSP to work with Digidesign hardware. DAE is not required. The latest versions of DigiSystem™ INIT are available at the Digidesign web site, *http://www.digidesign.com*.

Digidesign Sound Drivers is another INIT that depends on DigiSystem™ INIT and allows a Digidesign card to work with the Sound Manager. If you have the Digidesign Sound Drivers installed, it doesn't matter whether they're selected for use by the Sound Manager in the Monitors & Sound control panel, and loading them doesn't interfere with the operation of MSP.

In comparison to the Sound Manager, two choices are not available in the DSP Status window with Digidesign hardware: you can't change the Input Source (obviously, since there are just the inputs on the card), and you can't change the I/O Vector Size. There is a problem with dropouts when MSP uses I/O Vector Sizes larger than 512 on Digidesign hardware. In any case, 512 samples is the minimum allowable vector size and larger sizes only increase the audio input-output latency without any appreciable performance improvement.

Changing the sampling rate while MSP is running could result in increased latency between audio input and output and possible audio distortion. There are two solutions if this occurs: You can quit Max and reopen the application; MSP will remember the sampling rate you chose in the DSP Status window so there will be no need to change it again. Alternatively, change the sampling rate *before* turning on the audio for the first time.

If you are using a certain Digidesign hardware, you may not get any audio input or output unless you configure their driver with the specific audio interface box you are using. To do this, use the following message

```
; dsp driver setup
```

You'll see a dialog allowing a choice of audio cards, interfaces, and some other options. Note that you can *only* use the hardware setup dialog before you turn on the audio for the first time after launching Max. You only need to do the configuration once; the settings are saved in a Digidesign preferences file and are global to all applications that use the hardware.

Lucid PCI24

The MSP audio driver file that supports the Lucid PCI24 directly is called the *PCI24 MSP Audiodriver*. You will notice a significant performance degradation when using the PCI24 with MSP. On a typical machine, 25 percent of the CPU is used just to do the I/O (versus about 1-3

percent for the Sound Manager). However, audio input-output latency is much shorter in comparison to the Sound Manager.

The Lucid PCI24 is a Motorola DSP56301-based card with two digital input and output channels. Two different sets of input and output jacks allow either S/PDIF or AES-EBU format I/O, but not both at the same time. At this time MSP only supports the S/PDIF jacks.

Our testing indicates that a machine containing both a Digidesign Audiomedia III card and a Lucid PCI24 card will fail to boot.

The PCI24 ships with a Sound Manager driver called the PCI24 Driver. To use the card with MSP, this file must be disabled and replaced with the *PCI24 Driver API* found in the MSP audiodrivers folder. If the PCI24 Driver and the PCI24 Driver API are both present in the Extensions folder, the former will load first and prevent the latter from working, and MSP will not be able to use the card directly.

If you have another DSP56301-based card in your computer, ensure that the PCI24 Driver or PCI24 Driver API INITs are not installed, otherwise you may have strange problems. These drivers have no idea that the 56301 they see on the PCI bus is not a PCI24, so they load their DSP code into the card, putting it into a state unrecognizable to the legitimate driver. Conversely, if you have a PCI24 card installed, do not attempt to use software for another DSP56301-based card. For instance, trying to initialize the MSP Sonorus StudI/O audio driver with a PCI24 card installed will freeze the machine.

In the DSP Status window, you cannot change the Input Source, but all other choices are available. A nice feature is the ability to use a 32K sampling rate.

When making any change in the DSP Status window, it is necessary for MSP to unload the PCI24 driver and reload it again. This may occasionally cause the audio output to be a rather loud digital noise signal. Changing the settings again usually corrects the problem.

In the current version of the PCI24 Driver API, there is a problem with an I/O Vector Size of 1024. Setting MSP to use this size causes the audio output to be disabled, the audio output from the card to be the audio input to the card, and the input to MSP to be mixture of the audio input to the card and the audio output MSP wanted to have played from the audio output of the card. All other I/O Vector Sizes listed in the pop-up menu work.

Sonorus StudI/O

The Sonorus StudI/O is a DSP56301-based card with four optical connectors for multi-channel digital input and output, and an analog monitor output. The card can either be configured for 16 channels as two ADAT inputs and outputs; or for 10 channels as one eight-channel ADAT input and output and one two-channel S/PDIF input and output.

The StudI/O does not require an INIT in the System Folder. However, there are two separate audio driver files in the MSP audiodrivers folder that support the card. To use the 10-channel configuration, use *StudI/O 10 ch MSP Audiodriver*; to use the 16-channel configuration, use the

StudI/O 16 ch MSP Audiodriver. Place one of these (but not both) in the Max folder. If you want to switch configurations, you need to quit Max and exchange the locations of the audio driver files, putting the one you want to use in the Max folder, and the one you don't want to use in the audiodrivers folder.

There are two potential problems with initializing the StudI/O audio driver in MSP. The first would be the presence of another company's 56301-based INIT in the Extensions folder. This causes an error when the StudI/O driver tries to initialize the 56301. The second would be if the StudI/O driver tries to initialize a 56301 card that isn't a StudI/O. The computer may freeze.

The analog stereo monitor output of the card consists of all the odd-numbered channels mixed together in the left channel and all the even-numbered channels mixed together in the right channel.

To attenuate the level of the monitor, which by default is full-volume and painfully loud if you just stick some headphones in the jack, use the following:

; dsp driver atten <value>

In this message, <value> is a number between 0 (full volume) and 63 (-63 dB). So, the higher the number, the softer the volume.

Sending a message directly to the dsp object internal to MSP

You can also set the synchronization mode of the StudI/O with a message to the dsp object.

; dsp driver sync internal

sets the synchronization to internal, which is the default when MSP starts up with the StudI/O.

; dsp driver sync a

sets the synchronization to the A (ADAT) input.

; dsp driver sync b

sets the synchronization to the B input, either ADAT (for 16 channels) or S/PDIF (for 10 channels).

; dsp driver sync wc

sets the synchronization to the Word Clock input.

; dsp driver sync dsp

sets the synchronization to the DSP Timer 1.

The number of channels will be listed in the DSP Status window as 10 or 16 depending on the audio driver file you are using. The I/O Vector Size is currently fixed at 1024. The sampling rate can be either 44.1kHz or 48kHz. Switching the sampling rate while the audio is on is not reliable. You may need to quit Max and restart before the card will work again.

Korg 1212I/O

The Korg 1212 I/O is supported by the file called *1212 I/O MSP Audiodriver*. It provides eight channels of digital input and output with an ADAT connector, two channels of S/PDIF input and output, and two channels of analog input and output. In addition, it allows you to sync to ADAT, word clock, or S/PDIF and can mix the incoming signal of any channel with the output from the computer. The 1212 I/O boasts the best latency performance of any of the audio interface cards currently supported by MSP.

MSP requires the 1212 I/O INIT version 1.1 or later to be installed in the Extensions folder. Version 1.0 will not work. If you need a more recent version of the 1212 I/O software and manual for the Mac OS, you can get it from the Korg web site at *http://www.korg.com/1212down.html*. If you switch from version 1.0 of the INIT to a later version, you must turn off your computer before any software will function properly with the new version. Restarting isn't sufficient.

The Korg card ships with a program called *1212 I/O Utility* that sets parameters for how the output will monitor the input. Do not be confused by this program's volume faders and routing options. They do not affect output from the computer—rather, they change the way input channels are monitored, mixed, and routed. MSP uses a default configuration for the 1212 I/O that overrides anything you set up in this program. This configuration is as follows:

- Monitors for all inputs are off. All input channels are routed to the corresponding output channels.
- Sampling rate internally clocked at 44.1 kHz or 48 kHz, depending on what it was set at the last time you ran MSP.
- Analog input gain set at maximum

By sending messages to the `dsp` object internal to MSP, you can change any of these settings (see next page). The 1212 I/O's channel assignments in MSP are similar to what you see in the 1212 I/O Utility window:

MSP Channel	*1212 I/O Input/Output*
1	Analog Left
2	Analog Right
3	S/PDIF Left
4	S/PDIF Right
5	ADAT 1
6	ADAT 2

7	ADAT 3
8	ADAT 4
9	ADAT 5
10	ADAT 6
11	ADAT 7
12	ADAT 8

The number of channels will be listed in the DSP Status window as 12. The I/O Vector Size is fixed at 512. The sampling rate can be either 44.1kHz or 48kHz.

Controlling special features of the 1212 I/O

Using the driver message to the dsp object internal to MSP, you can change the settings of the 1212 I/O's input-output delay, its input monitoring facility, and the synchronization mode.

The offset keyword changes the latency (delay) between audio input and output of the 1212 I/O. You may wish to do this if you're hearing audio distortion on the input signal (meaning the latency is too low), or want to attempt to reduce the latency below the default setting.

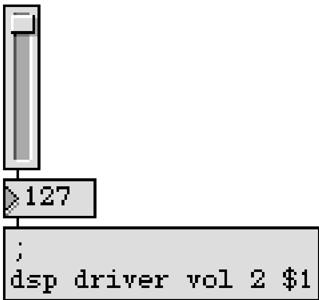
```
; dsp driver offset <n>
```

In this message, n ranges from 0 to 7. The default setting is n = 4. The smallest delay is apparently at n = 2, but the largest delay isn't necessarily at 7.

To change the volume of the input monitor for any input channel, use the vol keyword:

```
; dsp driver vol <chan> <level>
```

In this message, <chan> is a number between 1 and 12 corresponding to the source shown in the table above and <level> is 0 for off (the default) and 127 for 0 dB attenuation (full volume). When the monitor level is non-zero, the input source will be heard mixed with the MSP output on the output channel it's routed to. The following example shows how to use a slider as a fader on the input monitor for the right Analog channel:



Set the volume of the input monitor for the right Analog channel of Korg 1212 I/O

Input monitor volume changes can be sent while the audio is turned on in MSP, and they will have an immediate effect. Once again, a reminder that these volume settings are for monitoring the input to the 1212 I/0, not for changing the volume of the output of MSP.

To change the output channel for an input channel's monitor, use the following:

```
; dsp driver route <input channel> <output channel>
```

In this message, <input channel> is the MSP channel (1-12) corresponding to the input source you want to route, and <output channel> is the MSP channel corresponding to the output where you want to monitor the input. To monitor the S/PDIF inputs on the analog channels, you would use the following:

```
; dsp driver route 3 1
; dsp driver route 4 2
```

Note that volume changes occur on ***inputs***, not outputs. To set the input monitor level you will be hearing on MSP channels 1 and 2 to full volume in the example above, you need to send the messages ; dsp driver vol 3 127 and ; dsp driver vol 4 127, ***not*** the messages ; dsp driver vol 1 127 and ; dsp driver vol 2 127.

If the audio is turned on when you make a routing change, no immediate effect will be heard. You need to stop and restart the audio in order for a routing change to take effect.

To set the input gain for the two analog channels, use the following:

```
; dsp driver inputgain <left> <right>
```

In this message <left> and <right> are the input gains for the left and right analog channels (MSP channels 1 and 2). 0 is no attenuation, and 255 is maximum attenuation that mutes the input.

To set the synchronization mode used by the card, use the following:

; dsp driver sync internal — For internal sync (the default)

; dsp driver sync adat — For ADAT sync

; dsp driver sync word — For Word Clock or S/PDIF sync. Word Clock takes priority over S/PDIF sync if both are present.

Input

signal — In left inlet: The signal is multiplied by the signal coming into the right inlet, or a constant value received in the right inlet.

In right inlet: The signal is multiplied by the signal coming into the left inlet, or a constant value received in the left inlet.

float or int — In left inlet: A factor by which to multiply the signal coming into the right inlet. If a signal is also connected to the left inlet, a float or int is ignored.

In right inlet: A factor by which to multiply the signal coming into the left inlet. If a signal is also connected to the right inlet, a float or int is ignored.

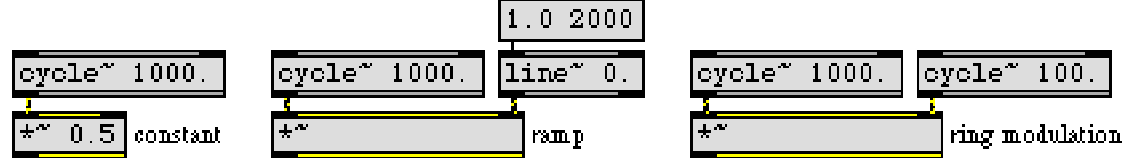
Arguments

float or int — Optional. Sets an initial value by which to multiply the signal coming into the left inlet. If a signal is connected to the right inlet, the argument is ignored. If no argument is present, and no signal is connected to the right inlet, the initial value is 0 by default.

Output

signal — The product of the two inputs.

Examples



Scale a signal's amplitude by a constant or changing value, or by another audio signal

See Also

/~	Divide one signal by another
Tutorial 2	Fundamentals: Adjustable oscillator
Tutorial 8	Synthesis: Tremolo and ring modulation

Note: Any signal inlet of any MSP object automatically uses the sum of all signals received in that inlet. Thus, the +~ object is necessary only to show signal addition explicitly, or to add a float or int offset to a signal.

Input

signal — In left inlet: The signal is added to the signal coming into the right inlet, or a constant value received in the right inlet.

In right inlet: The signal is added to the signal coming into the right inlet, or a constant value received in the left inlet.

float or int — In left inlet: An offset to add to the signal coming into the right inlet. If a signal is also connected to the left inlet, a float or int is ignored.

In right inlet: An offset to add to the signal coming into the left inlet. If a signal is also connected to the right inlet, a float or int is ignored.

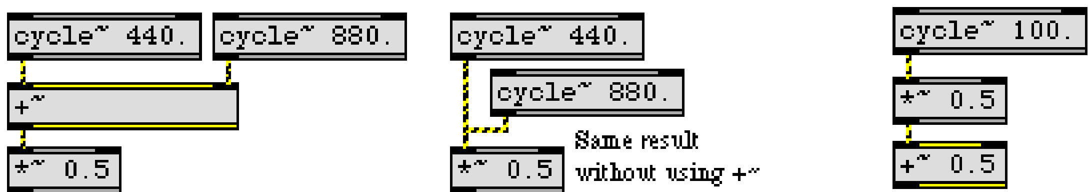
Arguments

float or int — Optional. Sets an initial offset to add to the signal coming into the left inlet. If a signal is connected to the right inlet, the argument is ignored. If no argument is present, and no signal is connected to the right inlet, the initial value is 0 by default.

Output

signal — The sum of the two inputs.

Examples



Mix signals... ...or add a DC offset to a signal

See Also

-~ — Subtract one signal from another

Input

signal — In left inlet: The signal coming into the right inlet or a constant value received in the right inlet is subtracted from this signal.

In right inlet: The signal is subtracted from the signal coming into the left inlet, or a constant value received in the left inlet.

float or int — In left inlet: Subtracts the signal coming into the right inlet from this value. If a signal is also connected to the left inlet, a float or int is ignored.

In right inlet: An amount to subtract from the signal coming into the left inlet. If a signal is also connected to the right inlet, a float or int is ignored.

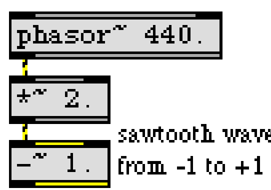
Arguments

float or int — Optional. Sets an initial amount to subtract from the signal coming into the left inlet. If a signal is connected to the right inlet, the argument is ignored. If no argument is present, and no signal is connected to the right inlet, the initial value is 0 by default.

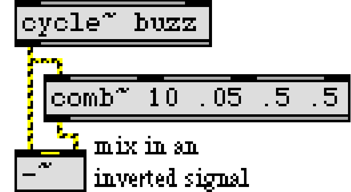
Output

signal — The difference between the two inputs.

Examples



Negative DC offset



Subtraction used to invert a signal before adding it in

See Also

+~ — Add signals

Note: Division is not a computationally efficient operation. The /~ object is optimized to multiply a signal coming into the left inlet by the reciprocal of either the initial argument or an int or float received in the right inlet. However, when two signals are connected, /~ uses the significantly more inefficient division procedure.

Input

signal — In left inlet: The signal is divided by a signal coming into the right inlet, or a constant value received in the right inlet.

In right inlet: The signal is used as the divisor, to be divided into the signal coming into the left inlet, or the constant value received in the left inlet.

float or int — In left inlet: The number is divided by the signal coming into the right inlet. If a signal is also connected to the left inlet, a float or int is ignored.

In right inlet: A number by which to divide the signal coming into the left inlet. If a signal is also connected to the right inlet, a float or int is ignored.

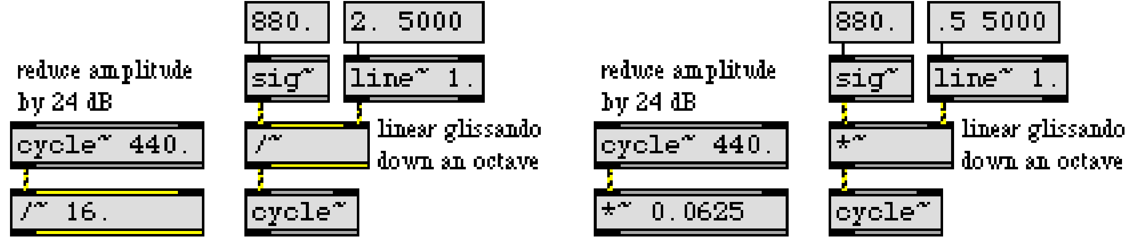
Arguments

float or int — Optional. Sets an initial value by which to divide the signal coming into the left inlet. If a signal is connected to the right inlet, the argument is ignored. If no argument is present, and no signal is connected to the right inlet, the initial value is 1 by default.

Output

signal — The ratio of the two inputs, i.e., the left input divided by the right input.

Examples



It is more computationally efficient to use an equivalent multiplication when possible

See Also

*~ — Multiply two signals

Is less than, *comparison of two signals*

Input

signal — In left inlet: The signal is compared to a signal coming into the right inlet, or a constant value received in the right inlet. If it is less than the value in the right inlet, 1 is sent out; otherwise, 0 is sent out.

In right inlet: The signal is used for comparison with the signal coming into the left inlet.

float or int — In right inlet: A number to be used for comparison with the signal coming into the left inlet. If a signal is also connected to the right inlet, a float or int is ignored.

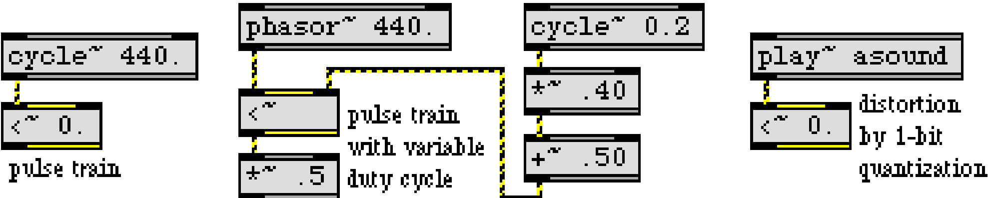
Arguments

float or int — Optional. Sets an initial comparison value for the signal coming into the left inlet. 1 is sent out if the signal is less than the argument; otherwise, 0 is sent out. If a signal is connected to the right inlet, the argument is ignored. If no argument is present, and no signal is connected to the right inlet, the initial value is 0 by default.

Output

signal — If the signal in the left inlet is less than the value in the right inlet, 1 is sent out; otherwise, 0 is sent out.

Examples



Convert any signal to only 1 and 0 values

See Also

==~	*Is equal to*, comparison of two signals
>~	*Is greater than*, comparison of two signals

Input

signal — In left inlet: The signal is compared to a signal coming into the right inlet, or a constant value received in the right inlet. If it is equal to the value in the right inlet, 1 is sent out; otherwise, 0 is sent out.

In right inlet: The signal is used for comparison with the signal coming into the left inlet.

float or int — In right inlet: A number to be used for comparison with the signal coming into the left inlet. If a signal is also connected to the right inlet, a float or int is ignored.

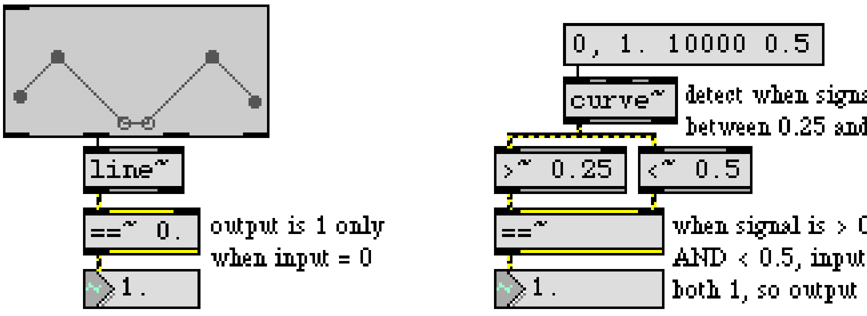
Arguments

float or int — Optional. Sets an initial comparison value for the signal coming into the left inlet. 1 is sent out if the signal is equal to the argument; otherwise, 0 is sent out. If a signal is connected to the right inlet, the argument is ignored. If no argument is present, and no signal is connected to the right inlet, the initial value is 0 by default.

Output

signal — If the signal in the left inlet is equal to the value in the right inlet, 1 is sent out; otherwise, 0 is sent out.

Examples



Detect when a signal equals a certain value, or when two signals equal each other

See Also

<~	*Is less than*, comparison of two signals
>~	*Is greater than*, comparison of two signals
change~	Report signal direction
edge~	Detect logical signal change

Is greater than, *comparison of two signals* >~

Input

signal In left inlet: The signal is compared to a signal coming into the right inlet, or a constant value received in the right inlet. If it is greater than the value in the right inlet, 1 is sent out; otherwise, 0 is sent out.

In right inlet: The signal is used for comparison with the signal coming into the left inlet.

float or int In right inlet: A number to be used for comparison with the signal coming into the left inlet. If a signal is also connected to the right inlet, a float or int is ignored.

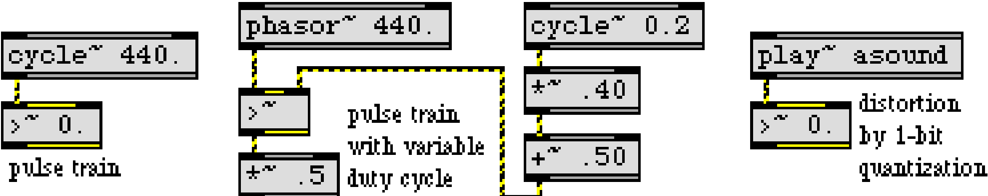
Arguments

float or int Optional. Sets an initial comparison value for the signal coming into the left inlet. 1 is sent out if the signal is greater than the argument; otherwise, 0 is sent out. If a signal is connected to the right inlet, the argument is ignored. If no argument is present, and no signal is connected to the right inlet, the initial value is 0 by default.

Output

signal If the signal in the left inlet is greater than the value in the right inlet, 1 is sent out; otherwise, 0 is sent out.

Examples



Convert any signal to only 1 and 0 values

See Also

<~	*Is less than*, comparison of two signals
==~	*Is equal to*, comparison of two signals
sah~	Sample and hold

abs~

Absolute value of a signal

Input

signal Any signal.

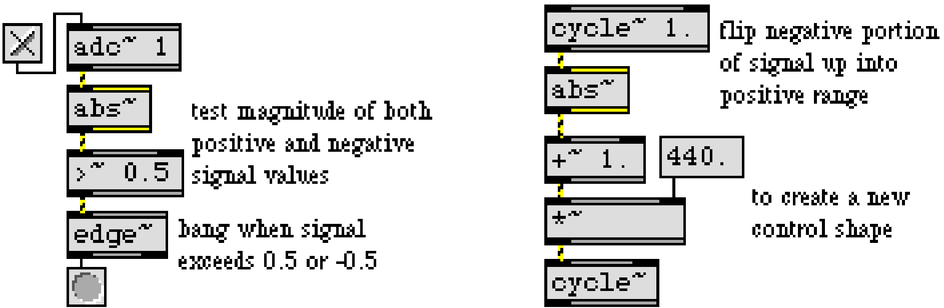
Arguments

None.

Output

signal A signal consisting of samples which are the absolute (i.e., non-negative) value of the samples in the input signal.

Examples



Convert negative signal values to positive signal values

See Also

avg~ Signal average

Input

int A non-zero number turns on audio processing in all loaded patches. 0 turns off audio processing in all loaded patches.

start Turns on audio processing in all loaded patches.

stop Turns off audio processing in all loaded patches.

startwindow Turns on audio processing only in the patch in which this **adc~** is located, and in subpatches of that patch. Turns off audio processing in all other patches.

(mouse) Double-clicking on **adc~** opens the DSP Status window.

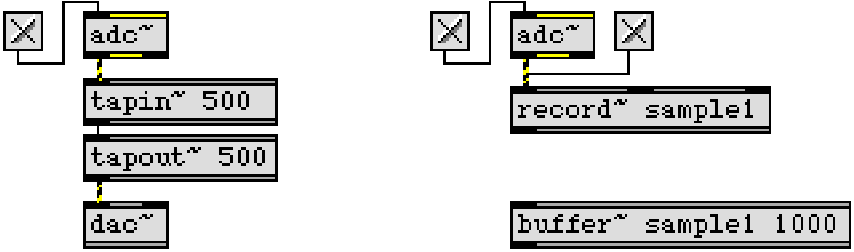
Arguments

int Optional. One or more numbers specifying which audio input channels will be sent out the outlet(s). Each argument will cause a corresponding outlet to be created. You can specify up to sixteen input channels using numbers 1 to 16, but the actual number of input channels available will depend on the hardware installed in the computer on which the patch is being used. If the Sound Manager is being used, there will be two input channels available. Other audio drivers may have additional channels. If no argument is typed in, **adc~** will have two outlets, for input channels 1 and 2.

Output

signal The signal arriving at the computer's input is sent out, one channel per outlet. If there are no typed-in arguments, the channels are 1 and 2, numbered left-to-right; otherwise the channels are in the order specified by the arguments.

Examples



Audio input for processing and recording

See Also

ezadc~ Audio on/off; analog-to-digital converter
dac~ Audio out; digital-to-analog converter
Tutorial 13 Sampling: Recording and playback

allpass~ *Allpass filter*

Input

signal — In left inlet: Any signal to be filtered. The filter mixes the current input sample with an earlier output sample, according to the formula:

$$y_n = -gx_n + x_{n-(DR/1000)} + gy_{n-(DR/1000)}$$

where R is the sampling rate and D is a delay time in milliseconds.

In middle inlet: Delay time (D) in milliseconds for a past output sample to be added into the current output.

In right inlet: Gain coefficient (g), for scaling the amount of the input and output samples to be sent to the output.

float or int — The filter parameters in the middle and right inlets may be specified by a float or int instead of a signal. If a signal is also connected to the inlet, the float or int is ignored.

clear — Clears **allpass~**'s memory of previous outputs, resetting them to 0.

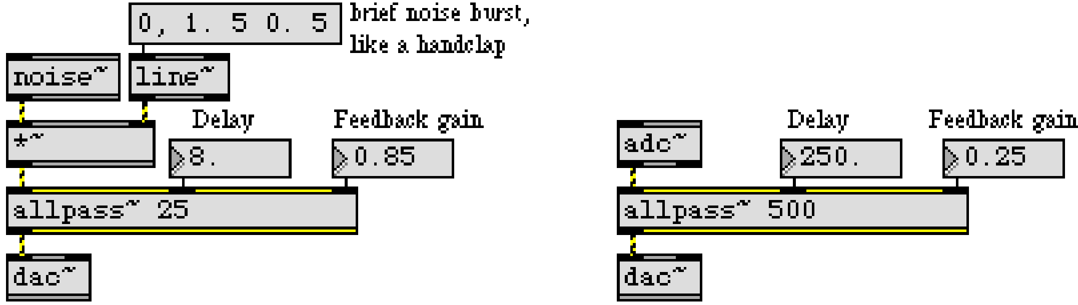
Arguments

float — Optional. Up to four numbers, to set the maximum delay time and initial values for the delay time D and gain coefficient g. If a signal is connected to a given inlet, the coefficient supplied as an argument for that inlet is ignored. If no arguments are present, the maximum delay time defaults to 10 milliseconds.

Output

signal — The filtered signal.

Examples



Short delay with feedback to blur the input sound, or longer delay for discrete echos

See Also

biquad~	Two-pole two-zero filter
comb~	Comb filter

lores~	Resonant lowpass filter
reson~	Resonant bandpass filter

avg~ *Signal average*

Input

bang — Triggers a report of the average (absolute) amplitude of the signal received since the previous bang, and clears **avg~**'s memory in preparation for the next report.

signal — The signal to be averaged.

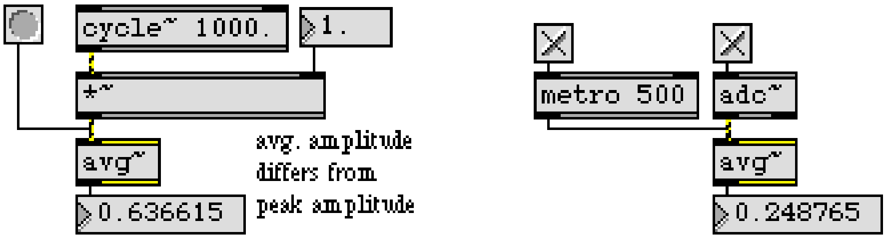
Arguments

None.

Output

float — When bang is received in the inlet, **avg~** reports the average amplitude of the signal received since the previous bang.

Examples



Report the average (absolute) amplitude of a signal

See Also

meter~ — Visual peak level indicator

Input

None.

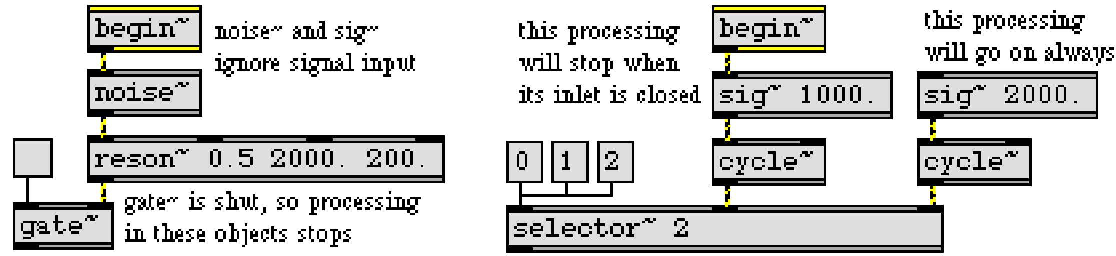
Arguments

None.

Output

signal **begin~** outputs a constant signal of 0. It is used to designate the beginning of a portion of a signal network that you wish to be turned off when it's not needed. You connect the outlet of **begin~** to the signal inlet of another object to define the beginning of a signal network that will eventually pass through a **gate~** or **selector~**. One **begin~** can be used for each **gate~** or **selector~** signal inlet. When the signal coming into **gate~** or **selector~** is shut off, no processing occurs in any of the objects in the signal network between the **begin~** and the **gate~** or **selector~**.

Examples



See Also

selector~	Assign one of several inputs to an outlet
gate~	Route a signal to one of several outlets
Tutorial 5	Fundamentals: Turning signals on and off

biquad~ *Two-pole, two-zero filter*

Input

signal In left inlet: Signal to be filtered. The filter mixes the current input sample with the two previous input samples and the two previous output samples according to the formula: $y_n = a_0x_n + a_1x_{n-1} + a_2x_{n-2} - b_1y_{n-1} - b_2y_{n-2}$.

In 2nd inlet: Amplitude coefficient a_0, for scaling the amount of the current input to be passed directly to the output.

In 3rd inlet: Amplitude coefficient a_1, for scaling the amount of the previous input sample to be added to the output.

In 4th inlet: Amplitude coefficient a_2, for scaling the amount of input sample $n-2$ to be added to the output.

In 5th inlet: Amplitude coefficient b_1, for scaling the amount of the previous output sample to be added to the current output.

In right inlet: Amplitude coefficient b_2, for scaling the amount of output sample $n-2$ to be added to the current output.

float The coefficients in inlets 2 to 6 may be specified by a float instead of a signal. If a signal is also connected to the inlet, the float is ignored.

list The five coefficients can be provided as a list in the left inlet. The first number in the list is coefficient a_0, the next is a_1, and so on. If a signal is connected to a given inlet, the coefficient supplied in the list for that inlet is ignored.

clear Clears **biquad~**'s memory of previous inputs and outputs, resetting x_{n-1}, x_{n-2}, y_{n-1}, and y_{n-2} to 0.

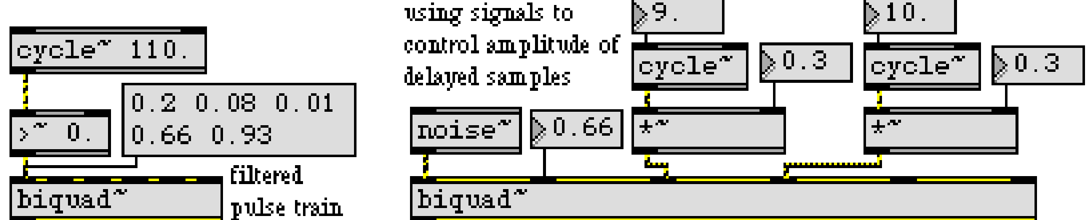
Arguments

float Optional. Up to five numbers, to set initial values for the coefficients a_0, a_1, a_2, b_1, and b_2. If a signal is connected to a given inlet, the coefficient supplied as an argument for that inlet is ignored.

Output

signal The filtered signal.

Examples



Filter coefficients may be supplied as numerical values or as varying signals

See Also

comb~	Comb filter
reson~	Resonant bandpass filter

Input

read — Reads an AIFF or Sound Designer II sound file into the sample memory of the **buffer~**. If no symbol argument appears after the word read, a standard open file dialog is opened showing available AIFF and Sound Designer II files. The word read, followed by a filename that is located in Max's file search path, reads that file into **buffer~** immediately without opening the dialog box. The filename may be followed by a float indicating a starting time in the file, in milliseconds, to begin reading. (The beginning of the file is 0.) The starting time may be followed by a float duration, in milliseconds, of sound to be read into **buffer~**. This duration overrides the current size of the object's sample memory. If the duration is negative, **buffer~** reads in the entire file and resizes its sample memory accordingly. If duration argument is zero or not present, the **buffer~** object's sample memory is not resized if the sound file is larger than the current sample memory size. The duration may be followed by a number of channels to be read in. If the number of channels is not specified, **buffer~** reads in the number of channels indicated in the header of the AIFF or Sound Designer II file. Whether or not the number of channels is specified in the read message, the previous number of channels in a **buffer~** is changed to the number of channels read from the file.

readagain — Reads sound data from the most recently loaded AIFF or Sound Designer II file (specified in a previous read or replace message).

replace — Same as the read message with a negative duration argument. replace, followed by a symbol , treats the symbol as a filename located in Max's file search path. If no argument is present, **buffer~** opens a standard open file dialog showing available AIFF and Sound Designer II files. Additional arguments specify starting time, duration, and number of channels as with the read message.

write — Saves the contents of **buffer~** into a sound file. A standard file dialog is opened for naming the file unless the word write is followed by a symbol, in which case the file is saved in the current default directory, using the symbol as the filename. If the current format of the buffer~ object is AIFF, the write message will save it as an AIFF file. If the current format is Sound Designer II, the write message will save it as a Sound Designer II file. The file format is AIFF by default. If a Sound Designer II file is opened using the read message, or the **buffer~** was previoulsly saved as a Sound Designer II file, the file format changes to Sound Designer II.

writeaiff — Saves the contents of **buffer~** into an AIFF file. A standard file dialog is opened for naming the file unless the word write is followed by a symbol, in which case the file is saved in the current default directory, using the symbol as the filename.

writesd2 — Saves the contents of **buffer~** into a Sound Designer II file. A standard file dialog is opened for naming the file unless the word write is followed by a symbol, in which case the file is saved in the current default directory, using the symbol as the filename.

clear — Erases the contents of **buffer~**.

size — The word size, followed by a duration in milliseconds, sets the size of the **buffer~** object's sample memory. This limits the amount of data that can be stored, unless this size limitation is overridden by a replace message or a duration argument in a read message.

sr — The word sr, followed by a sampling rate, sets the **buffer~** object's sampling rate. By default, the sampling rate is the current output sampling rate, or the sampling rate of the most recently loaded sound file.

name — The word name, followed by a symbol, changes the name by which other objects such as **cycle~**, **groove~**, **lookup~**, **peek~**, **play~**, **record~**, and **wave~** can refer to the **buffer~**. Objects that were referring to the **buffer~** under its old name lose their connection to it. Every **buffer~** object should be given a unique name; if you give a **buffer~** object a name that already belongs to another **buffer~**, that name will no longer be associated with the **buffer~** that first had it.

(remote) — The contents of **buffer~** can be altered by the **peek~** and **record~** objects.

(mouse) — Double-clicking on **buffer~** opens an editing window where you can view the contents of the **buffer~**.

Arguments

symbol — Obligatory. The first argument must be a name by which other objects can refer to the **buffer~** to access its contents.

symbol — Optional. After the **buffer~**'s name, you may type the name of a sound file to load when the **buffer~** is created.

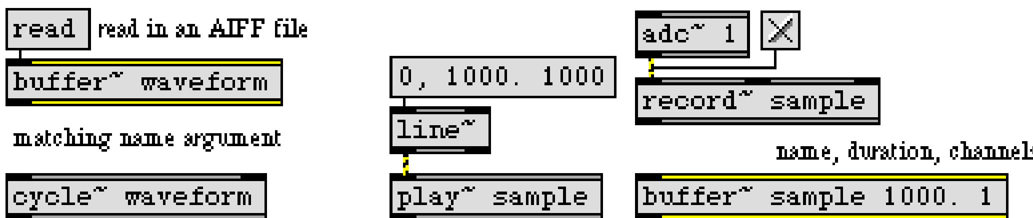
float or int — Optional. After the optional filename argument, a duration may be provided, in milliseconds, to set the size of the **buffer~**, which limits the amount of sound that will be stored in it. (A new duration can be specified as part of a read message, however.) If no duration is typed in, the **buffer~** has no sample memory. It does not, however, limit the size of a sound file that can be read in.

int — Optional. After the duration, an additional argument may be typed in to specify the number of audio channels to be stored in the **buffer~**. (This is to tell **buffer~** how much memory to allocate initially; however, if a sound file with more channels is read in, **buffer~** will allocate more memory for the additional channels.) The maximum number of channels **buffer~** can hold is four. By default, **buffer~** has one channel.

Output

float When the user clicks or drags with the mouse in **buffer~**'s editing window, the cursor's time location in the **buffer~**, in milliseconds, is sent out the outlet.

Examples



buffer~ *can be used as a waveform table for an oscillator, or as a sample buffer*

See Also

cycle~	Table lookup oscillator
groove~	Variable-rate looping sample playback
lookup~	Transfer function lookup table
peek~	Read and write sample values
play~	Position-based sample playback
record~	Record sound into a buffer
sfplay~	Play sound file from disk
sfrecord~	Record to sound file on disk
wave~	Variable-size table lookup oscillator
Tutorial 3	Fundamentals: Wavetable oscillator
Tutorial 12	Synthesis: Waveshaping
Tutorial 13	Sampling: Recording and playback

Input

signal — An excerpt of the signal is stored as text for viewing, editing, or saving to a file. (The length of the excerpt can be specified as a typed-in argument to the object.)

write — Saves the contents of **capture~** into a text file. A standard file dialog is opened for naming the file. The word write, followed by a symbol, saves the file, using the symbol as the filename, in the same folder as the patch containing the **capture~**. If the patch has not yet been saved, the **capture~** file is saved in the same folder as the Max application. Note that unlike the text display window, the size of the text file is not limited to 32,000 characters.

clear — Erases the contents of **capture~**.

(mouse) — Double-clicking on **capture~** opens a window for viewing and editing its contents.

Arguments

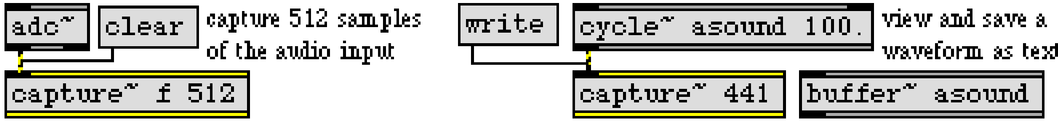
f — Optional. If the first argument is the letter f, **capture~** stores the first signal samples it receives, and then ignores subsequent samples once its storage buffer is full. If the letter f is not present, **capture~** stores the *most recent* signal samples it has received, discarding earlier samples if necessary.

int — Optional. Limits the number of samples (and thus the length of the excerpt) that can be held by **capture~**. If no number is typed in, **capture~** stores 4096 samples. The maximum possible number of samples is limited only by the amount of memory available to the Max application; however, the editing window can only show the first 32,000 characters. A second number argument may be typed in to set the precision (the number of digits to the right of the decimal point) with which samples will be shown in the editing window.

Output

None.

Examples



Capture a portion of a signal as text, to view, save, copy and paste, etc.

See Also

scope~ — Signal oscilloscope

change~

Input

signal Any signal.

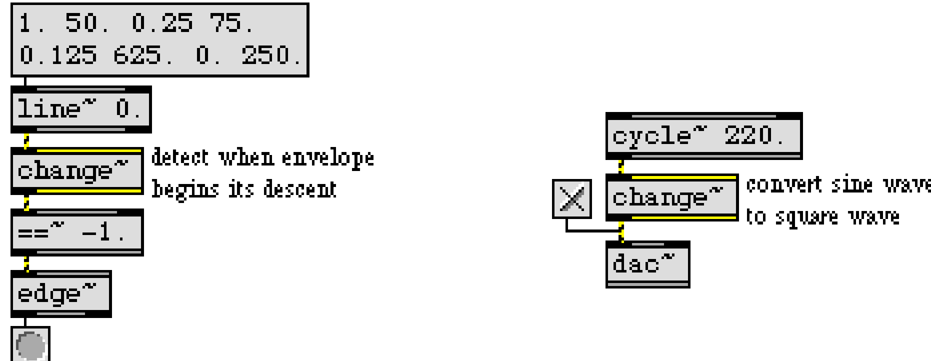
Arguments

None.

Output

signal When the current sample is greater in value than the previous sample, **change~** outputs a sample of 1. When the current sample is the same as the previous sample, **change~** outputs a sample of 0. When the current sample is less than the previous sample, **change~** outputs a sample of -1.

Examples



Detect whether a signal is increasing, decreasing, or remaining constant

See Also

edge~	Detect logical signal transitions
thresh~	Detect signal above a set value

Input

signal — In left inlet: Any signal, which will be restricted within the minimum and maximum limits received in the middle and right inlets.

In middle inlet: Minimum limit for the range of the output signal.

In right inlet: Maximum limit for the range of the output signal.

float or int — The middle and right inlets can receive a float or int instead of a signal to set the minimum and/or maximum.

Arguments

float — Optional. Initial minimum and maximum limits for the range of the output signal. If no argument is supplied, the minimum and maximum limits are both initially set to 0. If a signal is connected to the middle or right inlet, the corresponding argument is ignored.

Output

signal — The input signal is sent out, limited within the specified range. Any value in the input signal that exceeds the minimum or maximum limit is set equal to that limit.

Examples

Output is a clipped version of the input

See Also

<~ — *Is less than*, comparison of two signals
>~ — *Is greater than*, comparison of two signals

comb~

Input

signal In left inlet: Signal to be filtered. The filter mixes the current input sample with earlier input and/or output samples, according to the formula:

$$y_n = ax_n + bx_{n-(DR/1000)} + cy_{n-(DR/1000)}$$

where *R* is the sampling rate and *D* is a delay time in milliseconds.

In 2nd inlet: Delay time (*D*) in milliseconds for a past sample to be added into the current output.

In 3rd inlet: Amplitude coefficient (*a*), for scaling the amount of the input sample to be sent to the output.

In 4th inlet: Amplitude coefficient (*b*), for scaling the amount of the delayed past input sample to be added to the output.

In right inlet: Amplitude coefficient (*c*), for scaling the amount of the delayed past output sample to be added to the output.

float or int The filter parameters in inlets 2 to 5 may be specified by a float instead of a signal. If a signal is also connected to the inlet, the float is ignored.

clear Clears **comb~**'s memory of previous outputs, resetting them to 0.

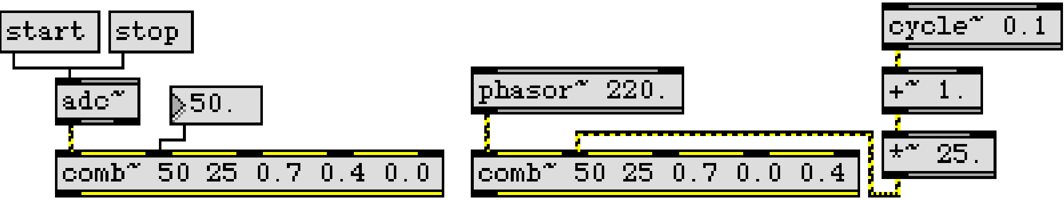
Arguments

float Optional. Up to five numbers, to set the maximum delay time and initial values for the delay time *D* and coefficients *a*, *b*, and and *c*. If a signal is connected to a given inlet, the coefficient supplied as an argument for that inlet is ignored. If no arguments are present, the maximum delay time defaults to 10 milliseconds, and all other values default to 0.

Output

signal The filtered signal.

Examples



Filter parameters may be supplied as float *values or as signals*

See Also

allpass~	Allpass filter
delay~	Delay line specified in samples
reson~	Resonant bandpass filter

Input

signal — Input to a cosine function. The input is stated as a fraction of a cycle (typically in the range from 0 to 1), and is multiplied by 2π before being used in the cosine function.

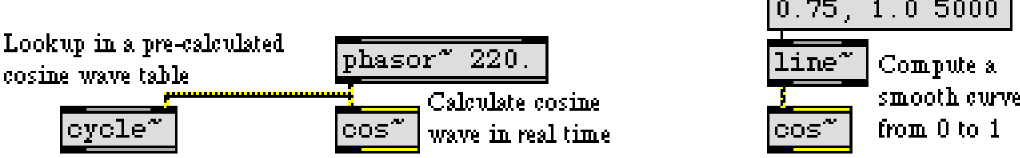
Arguments

None.

Output

signal — The cosine of 2π times the input. The method used in this object to calculate the cosine directly is typically less efficient than using the stored cosine in a **cycle~** object.

Examples



Cosine of the input (a fraction of a cycle) is calculated and sent out

See Also

cycle~	Table lookup oscillator
phasor~	Sawtooth wave generator
wave~	Variable-size table lookup oscillator

Input

bang If the audio is on, the output signal begins counting from its current minimum value, increasing by one each sample. If the signal is already currently counting, it resets to the minimum value and continues upward.

int In left inlet: Sets a new current minimum value, and the output signal begins counting upward from this value.

int In right inlet: Sets the maximum value. When the count reaches this value, it starts over at the minium value on the next sample. A value of 0 (the default) eliminates the maximum, and the count continues increasing without resetting.

stop Causes **count~** to output a signal with its current minimum value.

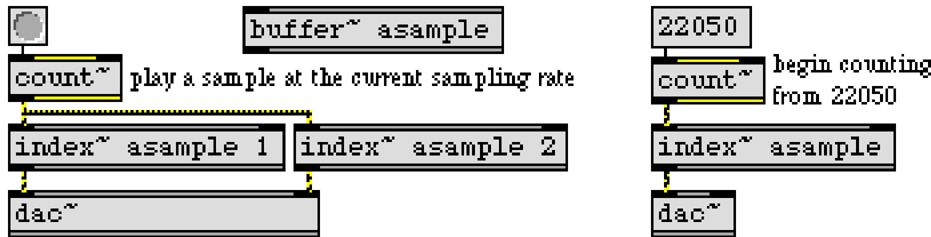
Arguments

int Optional. The first argument sets initial minimum value for the counter. The default value is 0. The second argument sets the initial maximum value for the counter, the default value is 0, which means there is no maximum value.

Output

signal When the audio is first turned on, **count~** always sends out its current minimum value. When a bang or int is received, the count begins increasing from the current minimum value.

Examples



Send out a running count of the passing samples, beginning at a given point

See Also

index~ Sample playback without interpolation
mstosamps~ Convert milliseconds to samples
sampstoms~ Convert samples to milliseconds
Tutorial 13 Sampling: Recording and playback

curve~ *Exponential ramp generator*

Input

list — The first number specifies a target value; the second number specifies an amount of time, in milliseconds, to arrive at that value; and the optional third number specifies a *curve parameter*, for which values from 0 to 1 produce an *exponential* curve, and values from -1 to 0 produce a *logarithmic* curve. The closer to 0 the curve parameter is, the more the curve resembles a straight line, and the farther away the parameter is from 0, the more the curve resembles a step. In the specified amount of time, **curve~** generates an exponential ramp signal from the currently stored value to the target value.

curve~ accepts up to 42 target-time-parameter triples to generate a series of exponential ramps. (For example, the message 0 1000 .5 1 1000 -.5 would go from the current value to 0 in one second, then to 1 in one second.) Once one of the ramps has reached its target value, the next one starts. A new list, float, or int in the left inlet clears any ramps that have not yet generated.

float or int — In left inlet: The number is the target value, to be arrived at in the time specified by the number in the middle inlet. If no time has been specified since the last target value, the time is considered to be 0 and the output signal jumps immediately to the target value.

In middle inlet: The time, in milliseconds, in which the output signal will arrive at the target value.

In right inlet: The number is the curve parameter. Values from 0 to 1 produce an exponential curve, and values from -1 to 0 produce a logarithmic curve. The closer to 0 the number is, the more the curve resembles a straight line; the farther away the number is from 0, the more the curve resembles a step.

Arguments

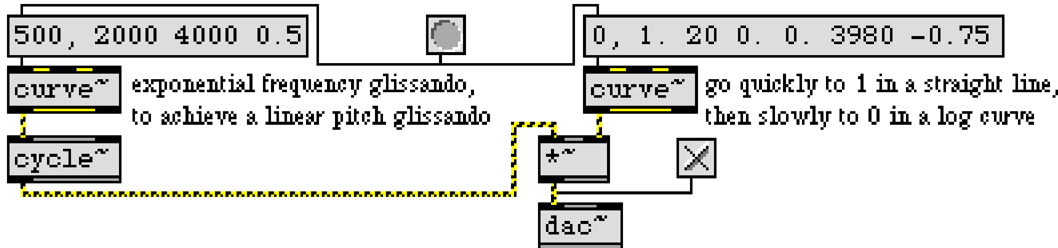
float or int — Optional. The first argument sets an initial value for the signal output. The second argument sets the initial curve parameter. The default values for the initial signal output and curve parameter are 0.

Output

signal — Out left outlet: The current target value, or an exponential curve moving toward the target value according to the most recently received target value, transition time, and curve parameter.

bang — Out right outlet. When **curve~** has finished generating all of its ramps, bang is sent out.

Examples



Curved ramps used as control signals for frequency and amplitude

See Also

line~	Ramp generator

cycle~ *Table lookup oscillator*

The **cycle~** object is an interpolating oscillator that reads repeatedly through one cycle of a waveform, using a wavetable of 512 samples. Its default waveform is one cycle of a cosine wave. It can use other waveforms by accessing samples from a **buffer~** object. The 513th sample in the wavetable source (the **buffer~**) is used for interpolation beyond the 512th sample. For repeating waves, it's usually desirable for the 513th sample to be the same as the first sample, so there will be no discontinuity when the waveform wraps around from the end to the beginning. If only 512 samples are available, **cycle~** assumes a 513th sample equal to the 1st sample.

Input

signal — In left inlet: Frequency of the oscillator. Negative values are allowed.

In right inlet: Phase, expressed as a fraction of a cycle, from 0 to 1. Other values are wrapped around to stay in the 0 to 1 range. If the frequency is 0, connecting a **phasor~** to this inlet is an alternative method of producing an oscillator. If the frequency is non-zero, connecting a **cycle~** or other repeating function to this inlet produces phase modulation, which is similar to frequency modulation.

float or int — In left inlet: Sets the frequency of the oscillator. If there is a signal connected to the left inlet, this number is ignored.

In right inlet: Sets the phase (from 0 to 1) of the oscillator. Other values wrap around to stay between 0 and 1. If the frequency remains fixed, **cycle~** keeps track of phase changes to keep the oscillator in sync with other **cycle~** or **phasor~** objects at the same frequency. If there is a signal connected to the right inlet, this number is ignored.

set — The word set, followed by the name of a **buffer~** object, changes the wavetable used by **cycle~**. The name can optionally be followed by an int specifying the sample offset into the named **buffer~** object's sample memory. **cycle~** uses only the first (left) channel of a multi-channel **buffer~**.

The word set with no arguments reverts **cycle~** to the use of its default cosine wave.

Arguments

float or int — Optional. The initial frequency of the oscillator. If no frequency argument is present, the initial frequency is 0.

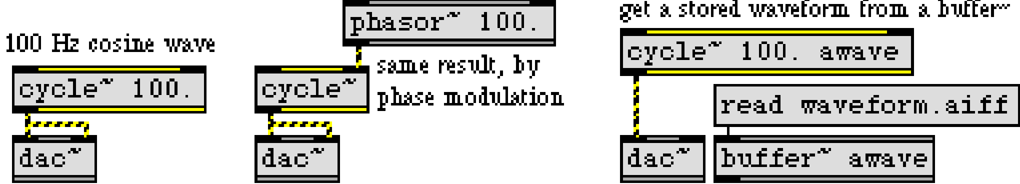
symbol — Optional. The name of a **buffer~** object used to store the oscillator's wavetable. If a float or int frequency argument is present, the **buffer~** name follows the frequency. (No frequency argument is required, however.) If no **buffer~** name is given, **cycle~** uses a stored cosine wave.

int — Optional. If a **buffer~** name has been given, an additional final argument can used to specify the sample offset into the named **buffer~** object's sample memory. **cycle~** only uses the first channel of a multi-channel **buffer~**.

Output

signal A waveform (cosine by default) repeating at the specified frequency, with the specified phase.

Examples



Repeated cosine or any other waveform

See Also

buffer~	Store a sound sample
cos~	Cosine function
phasor~	Sawtooth wave generator
wave~	Variable-size table lookup oscillator
Tutorial 2	Fundamentals: Adjustable oscillator
Tutorial 3	Fundamentals: Wavetable oscillator

Input

signal A signal coming into an inlet of **dac~** is sent to the audio output channel corresponding to the inlet. The signal must be between -1 and 1 to avoid clipping by the DAC.

start Turns on audio processing in all loaded patches.

stop Turns off audio processing in all loaded patches.

startwindow Turns on audio processing only in the patch in which this **dac~** is located, and in subpatches of that patch. Turns off audio processing in all other patches.

int A non-zero number is the same as start. 0 is the same as stop.

(mouse) Double-clicking on **dac~** opens the DSP Status window.

Arguments

int Optional. One or more numbers specifying which audio output channels will be played. Each argument will cause a corresponding inlet to be created. You can specify up to sixteen output channels using numbers 1 to 16, but the actual number of output channels available will depend on the hardware installed in the computer on which the patch is being used. If the Sound Manager is being used, there will be two output channels available. Other audio drivers may have additional channels. If no argument is typed in, **dac~** will have two inlets, for output channels 1 and 2.

Output

None. The signal received in the inlet is played out the corresponding audio output channel.

Examples

Switch audio on and off, send signal to the audio outputs

See Also

ezdac~ Audio on/off; digital-to-analog converter
adc~ Audio in; analog-to-digital converter
Tutorial 1 Fundamentals: Test tone

Input

signal In left inlet: The signal to be delayed.

int In right inlet: The delay time in samples. The delay time cannot be less than 0 (no delay) nor can it be greater than the maximum delay time set by the argument to **delay~**.

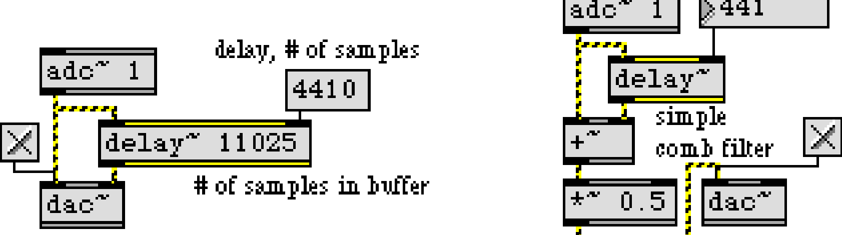
Arguments

int Optional. The first argument sets the maximum delay in samples. This determines the amount of memory allocated for the delay line. The default value is 512. The second argument sets the initial delay time in samples. The default value is 0.

Output

signal The output consists of the input delayed by the specified number of samples. The difference sbetween **delay~** and **tapin~**/**tapout~** are as follows: First, delay times with **delay~** are specified in terms of samples rather than milliseconds, so they will change duration if the sampling rate changes. Second, the **delay~** object can reliably delay a signal a number of samples that is less than a vector size. Finally, unlike **tapin~** and **tapout~** , you cannot feed the output of **delay~** back to its input. If you wish to use feedback with short delays, consider using the **comb~** object.

Examples



Delay signal for a specific number of samples, for echo or filtering effects

See Also

comb~	Comb filter
tapin~	Input to a delay line
tapout~	Output from a delay line

delta~

Signal of sample differences

Input

signal Any signal.

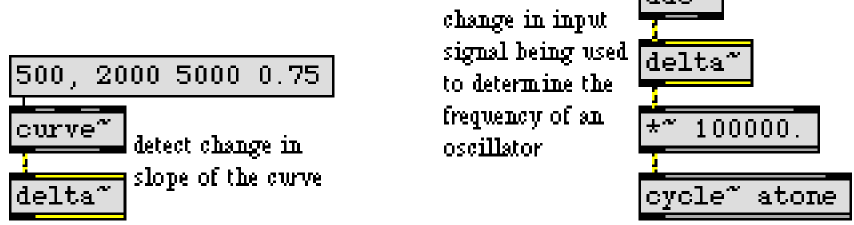
Arguments

None.

Output

signal The output consists of samples that are the difference between the current input sample and the previous input sample. For example, if the input signal contained 1,.5,2,.5, the output would be 1,-.5,1.5,-1.5.

Examples



Report the difference between one sample and the previous sample

See Also

avg~ Report average amplitude

Input

bang — Triggers a report out the **dspstate~** object's outlets, telling whether the audio is on or off, the current sampling rate, and the signal vector size.

(on/off) — The **dspstate~** object reports DSP information whenever the audio is turned on or off.

signal — If a signal is connected to **dspstate~**'s inlet, **dspstate~** reports that signal's sampling rate and vector size, rather than the global sampling rate and signal vector size.

Arguments

None.

Output

int — Out left outlet: If the audio is on or being turned on, 1 is sent out. If the audio is off or being turned off, 0 is sent out.

float — Out middle outlet: Sampling rate of the connected signal or the global sampling rate.

int — Out right outlet: Current signal vector size.

Examples

Trigger an action when audio is turned on or off; use sample rate to calculate timings

See Also

sampstoms~	Convert samples to milliseconds
mstosamps~	Convert milliseconds to samples
Tutorial 20	MIDI control: Sampler
Tutorial 24	Analysis: Using the FFT

edge~ *Detect logical signal transitions*

Input

signal A signal that will change between zero and non-zero values, such as the output of a signal comparison operator.

Arguments

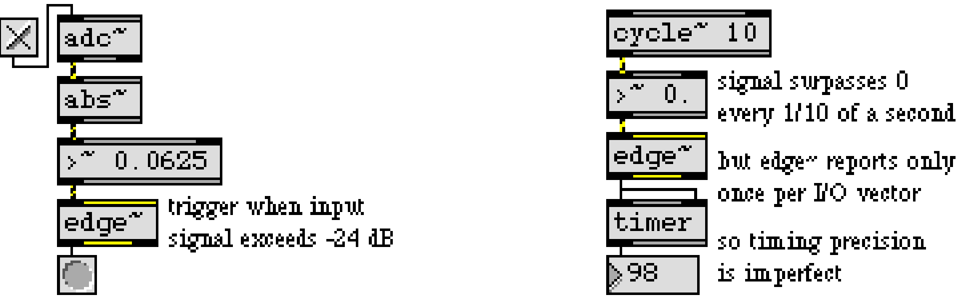
None.

Output

bang Out left outlet: Sent when the input signal changes from zero to non-zero. Experimentation indicates that the minimum time between bangs is 25 ms when the Max scheduler is being used; when the *Scheduler in Audio Interrupt* option is checked in the DSP Status window, the minimum time between bangs is the time represented by the number of samples in the current input/output vector size.

Out right outlet: Sent when the input signal changes from non-zero to zero. The output will not happen more often than the time represented by the number of samples in the current input/output vector size.

Examples



Send a triggering Max message when a significant moment occurs in a signal

See Also

change~	Report signal direction
thresh~	Detect signal above a set level

Input

(mouse) Clicking on **ezadc~** toggles audio processing on or off. Audio on is represented by the object being hilighted.

int A non-zero number turns on audio processing in all loaded patches. 0 turns off audio processing in all loaded patches.

start Turns on audio processing in all loaded patches.

stop Turns off audio processing in all loaded patches.

local The word local, followed by 1, makes a click to turn on **ezadc~** equivalent to sending it the startwindow message. local 0 returns **ezadc~** to its default mode where a click to turn it on is equivalent to the start message.

startwindow Turns on audio processing only in the patch in which this **ezadc~** is located, and in subpatches of that patch. Turns off audio processing in all other patches.

Arguments

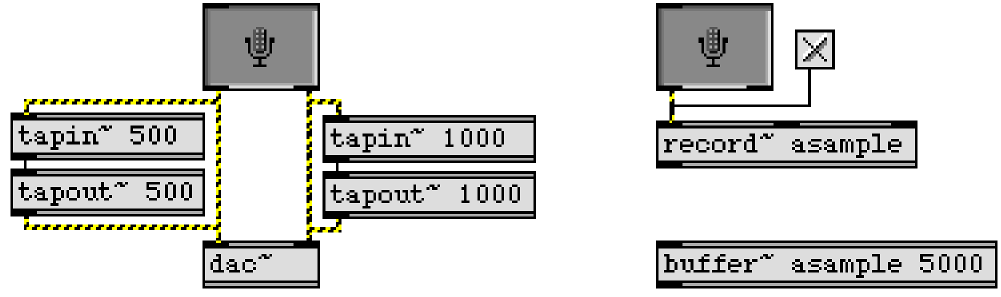
None.

Output

signal Out left outlet: Audio input from channel 1.

Out right outlet: Audio input from channel 2.

Examples



Audio input for processing and recording

See Also

ezdac~ Audio on/off; digital-to-analog converter
adc~ Audio in; analog-to-digital converter

ezdac~ Audio on/off; analog-to-digital converter

Input

signal — In left inlet: The signal is sent to audio output channel 1. The signal in each inlet must be between -1 and 1 to avoid clipping by the DAC.

In right inlet: The signal is sent to audio output channel 2.

(mouse) — Clicking on **ezdac~** toggles audio processing on or off. Audio on is represented by the object being hilighted.

int — A non-zero number turns on audio processing in all loaded patches. 0 turns off audio processing in all loaded patches.

start — Turns on audio processing in all loaded patches.

stop — Turns off audio processing in all loaded patches.

local — The word local, followed by 1, makes a click to turn on **ezdac~** equivalent to sending it the startwindow message. local 0 returns **ezdac~** to its default mode where a click to turn it on is equivalent to the start message.

startwindow — Turns on audio processing only in the patch in which this **ezdac~** is located, and in subpatches of that patch. Turns off audio processing in all other patches.

Arguments

None.

Output

None. The signal received in the inlet is sent to the corresponding audio output channel.

Examples

Switch audio on and off, send signal to the audio outputs

See Also

ezadc~ — Audio on/off; analog-to-digital converter

ezdac~

adc~	Audio out; digital-to-analog converter
Tutorial 3	Fundamentals: Wavetable oscillator

Fast Fourier transform

Input

signal — In left inlet: The real part of a complex signal that will be transformed.

In right inlet: The imaginary part of a complex signal that will be transformed.

If signals are connected only to the left inlet and left outlet, a real FFT (fast Fourier transform) will be performed. Otherwise, a complex FFT will be performed.

Arguments

int — Optional. The first argument specifies the number of points (samples) in the FFT. It must be a power of two. The default number of points is 512. The second argument specifies the number of samples between successive FFTs. This must be at least the number of points, and must be also be a power of two. The default interval is 512. The third argument specifies the offset into the interval where the FFT will start. This must either be 0 or a multiple of the signal vector size. **fft~** will correct bad arguments, but if you change the signal vector size after creating an **fft~** and the offset is no longer a multiple of the vector size, the **fft~** will not operate when signal processing is turned on.

Output

signal — Out left outlet: The real part of the Fourier transform of the input. The output begins after all the points of the input have been received.

Out middle outlet: The imaginary part of the Fourier transform of the input. The output begins after all the points of the input have been received.

Out right outlet: A sync signal that ramps from 0 to the number of points minus 1 over the period in which the FFT output occurs. You can use this signal as an input to the **index~** object to perform calculations in the frequency domain. When the FFT is not being sent out (in the case where the interval is larger than the number of points), the sync signal is 0.

Examples

Fast Fourier transform of an audio signal

See Also

ifft~	Inverse fast Fourier transform

index~	Sample playback without interpolation
Tutorial 24	Analysis: Using the FFT

ftom *Convert frequency to a MIDI note number*

The **ftom** object is not a signal object but it is useful for generating MIDI note values from frequency-based calculations required by signal objects such as **cycle~** and **phasor~**.

Input

float or int A frequency value. The corresponding MIDI pitch value (from 0 to 127) is sent out the outlet.

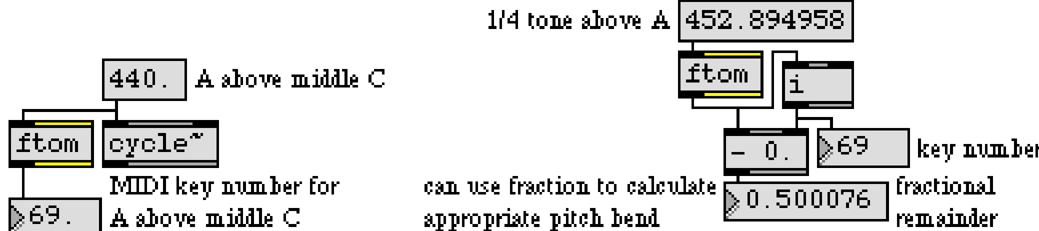
Arguments

None.

Output

float The MIDI pitch value that corresponds to the input frequency. Because most frequencies fall *between* exact tempered pitches, the output usually has a fractional part. Only the integer part will be used by MIDI objects which expect an int; the fractional part will be truncated. The fractional part can potentially be used to calculate an additional pitch offset for applying MIDI pitch bend.

Examples



Find the MIDI key number to play the same pitch as an MSP oscillator

See Also

mtof Convert a MIDI note number to frequency

The **function** object is not a signal object but it is useful for graphical input to the **line~** signal object.

Input

(mouse) You can use the mouse to draw points in a line segment function; the finished function can then be sent to a **line~** object for use as a control signal in MSP. Clicking on empty space in the **function** adds a breakpoint, which you can begin to move immediately by dragging (unless **function** has been sent the clickadd 0 message). Clicking on a breakpoint allows you to move the breakpoint by dragging (unless **function** has been sent the clickmove 0 message). The X and Y values of the breakpoint are displayed in the upper part of the object's box. Shift-clicking on a breakpoint deletes that point from the function. Command-clicking on a breakpoint toggles the sustain property of the point. Sustain points are outlined in white. Whenever an editing operation with the mouse is completed, a bang is sent out the right outlet.

Points with a Y value of 0 are outlined circles; other points are solid. This allows you to see at a glance whether a function starts or ends at Y = 0.

float or int The value is taken as an X value and outputs a corresponding Y value out the left outlet. The Y value is produced by linear floating-point interpolation of the function. If the X value lies outside the first or last breakpoint, the Y value is 0.

bang Triggers a list output of the current breakpoints from the middle-left outlet formatted for use by the **line~** object. As an example, if the **function** contained breakpoints at X = 1, Y = 0; X = 10, Y = 1; and X = 20, Y = 0, the output would be 0, 1 9 0 10

If there are any sustain points in the function, bang outputs a list of all the points up to the sustain point. Additional points in the function, up to a subsequent sustain point or the end point, whichever applies, can be output by sending the next message. See the description of the next and sustain messages for additional information.

next The next message continues a list output from the sustain point where the output of the last bang or next message ended. For instance, if the **function** contained breakpoints at (a) X = 1, Y = 0; (b) X = 10, Y = 1; and (c) X = 20, Y = 0, and point b was a sustain point, a bang message would output 0, 1 9 and a subsequent next message would output 1, 0 10. After a next message reaches the end point, a subsequent next message is equivalent to a bang message. next is also equivalent to a bang when no bang has been sent that reached a sustain point, or when a **function** contains no sustain points.

list If the list contains two values, a new point is added to the **function**. The first value is X, the second is Y.

If the list contains three values, an existing point in the **function** is modified. The first value is the index (starting at 0) of a breakpoint to modify, the second is the new X value for the breakpoint, and the third is the new Y value for the breakpoint. (If the index number in the list refers to a breakpoint that does not exist, the message is ignored.)

nth — The word nth, followed by a number, uses the number as the index (starting at 0) of a breakpoint, and outputs the Y value of the breakpoint out the left outlet. If no breakpoint with the specified index exists, no output occurs.

clear — The word clear by itself erases all existing breakpoints. The word clear can also be followed by one or more breakpoint indices (starting at 0) to clear selected breakpoints.

dump — Outputs a series of two-item lists, containing the X and Y values for each of the breakpoints, out the **function** object's middle-right outlet.

range — The word range, followed by two float or int values, sets the minimum and maximum display range for Y values. The actual values of breakpoints are not modified, so this message could cause breakpoints to disappear from view.

setrange — The word setrange, followed by two float or int values, sets the minimum and maximum display range for Y values, then modifies the Y values of all breakpoints so that they remain in the same place given the new range.

domain — The word domain, followed by a float or int value, sets the maximum displayed X value. The minimum value is always 0. The actual values of breakpoints are not modified, so this message could cause breakpoints whose X values are greater than the new maximum displayed X value to disappear.

setdomain — The word setdomain, followed by a float or int value, sets the maximum displayed X value, then modifies the X values of all breakpoints so that they remain in the same place given the new domain.

color — The word color, followed by a number between 0 and 15, sets the color of the displayed breakpoints to the specified color. The colors corresponding to the index are displayed in the **Color…** dialog in the Max menu.

(Color…) — You can change the color of breakpoints by selecting a **function** object in an unlocked Patcher window and choosing **Color…** from the Max menu.

clickadd — The message clickadd 0 prevents the user from adding new breakpoints by clicking. clickadd 1 allows the user to add new breakpoints. The default behavior allows the user to add new breakpoints. The current setting is saved with the object when its Patcher is saved.

clickmove The message clickmove 0 prevents the user from moving existing breakpoints by dragging them with the mouse. clickmove 1 allows the user to drag breakpoints. The default behavior allows the user to drag breakpoints. The current setting is saved with the object when its Patcher is saved.

fix The word fix, followed by a number specifying the index of a point and 0 or 1, prevents the user from changing the point if the second number is 1, and allows the user to change the point if the second number is 0. By default, points are moveable unless clickmove 0 has been sent to disable moving of all points.

sustain The word sustain, followed by number specifying the index of a point and 0 or 1, turns that point into a sustain point if the second number is 1, or into a regular point if the second number is 0. By default, points are regular (non-sustain). The behavior of sustain points is discussed in the description of the bang message above. You can also toggle the sustain property of a point by command-clicking on it.

(preset) You can save and restore the breakpoint settings of **function** using a **preset** object.

Arguments

None.

Output

float Out left outlet: The interpolated Y value is sent out in response to a float or int X value received in the inlet; or a stored Y value is sent out in response to an nth message.

list Out middle-left outlet: When bang is received, a float is sent out for the first stored Y value, followed by a list containing pairs of numbers indicating each subsequent stored Y value and its transition time (the difference between X and the previous X). This format is intended for input to the **line~** object.

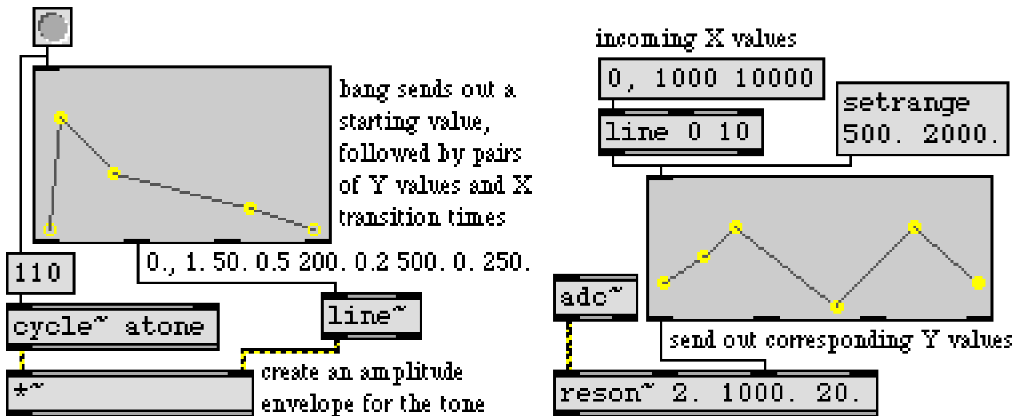
Out middle-right outlet: A series of two-item lists, containing the X and Y values of each of the **function**'s breakpoints, is sent out when a dump message is received.

bang Out right outlet: When a mouse editing operation is completed, a bang is sent out.

function

Graphical breakpoint function editor

Examples



Send line segment information to **line~**, *or look up (and interpolate) individual Y values*

See Also

line~	Ramp generator
Tutorial 7	Synthesis: Additive synthesis

gain~

Input

signal — In left inlet: The input signal to be scaled by the slider.

int — In left inlet: Sets the value of the slider, ramps the output signal to the level corresponding to the new value over the specified ramp time, and outputs the slider's value out the right outlet.

float — In left inlet: Converted to int.

In right inlet: Sets the ramp time in milliseconds. The default is 10 milliseconds.

bang — Sends the current slider value out the right outlet.

set — In left inlet: Sets the value of the slider, ramps the output signal to the level corresponding to the new value over the specified ramp time, but does not output the slider's value out the right outlet.

size — The word size, followed by a number, sets the range of **gain~** to the number. The values of the slider will then be 0 to the range value minus 1.

color — The word color, followed by a number from 0 to 15, sets the color of the striped center portion of **gain~** to one of 16 object colors, which are also available by choosing **Color…** from the Max menu.

Arguments

Three **gain~** parameters can be set by selecting it in an unlocked Patcher window and choosing **Get Info…** from the Max menu. The first is the range, the second is the base value, and the third is the increment. In the following expression that calculates the output scale factor based on the input value (the same as the **linedrive** object), the range is a, the base value is b, the increment is c, the input is x, e is the base of the natural logarithm (approx. 2.718282) and the output is y.

$$y = b\, e^{-a \log c}\, e^{x \log c}$$

For more information about these parameters, see the **linedrive** object.

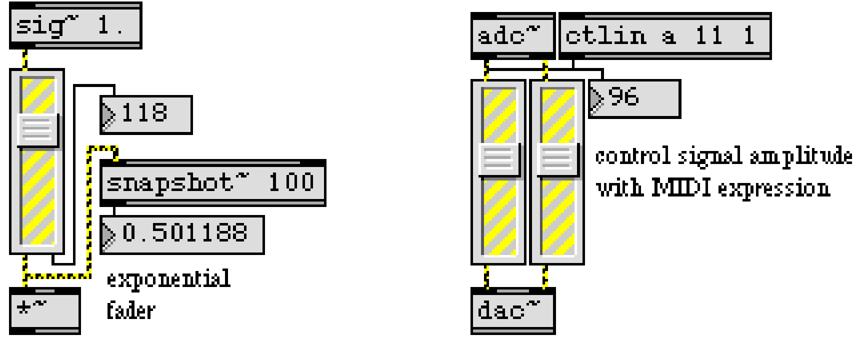
The default values of range (158), base (7.94231), and increment (1.071519) provide for a slider where 128 is full scale (multiplying by 1.0), 0 produces a zero signal, and 1 is 75.6 dB below the value at 127. A change of 10 in the slider produces a 6 dB change in the output. In addition, since the range is 158, slider values from 129 to 157 provide 17.4 dB of headroom. When the slider is at 157, the output signal is 17.4 dB louder than the input signal.

Output

signal — Out left outlet: The input signal, scaled by the current slider value as *x* in the equation shown above.

int — Out right outlet: The current slider value, when dragging on the slider with the mouse or when **gain~** receives an int or float in its left inlet.

Examples



Specialized fader to scale a signal exponentially or logarithmically

See Also

linedrive — Scale integers for use with **line~**

Input

int — In left inlet: Determines the outlet that will send out the signal coming in the right inlet. If the number is 0 or negative, the right inlet is shut off and a zero signal is sent out. If the number is greater than the number of outlets, the signal is sent out the rightmost outlet. If a signal is connected to the left inlet, **gate~** ignores ints or floats received in its left inlet.

float — Converted to int.

signal — In left inlet: If a signal is connected to the left inlet, **gate~** operates in a mode that uses signal values to determine the outlet that will receive its *input signal* (the signal coming in the right inlet). If the signal coming in the left inlet is 0 or negative, the inlet is shut off and a zero signal is sent out. If it is greater than or equal to 1, but less than 2, the input signal goes to the left outlet. If the signal is greater than or equal to 2 but less than 3, the input signal goes to the next outlet to the right, and so on. If the signal in the left inlet is greater than the number of outlets, the rightmost outlet is used.

In right inlet: The input signal to be passed through to one of the **gate~** object's outlets, according to the most recently received int or float in the left inlet, or the value of the signal coming in the left inlet.

If the signal network connected to the right inlet of **gate~** contains a **begin~** object—and a signal is not connected to the left inlet of the **gate~**—all processing between the **begin~** outlet and the **gate~** inlet will be turned off when **gate~** is shut off.

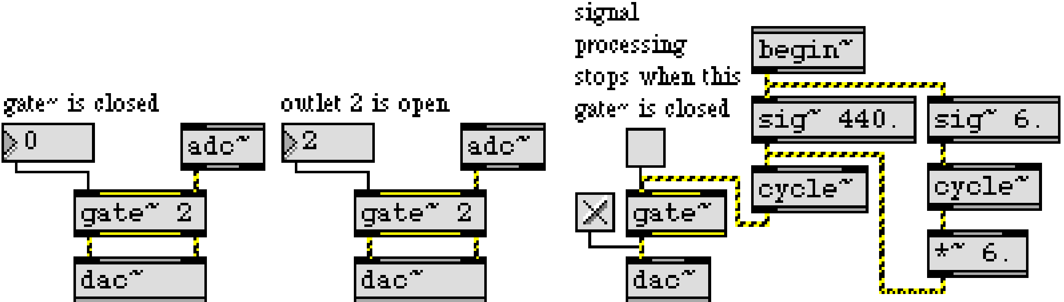
Arguments

int — Optional. The first argument specifies the number of outlets. The default is 1. The second argument sets the outlet that will initially send out the input signal. The default is 0, where all signals are shut off and zero signals are sent out all outlets. If a signal is connected to the left inlet, the second argument is ignored.

Output

signal — Depending on the value of the left inlet (either signal or number), one of the object's outlets will send out the input signal and rest will send out zero signals, or (if the inlet is closed) all outlets will send out zero signals.

Examples



gate~ *routes the input signal to one of its outlets, or shuts it off entirely*

See Also

selector~	Assign one of several inputs to an outlet
begin~	Define a switchable part of a signal network
Tutorial 4	Fundamentals: Routing signals

Input

signal — In left inlet: Defines the sample increment for playback of a sound from a **buffer~**. A sample increment of 0 stops playback. A sample increment of 1 plays the sample at normal speed. A sample increment of -1 plays the sample backwards at normal speed. A sample increment of 2 plays the sample at twice the normal speed. A sample increment of .5 plays the sample at half the normal speed. The sample increment can change over time for vibrato or other types of speed effects.

If a loop start and end have been defined for **groove~** and looping is turned on, when the sample playback reaches the loop end the sample position is set to the loop start and playback continues at the current sample increment.

In middle inlet: Sets the starting point of the loop in milliseconds.

In right inlet: Sets the end point of the loop in milliseconds.

int or float — In left inlet: Sets the sample playback position in milliseconds. 0 sets the playback position to the beginning.

In middle inlet: Sets the starting point of the loop in milliseconds. If a signal is connected to the inlet, int and float numbers are ignored.

In right inlet: Sets the end point of the loop in milliseconds. If a signal is connected to the inlet, int and float numbers are ignored.

startloop — Causes **groove~** to begin sample playback at the starting point of the loop. If no loop has been defined, **groove~** begins playing at the beginning.

loop — The word loop, followed by 1, turns on looping. loop 0 turns off looping. By default, looping is off.

set — The word set, followed by a symbol, switches the **buffer~** object containing the sample to be used by **groove~** for playback.

Arguments

symbol — Obligatory. Names the **buffer~** object containing the sample to be used by **groove~** for playback.

int — Optional. A second argument may specify the number of output channels: 1, 2, or 4. The default number of channels is 1. If the **buffer~** being played has fewer channels than the number of **groove~** output channels, the extra channels output a zero signal. If the **buffer~** has more channels, channels are mixed.

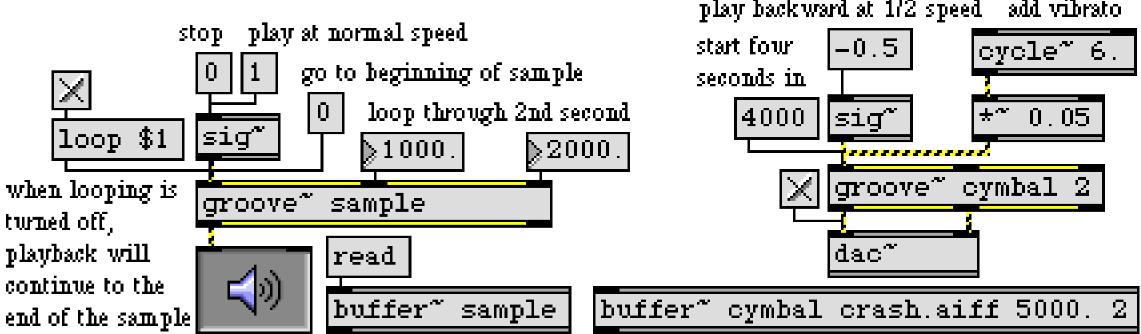
Output

signal — Out left outlet: Sample output. If **groove~** has two or four output channels, the left outlet plays the left channel of the sample.

Out middle outlets: Sample output. If **groove~** has two or four output channels, the middle outlets play the channels other than the left channel.

Out right outlet: Sync output. During the loop portion of the sample, this outlet outputs a signal that goes from 0 when the loop starts to 1 when the loop ends.

Examples



See Also

buffer~	Store a sound sample
play~	Position-based sample playback
record~	Record sound into a buffer
Tutorial 14	Sampling: Playback with loops
Tutorial 20	MIDI control: Sampler

Input

signal — In left inlet: The real part of a complex signal that will be inverse transformed.

In right inlet: The imaginary part of a complex signal that will be inverse transformed.

If signals are connected only to the left inlet and left outlet, a real IFFT (inverse fast Fourier transform) will be performed. Otherwise, a complex IFFT will be performed.

Arguments

int — Optional. The first argument specifies the number of points (samples) in the IFFT. It must be a power of two. The default number of points is 512. The second argument specifies the number of samples between successive IFFTs. This must be at least the number of points, and must be also be a power of two. The default interval is 512. The third argument specifies the offset into the interval where the IFFT will start. This must either be 0 or a multiple of the signal vector size. **ifft~** will correct bad arguments, but if you change the signal vector size after creating an **ifft~** and the offset is no longer a multiple of the vector size, the **ifft~** will not operate when signal processing is turned on.

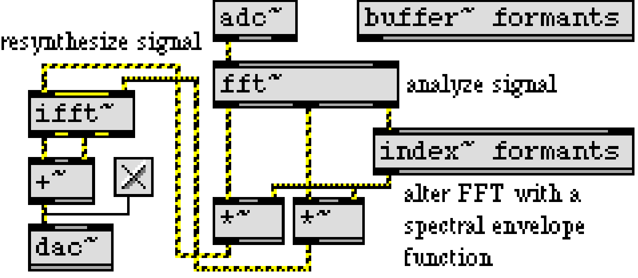
Output

signal — Out left outlet: The real part of the inverse Fourier transform of the input. The output begins after all the points of the input have been received.

Out middle outlet: The imaginary part of the inverse Fourier transform of the input. The output begins after all the points of the input have been received.

Out right outlet: A sync signal that ramps from 0 to the number of points minus 1 over the period in which the IFFT output occurs. When the IFFT is not being output (in the case where the interval is larger than the number of points), the sync signal is 0.

Examples



Using **fft~** *and* **ifft~** *for analysis and resynthesis*

See Also

fft~	Fast Fourier transform
Tutorial 24	Analysis: Using the FFT

Input

signal In left inlet: The sample index to read from a **buffer~** object's sample memory.

int In right inlet: The channel (1-4) of the **buffer~** to use for output. By default, **index~** uses the first channel of the **buffer~**.

set The word set, followed by the name of a **buffer~** object, causes **index~** to read from that **buffer~**.

(mouse) Double-clicking on **index~** opens an editing window where you can view the contents of its associated **buffer~** object.

Arguments

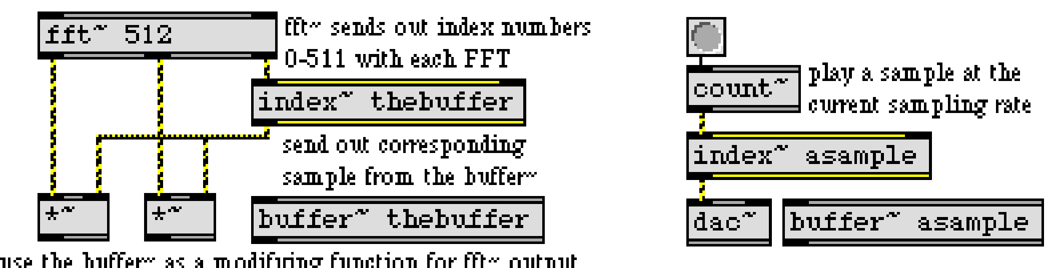
symbol Obligatory. Names the **buffer~** object whose sample memory is used by **index~** for playback.

int Optional. Following the name of the **buffer~**, you may specify which channel to use within the associated **buffer~**. The default channel is 1.

Output

signal The output consists of samples at the sample indices specified by the input. No interpolation is performed if the input sample index is not an integer.

Examples



Look up specific samples in the **buffer~**, *using* **index~**

See Also

buffer~	Store a sound sample
fft~	Fast Fourier transform
Tutorial 13	Sampling: Recording and playback

Input

bang — In left inlet: Causes a report of information about a sample contained in the associated **buffer~** object.

(mouse) — Double-clicking on **info~** opens an editing window where you can view the contents of its associated **buffer~** object.

Arguments

symbol — Obligatory. Names the **buffer~** object for which **info~** will report information.

Output

Most of the information reported by **info~** is taken from the sound file most recently read into the associated **buffer~**. If this information is not present, only the sampling rate is sent out the left outlet. No output occurs for any item that's missing from the sound file.

float — Out left outlet: The sampling rate of the sample.

Out 3rd outlet: Sustain loop start, in milliseconds.

Out 4th outlet: Sustain loop end, in milliseconds.

Out 5th outlet: Release loop start, in milliseconds.

Out 6th outlet: Release loop end, in milliseconds.

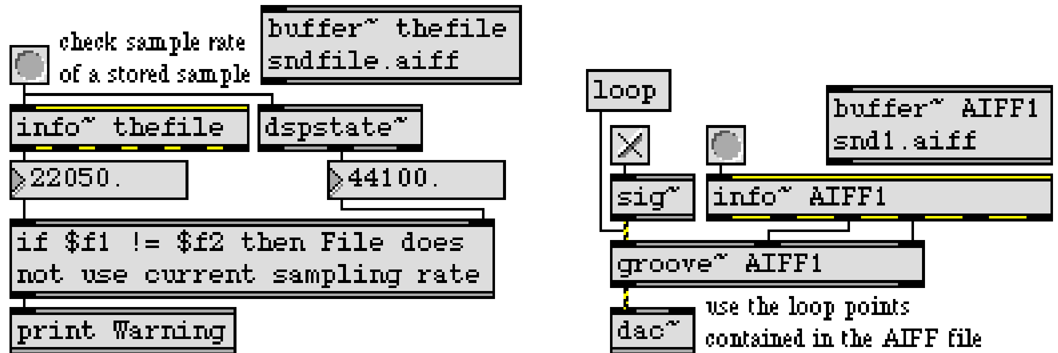
Out 7th outlet: Total time of the associated **buffer~** object, in milliseconds.

Out 8th outlet: Name of the most recently read audio file.

list — Out 2nd outlet: Instrument information about the sample, as follows:

1. The MIDI pitch of the sample.
2. The detuning from the original MIDI note number of the sample, in pitch bend units.
3. The lowest MIDI note number to use when playing this sample.
4. The highest MIDI note number to use when playing this sample.
5. The lowest MIDI velocity to use when playing this sample.
6. The highest MIDI velocity to use when playing this sample.
7. The gain of the sample (0-127).

Examples



Check sample rate of a sample; use other information contained in a sample

See Also

buffer~	Store a sound sample
mstosamps~	Convert milliseconds to samples
sfinfo~	Report sound file information
Tutorial 14	Sampling: Playback with loops
Tutorial 20	MIDI control: Sampler

Distort a sawtooth waveform

Input

signal — In left inlet: The input to **kink~** should be a sawtooth waveform output from a **phasor~** object that repeatedly goes from 0 to 1.

In right inlet: The multiplier that affects the slope of the output between an output (Y) value of 0 and 0.5. After the output reaches 0.5, the waveform will increase to 1 so that the entire output moves from 0 to 1 in the same period of time as the input. A slope multiplier of 1 (the default) produces no distortion Slope multipliers below 1 have a slower rise to 0.5 than the input, and slope multipliers above 1 have a faster rise to 0.5 than the input.

float — In right inlet: Same as signal. If a signal is attached to the right inlet, float input is ignored.

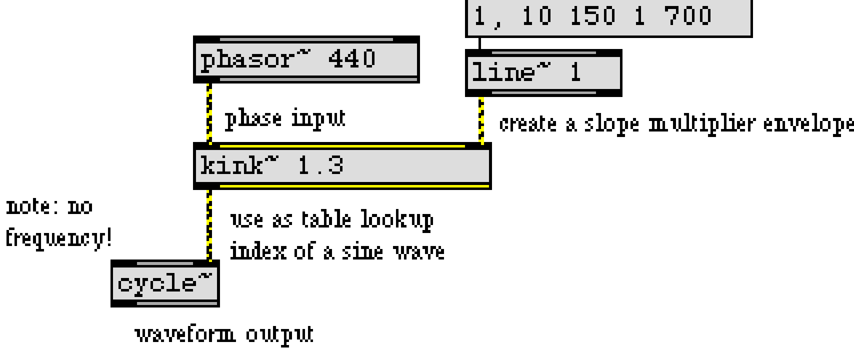
Arguments

float — Optional. Sets the default slope multiplier. If a signal is attached to the right inlet, this argument is ignored.

Output

signal — The output of **kink~** should be fed to the right inlet of **cycle~** (at zero frequency) to produce a distorted sine wave (a technique known as *phase distortion synthesis*). As the slope multiplier in the right inlet of **kink~** deviates from 1, additional harmonics are introduced into the waveform output of **cycle~**. If the slope multiplier is rapidly increased and then decreased using a **line~**, the output of **cycle~** may resemble an attack portion of an instrumental sound.

Examples



Typical use of **kink~** *between* **phasor~** *and* **cycle~**.

See Also

phasor~	Generate a sawtooth wave
cycle~	Table lookup oscillator

The **linedrive** object is not a signal object but it is useful for controlling the **line~** object.

Input

int or float In left inlet: The number is converted according to the following expression

$$y = b\, e^{-a \log c}\, e^{x \log c}$$

where x is the input, y is the output, a, b, and c are the three typed-in arguments, and e is the base of the natural logarithm (approx. 2.718282).

The output is a two-item list containing y followed by the delay time most recently received in the right inlet.

int In right inlet: Sets the current delay time appended to the scaled output. A connected **line~** object will ramp to the new target value over this interval.

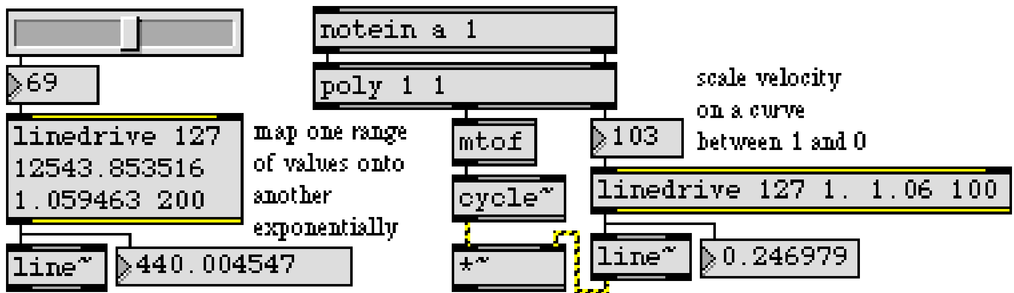
Arguments

int or float Obligatory. The first argument is the maximum input value, the second argument is the maximum output value. The third argument specifies the nature of the scaling curve. The smaller the third argument (from 1 down toward 0), the more logarithmic the curve is; the larger it is (from 1 upward), the more steeply exponential the curve is. Only positive numbers are appropriate for the third argument. The fourth argument is the initial delay time in milliseconds. This value can be changed via the right inlet.

Output

list When an int or float is received in the left inlet, a list is sent out containing a scaled version of the input (see the formula above) and the current delay time.

Examples



Send information for an exponential curve to **line~**

See Also

line~ Ramp generator

line~ — *Ramp generator*

Input

list The first number specifies a target value and the second number specifies a total amount of time (in milliseconds) in which **line~** should reach the target value. In the specified amount of time, **line~** generates a ramp signal from its current value to the target value.

line~ accepts up to 64 target-time pairs in a list, to generate compound ramps. (An example would be 0 1000 1 1000, which would go from the current value to 0 in a second, then to 1 in a second.) Once one of the ramps has reached its target value, the next one starts. A subsequent list, float, or int in the left inlet clears all ramps yet to be generated.

float or int In left inlet: The number is the target value, to be arrived at in the time specified by the number in the right inlet. If no time has been specified since the last target value, the time is considered to be 0 and the output signal jumps immediately to the target value.

In right inlet: The number is the time, in milliseconds, in which the output signal will arrive at the target value.

Arguments

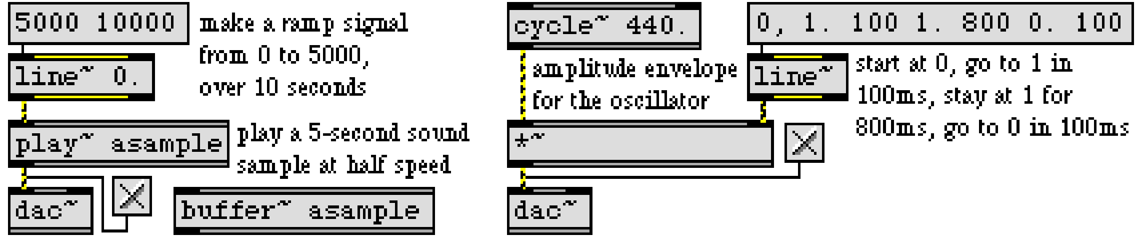
float or int Optional. Sets an initial value for the signal output. The default value is 0.

Output

signal Out left outlet: The current target value, or a ramp moving toward the target value according to the currently stored value and the target time.

bang Out right outlet. When **line~** has finished generating all of its ramps, bang is sent out.

Examples



Linearly changing signal, or a function made up of several line segments

See Also

curve~	Exponential ramp generator
Tutorial 2	Fundamentals: Adjustable oscillator

Input

signal — In left inlet: **log~** sends out a signal that is the logarithm of the input signal, to the base specified by the typed-in argument or the value most recently received in the right inlet. If a value in the signal is less than or equal to 0, **log~** sends out a value of 0.00000001.

float or int — In right inlet: Sets the base of the logarithm. The default is 0, which is equivalent to the natural logarithm (log to the base *e*, or 2.71828182). log to the base of 1 is always 0.

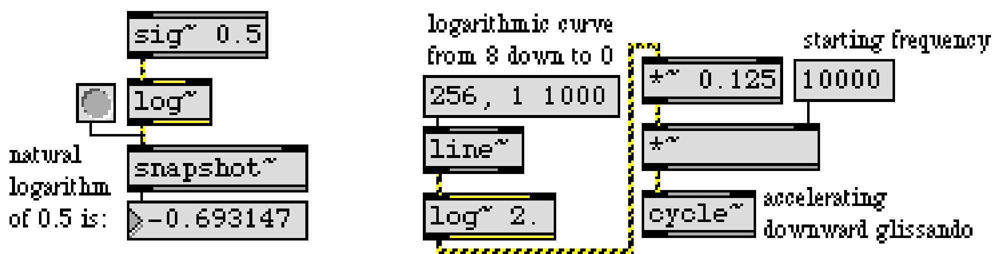
Arguments

float or int — Optional. Sets the base of the logarithm. The default value is 0.

Output

signal — The logarithm of the input signal to the base specified by the initial argument or the value most recently received in the right inlet.

Examples



Logarithm of a signal, to a specified base; can be used for creating curves

See Also

pow~	Signal power function
curve~	Exponential ramp generator
sqrt~	Square root of a signal

Transfer function lookup table

Input

signal — In left inlet: Signal values are mapped by amplitude to values stored in a **buffer~**. Each sample in the incoming signal within the range -1 to 1 is mapped to a corresponding value in the current table size number of samples of the **buffer~**. Signal values between -1 and 0 are mapped to the first half of the total number of samples after the current sample offset. Signal values between 0 and 1 are mapped to the next half of the samples. Input amplitude exceeding the range from -1 to 1 results in an output of 0.

In middle inlet: Sets the offset into the sample memory of a **buffer~** used to map samples coming in the left inlet. The sample at the specified offset corresponds to an input value of -1.

In right inlet: Sets the number of samples in a **buffer~** used for the table. Samples coming in the left inlet between -1 and 1 will be mapped by amplitude to the specified range of samples. The default value is 512. **lookup~** changes the table size before it computes each vector but not within a vector. It uses the first sample in a signal vector coming in the right inlet as the table size.

int or float — The settings of offset and table size can be changed with an number in the middle or right inlets. If a signal is connected to one of these inlets, a number in the corresponding inlet is ignored.

set — The word set, followed by a symbol, changes the associated **buffer~** object.

(mouse) — Double-clicking on **lookup~** opens an editing window where you can view the contents of its associated **buffer~** object.

Arguments

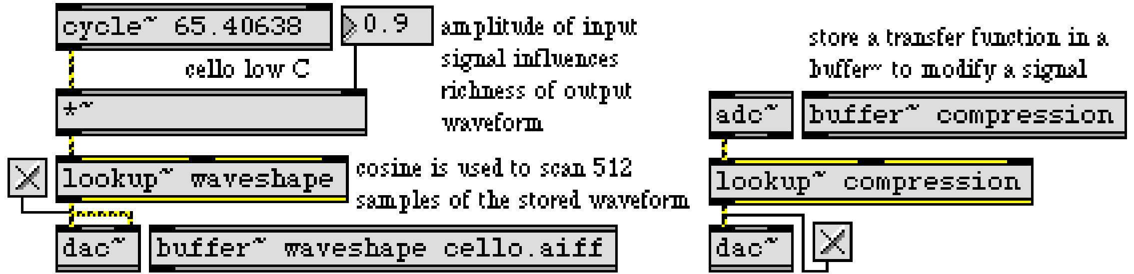
symbol — Obligatory. Names the **buffer~** object whose sample memory is used by **lookup~** for table lookup.

int — Optional. After the **buffer~** name, you may specify the sample offset in the sample memory of the **buffer~** used for a signal value of -1. The default offset is 0. The offset value is followed by an optional table size that defaults to 512. **lookup~** always uses the first channel in a multi-channel **buffer~**.

Output

signal — Each sample in the incoming signal within the range -1 to 1 is mapped to a corresponding position in the current table size number of samples of the named **buffer~** object, and the stored value is sent out..

Examples



See Also

buffer~	Store a sound sample
peek~	Read and write sample values
Tutorial 12	Synthesis: Waveshaping

lores~ *Resonant lowpass filter*

Input

signal In left inlet: Any signal to be filtered.

In middle inlet: Sets the lowpass filter cutoff frequency.

In right inlet: Sets a “resonance factor” between 0 (minimum resonance) and 1 (maximum resonance). Values very close to 1 may produce clipping with certain types of input signals.

int or float An int or float can be sent in the middle or right inlets to change the cutoff frequency or resonance. If a signal is connected one of the inlets, a number received in that inlet is ignored.

clear Clears the filter’s memory. Since **lores~** is a recursive filter, this message may be necessary to recover from blowups.

Arguments

int or float Optional. Numbers set the initial cutoff frequency and resonance. The default values for both are 0. If a signal is connected to the middle or right inlet, the argument corresponding to that inlet is ignored.

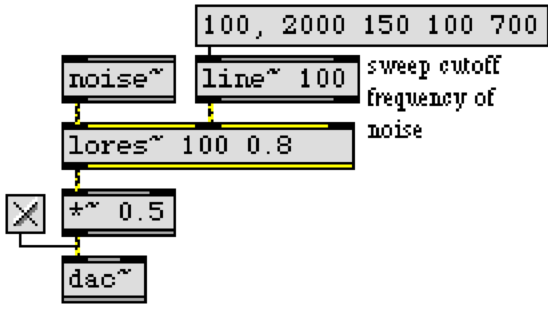
Output

signal The filtered input signal. The equation of the filter is

$$y_n = scale * x_n - c1 * y_{n-1} + c2 * y_{n-2}$$

where *scale*, *c1*, and *c2* are parameters calculated from the cutoff frequency and resonance factor.

Examples



Specify cutoff frequency and resonance of lowpass filter

See Also

biquad~ Two pole, two zero filter
reson~ Resonant bandpass filter

meter~

Input

signal — The peak amplitude of the incoming signal is displayed by the LEDs of the on-screen level meter.

interval — The word interval, followed by a number, sets the update time interval, in milliseconds, of the **meter~** display. The minimum update interval is 10 milliseconds, the maximum is 2 seconds, and the default is 100 milliseconds. This message also sets the rate at which **meter~** sends out the peak value received in that time interval.

(mouse) — When the Patcher window is unlocked, you can re-orient a **meter~** from horizontal to vertical by dragging its resize area and changing its shape.

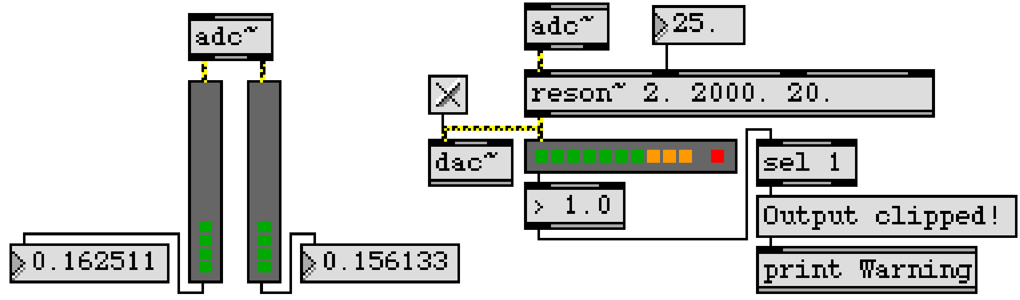
Arguments

None.

Output

float — The peak (absolute) value received in the previous update interval is sent out the outlet when audio processing is on.

Examples



meter~ *displays and sends out the peak amplitude of a signal*

See Also

scope~	Signal oscilloscope
Tutorial 22	Analysis: Viewing signal data

mstosamps~ *Convert milliseconds to samples*

Input

float or int Millisecond values received in the inlet are converted to a number of samples at the current sampling rate and sent out the object's right outlet. The output might contain a fractional number of samples. For example, at 44.1 kHz sampling rate, 3.2 milliseconds is 141.12 samples.

signal Incoming millisecond values in the signal are converted to a number of samples at the current sampling rate and output as a signal out the **mstosamps~** object's left outlet. The output may contain a fractional number of samples.

Arguments

None.

Output

signal Out left outlet: The number of samples corresponding to the millisecond values in the input signal.

float Out right outlet: The number of samples corresponding to the millisecond value received as a float or int in the inlet.

Examples

Time expressed in milliseconds comes out expressed in samples

See Also

dspstate~ Report current DSP settings
sampstoms~ Convert samples to milliseconds

The **mtof** object is not a signal object but it is useful for controlling signal objects that use frequency—such as **cycle~** and **phasor~**—from MIDI.

Input

float or int   A MIDI note number value from 0 to 127. The corresponding frequency is sent out the outlet.

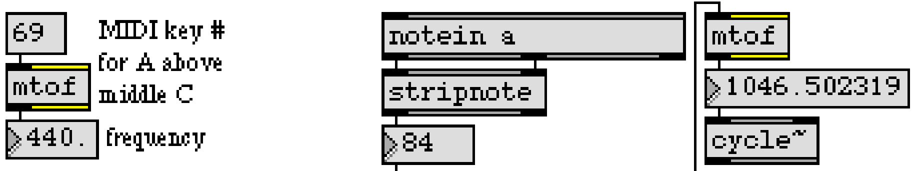
Arguments

None.

Output

float   The frequency corresponding to the received MIDI pitch value.

Examples



Use MIDI note number to provide frequency value for an oscillator

See Also

ftom   Convert frequency to a MIDI note number
Tutorial 19   MIDI control: Synthesizer

mute~ *Disable signal processing in a subpatch*

Input

int 1 turns off the signal processing in all objects contained in the subpatch connected to the **mute~** object's outlet, 0 turns it back on.

Arguments

None.

Output

Connect the **mute~** object's outlet to any inlet of a subpatch you wish to control. You can connect **mute~** to as many subpatch objects as you wish; however, **mute~** does *not* work with **bpatcher**s.

Examples

You can mute all processing in any patcher or other subpatch

See Also

begin~	Define a switchable part of a signal network
pass~	Define subpatch signal output
Tutorial 5	Fundamentals: Turning signals on & off

Input

None.

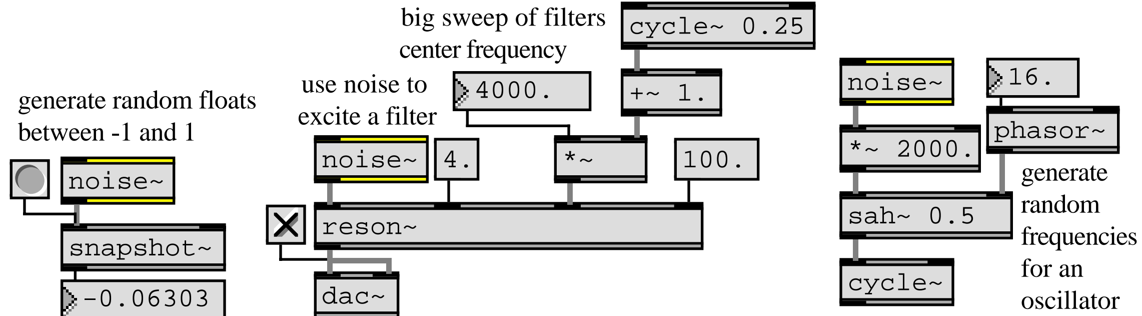
Arguments

None.

Output

signal The **noise~** object generates a signal consisting of uniformly distributed random (white noise) values between -1 and 1.

Examples



Random samples create white noise, which can be filtered in various ways

See Also

biquad~ Two-pole, two-zero filter
reson~ Resonant bandpass filter
Tutorial 3 Fundamentals: Wavetable oscillator

normalize~ *Scale on the basis of maximum amplitude*

Input

signal In left inlet: The input signal is *normalized*—scaled so that its peak amplitude is equal to a specified maximum.

In right inlet: The maximum output amplitude; an over-all scaling of the output.

float In right inlet: The maximum output amplitude may be sent as a float instead of a signal. If a signal is connected to the right inlet, a float received in the right inlet is ignored.

reset In left inlet: The word reset, followed by a number, resets the maximum input amplitude to the number. If no number follows reset, or if the number is 0, the maximum input amplitude is set to 0.000001.

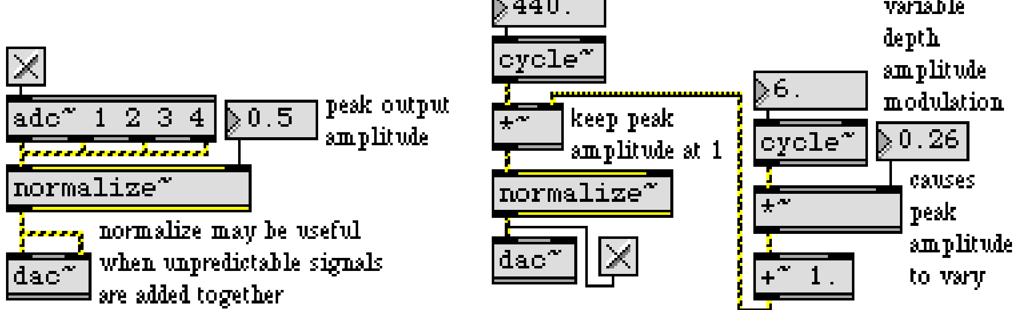
Arguments

float Optional. The initial maximum output amplitude. The default is 1.

Output

signal The input signal is scaled by the maximum output amplitude divided by the maximum input amplitude.

Examples



When precise scaling factor varies or is unknown, **normalize~** *sets peak amplitude*

See Also

*~ Multiply two signals

number~

number~ has two different display modes. In “Signal Monitor Mode” it displays the value of the signal received in the left inlet. In “Signal Output Mode” it displays the value of the float or int most recently received in the left inlet, or entered directly into the **number~** box (the signal being sent out the left outlet).

Input

signal — Any signal, the value of which is sampled and sent out the right outlet at regular intervals. When **number~** is in Signal Monitor display mode, the signal value is displayed.

float — In left inlet: The value is sent out the left outlet as a constant signal. When **number~** is in Signal Output display mode, the value is displayed. If the current ramp time is non-zero, the output signal will ramp between its previous value and the newly set value.

In right inlet: Sets a ramp time in milliseconds. The default time is 0.

int — Converted to float.

list — The first number sets the value of the signal sent out the left outlet, and the second number sets the ramp time in milliseconds.

(mouse) — Clicking on the triangular area at the left side of **number~** will toggle between Signal Monitor display mode (green waveform) and Signal Output display mode (yellow or green downward arrow). When in Signal Output display mode, clicking in the area that displays the number changes the value of the signal sent out the left outlet of **number~** and/or selects it for typing.

(typing) — When a **number~** is highlighted (indicated by a yellow downward arrow), numerical keyboard input changes its value. Clicking the mouse or pressing Return or Enter stores a pending typed number and sends it out the left outlet as the new signal value.

allow — The word allow, followed by a number, sets what display modes can be used. allow 1 restricts **number~** to signal output display mode. allow 2 restricts **number~** to input monitor display mode. allow 3 allows both modes, and lets the user switch between them by clicking on the left triangular area of **number~**.

mode — The word mode, followed by a number, sets the current display mode, if it is currently allowed (see the allow message). mode 1 sets signal output display mode. mode 2 sets signal input monitor display mode.

min — The word min, followed by an optional number, sets the minimum value of **number~** for signal output. Note that unlike a floating-point number box, the minimum value of **number~** is not restricted to being an integer value. If the word min is not followed by a number, any minimum value is removed.

max — The word max, followed by an optional number, sets the maximum value of **number~** for signal output. Note that unlike a floating-point number box, the maximum value of **number~** is not restricted to being an integer value. If the word max is not followed by a number, any maximum value is removed.

interval — The word interval, followed by a number, sets the sampling interval in milliseconds. This controls the rate at which the display is updated when **number~** is input monitor display mode, as well as the rate that numbers are sent out the object's right outlet.

flags — The word flags, followed by a number, sets characteristics of the appearance and behavior of **number~**. The characteristics (which are described under Arguments. below) are set by adding together values that designate the desired options, as follows: 4=**Bold type**, 64=**Send on mouse-up only**, 128=**Can't change with mouse**. For example, flags 196 would set all of these options.

Arguments

You can set minimum and maximum limits on the output signal value by selecting **number~** in an unlocked Pather and choosing **Get Info…** from the Max menu. When the patch is loaded, the signal value of **number~** is its last stored value. This is different from the **number box**, where the initial value is 0 or the minimum if defined. A newly created **number~** has no minimum or maximum signal output value.

Other display options available in the Get Info dialog box are: **Draw in Bold** (to display in bold typeface), **Send on Mouse Up** (to change the signal output value only when the mouse is released when dragging on **number~**), and **Can't Change** (to disallow changes with the mouse).

The allowable display modes can be set by checking or unchecking **Signal Monitor Mode** and **Signal Input Mode**. At least one mode must be checked.

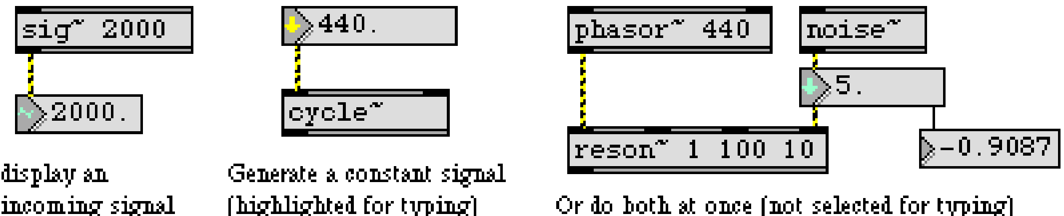
The font and size of a selected **number~** can be changed with the Font menu.

Output

signal — Out left outlet: When audio is on, **number~** sends a constant signal out its left outlet equal to the number most recently received in the left inlet (or entered by the user). It sends out this value independent of its signal input, and whether or not it is currently in Signal Output display mode. If the ramp time most recently received in the right inlet is set to a non-zero value, the output will interpolate between its previous value and a newly set value over the specified time.

float — Out right outlet: Samples of the input signal are sent out at a rate specified by the interval message.

Examples



Several uses for the **number~** *object*

See Also

line~	Ramp generator
sig~	Constant signal of a number
snapshot~	Convert signal values to numbers
Tutorial 22	Analysis: Viewing signal data

Input

signal Connect a signal to **pass~** before it is sent out an **outlet** of a subpatch. Normally, the signal is passed directly from input to output. However, when the audio in the subpatch is disabled using **mute~** or the enable 0 message to **pcontrol**, **pass~** will send a zero signal out its outlet.

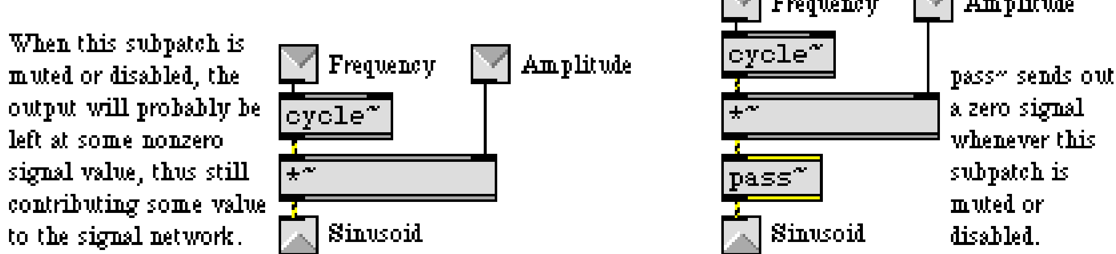
Arguments

None.

Output

signal When the audio in a subpatch containing **pass~** is enabled, the output is the same as the input. When the audio is disabled using **mute~** or the enable 0 message to **pcontrol**, the output is a zero signal.

Examples



pass~ *ensures that a muted signal is fully silenced*

See Also

mute~	Disable signal processing in a subpatch
Tutorial 5	Fundamentals: Turning audio signals on and off

The **peek~** object will function even when the audio is not turned on. You can use **peek~** to treat **buffer~** as a floating-point version of the Max **table** object in non-signal applications.

Input

int — In left inlet: A sample index into the associated **buffer~** object's sample memory. The value stored in the **buffer~** at that index is sent out **peek~**'s outlet. However, if a value has just been received in the middle inlet, **peek~** stores that value in the **buffer~** at the specified sample index, rather than sending out a number. If the number received in the left inlet specifies a sample index that does not exist in the **buffer~**'s currently allocated memory, nothing happens.

In middle inlet: Converted to float.

In right inlet: A channel (from 1 to 4) specifying the channel of a multi-channel **buffer~** to be used for subsequent reading or writing operations.

float — In left inlet: Converted to int.

In middle inlet: A sample value to be stored in the associated **buffer~** . The next sample index received in the left inlet causes the sample value to be stored at the index.

In right inlet: Converted to int.

list — In left inlet: The second number is stored in the associated **buffer~** at the sample index specified by the first number. If a third number is present in the list, it sets the channel of a multi-channel **buffer~** in which the value will be stored. Otherwise, the most recently set channel is used.

Note that for int, float, and list, if the message refers to a sample index that does not exist in the **buffer~**'s sample memory, nothing happens. You can ensure that memory is allocated to the **buffer~** by reading an existing file into it, by typing in a duration argument, or by setting its memory allocation with the size message.

set — In left inlet: The word set, followed by the name of a **buffer~** object, associates **peek~** with that newly named **buffer~** object.

(mouse) — Double-clicking on **peek~** opens an editing window where you can view the contents of its associated **buffer~** object.

Arguments

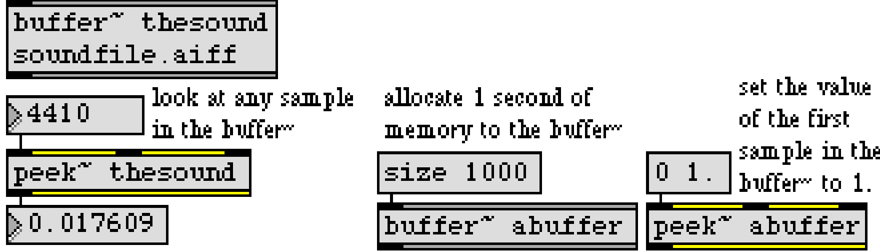
symbol — Obligatory. Names the **buffer~** object whose sample memory is used by **peek~** for reading and writing.

int Optional. Following the **buffer~** name, you can type in a number to specify the channel in a multi-channel **buffer~** to use for subsequent reading or writing operations. The default is 1.

Output

float The sample value in a **buffer~**, located at the table index specified by a float or int received in the left inlet, is sent out **peek~**'s outlet.

Examples



Peek at samples in a **buffer~**, *and/or set the value of the samples*

See Also

buffer~	Store a sound sample
poke~	Write sample values by index
table	Store and graphically edit an array of numbers

Input

signal — In left inlet: Sets the frequency of the sawtooth waveform.

int or float — In left inlet: Sets the frequency of the sawtooth waveform. If a signal is connected to this inlet, ints and floats are ignored.

In right inlet: Sets the phase of the waveform (from 0 to 1). The signal output continues from this value.

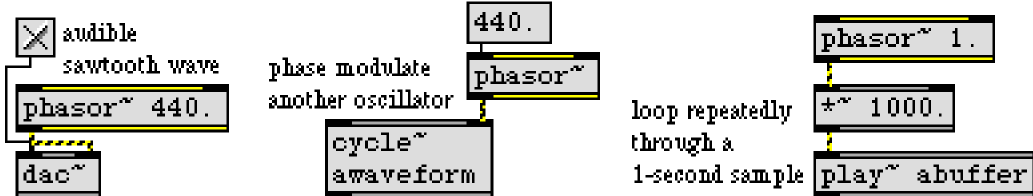
Arguments

int or float — Optional. Sets the initial frequency of the waveform. If a signal is connected to the left inlet, the argument is ignored.

Output

signal — Sawtooth waveform that increases from 0 to 1 repeatedly at the specified frequency.

Examples



A repeating ramp is useful both at audio and at sub-audio frequencies

See Also

cycle~	Table lookup oscillator
line~	Ramp generator
wave~	Variable-size table lookup oscillator
Tutorial 3	Analysis: Wavetable oscillator

Input

signal — In left inlet: The position (in milliseconds) into the sample memory of a **buffer~** object from which to play. If the signal is increasing over time, **play~** will play the sample forward. If it is decreasing, **play~** will play the sample backward. If it remains the same, **play~** outputs the same sample repeatedly, which is equivalent to a DC offset of the sample value.

set — The word set, followed by the name of a **buffer~** object, uses that **buffer~** for playback.

Arguments

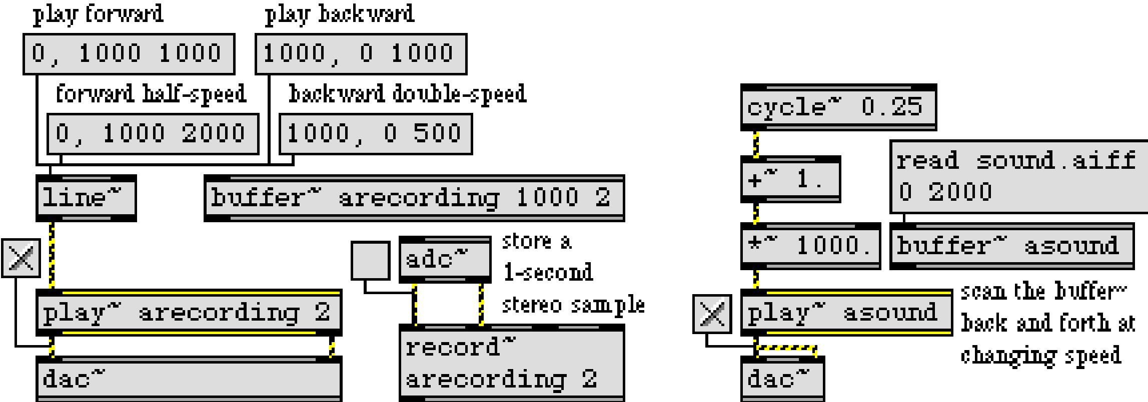
symbol — Obligatory. Names the **buffer~** object whose sample memory is used by **play~** for playback.

int — Optional, after the name argument. Specifies the number of output channels: 1, 2, or 4. The default number of channels is one. If the **buffer~** being played has fewer channels than the number of **play~** output channels, the extra channels output a zero signal. If the **buffer~** has more channels, channels are mixed.

Output

signal — Sample output read from a **buffer~**. If **play~** has two or four output channels, the left outlet's signal contains the left channel of the sample, and the other outlets' signals contain the additional channels.

Examples



play~ *is usually driven by a ramp signal from* **line~**, *but other signals create novel effects*

See Also

buffer~	Store a sound sample
groove~	Variable-rate looping sample playback
record~	Record a sample
Tutorial 13	Sampling: Recording and playback

Input

signal — In left inlet: Signal values you want to write into a **buffer~**.

In middle inlet: The sample index where values from the signal in the left inlet will be written. If the signal coming into the middle inlet has a value of -1, no samples are written.

float — Like the **peek~** object, **poke~** can write float values into a **buffer~**. Note, however, that the left two inlets are reversed on the **poke~** object compared to the **peek~** object.

In left inlet: Sets the value to be written into the **buffer~** athe specified sample index. If the sample index is not -1, the value is written.

In middle inlet: Converted to int.

In right inlet: Converted to int.

int — In left inlet: Converted to float.

In middle inlet: Sets the sample index for writing subsequent sample values coming in the left inlet. If there is a signal connected to this inlet, a float is ignored.

In right inlet: Sets the channel of the **buffer~** where sample values are written. The first (left) channel is specified as 1.

list — In left inlet: A list of two or more values will write the first value at the sample index specified by the second value. If a third value is present, it specifies the audio channel within the **buffer~** for writing the sample value.

set — The word set, followed by the name of a **buffer~**, changes the **buffer~** where **poke~** will write its incoming samples.

(mouse) — Double-clicking on **poke~** opens an editing window where you can view the contents of its associated **buffer~** object.

Arguments

symbol — Obligatory. Names the **buffer~** where **poke~** will write its incoming samples.

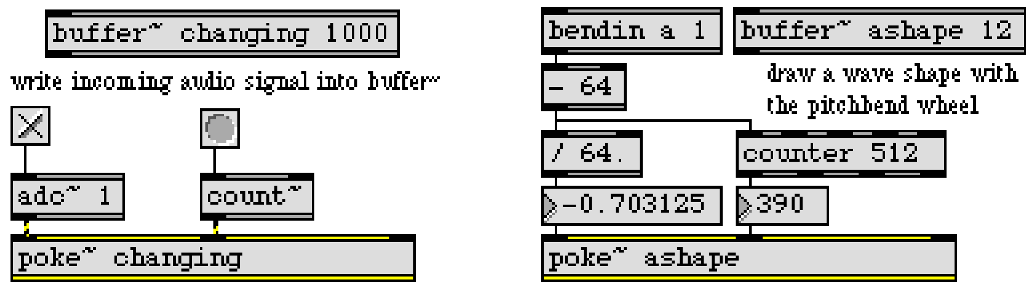
int — Optional. Sets the channel number of a multichannel **buffer~** where the samples will be written. The default channel is 1.

Output

None.

Write sample values by index

Examples



Write into a **buffer~** *using either signals or numbers*

See Also

buffer~	Store a sound sample
peek~	Read and write sample values

Input

pow~ raises the *base value* (set in the right inlet) to the power of the *exponent* (set in the left inlet). Either inlet can receive a signal, float or int.

signal — In left inlet: Sets the exponent.

In right inlet: Sets the base value.

float or int — In left inlet: Sets the exponent. If there is a signal connected to the left inlet, a number received in the left inlet is ignored.

In right inlet: Sets the base value. If there is a signal connected to the right inlet, a number received in the right inlet is ignored.

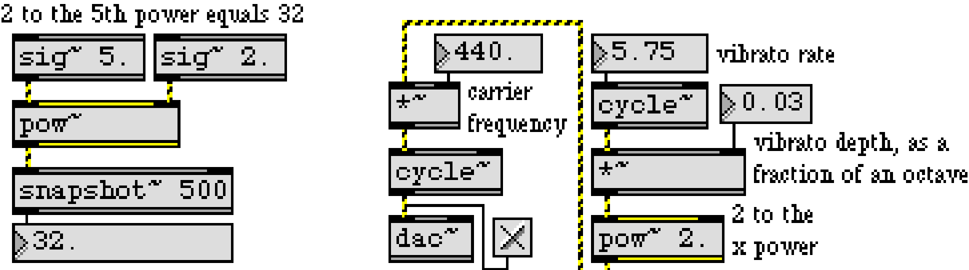
Arguments

float or int — Optional. Sets the base value. The default value is 0. If a signal is connected to the right inlet, the argument is ignored.

Output

signal — The base value (from the right inlet) raised to the exponent (from the left inlet).

Examples



Computes the mathematical expression x^y *for converting to logarithmic or exponential scale*

See Also

log~	Logarithm of a signal
curve~	Exponential ramp generator

Input

signal — The frequency at which a new random number between -1 and 1 is generated. **rand~** interpolates linearly between random values chosen at the specified rate.

float or int — Same as signal. If there is a signal connected to the inlet, a float or int is ignored.

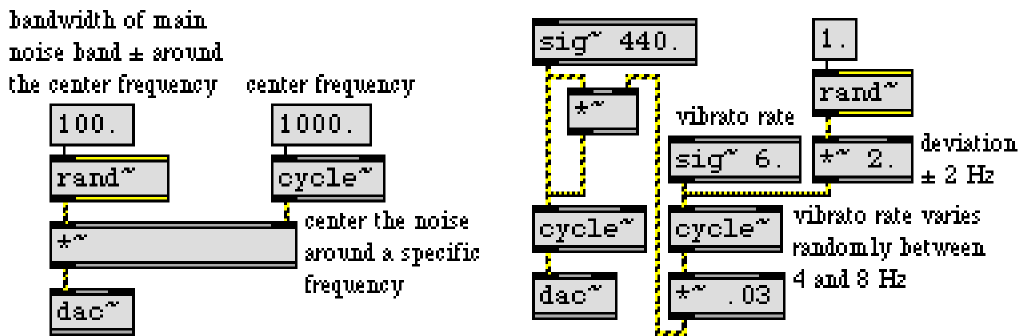
Arguments

float or int — Optional. Sets the initial frequency. The default value is 0. If a signal is connected to the inlet, the argument is ignored.

Output

signal — A signal consisting of line segments between random values in the range -1 to 1. The random values occur at the frequency specified by the input.

Examples



Use **rand~** *to create roughly band-limited noise, or as a control signal to create random variation*

See Also

noise~ — White noise generator

Input

signal The **receive~** object receives signals from all **send~** objects that share its name. It adds them together and sends the sum out its outlet. If no **send~** objects share the current name, the output of **receive~** is 0. The **send~** objects need not be in the same patch as the corresponding **receive~**.

set The word set, followed by a symbol, changes the name of the **receive~** so that it connects to different **send~** objects that have the symbol as a name. If no **send~** objects exist with the name, the output of **receive~** is 0.

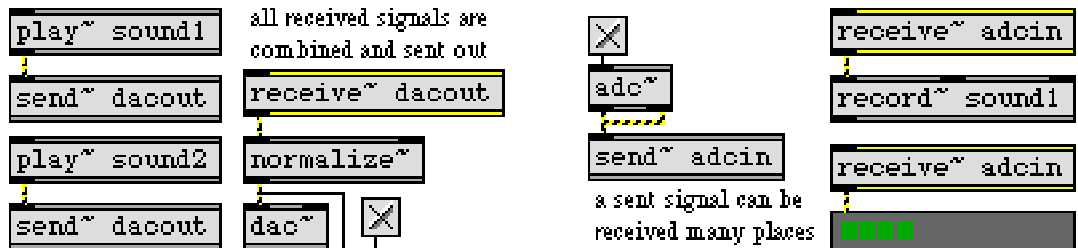
Arguments

symbol Obligatory. Sets the name of the **receive~** object.

Output

signal The combination of all signals coming into all **send~** objects with the same name as the **receive~**.

Examples



Signals can be received from any loaded Patcher, without patch cords

See Also

send~	Transmit signals without patch cords
Tutorial 4	Fundamentals: Routing signals

Input

signal — In left inlet: When recording is turned on, the signal is recorded into the sample memory of a **buffer~** at the current sampling rate.

In middle inlets: If **record~** has more than one input channel, these inlets record the additional channels into the **buffer~**.

int — In left inlet: Any non-zero number starts recording; 0 stops recording. Recording starts at the start point (see below) unless append mode is on.

int or float — In the inlet to the left of the right inlet: Set the start point within the **buffer~** (in milliseconds) for the recording. By default, the start point is 0 (the beginning of the **buffer~**).

In right inlet: Sets the end point of the recording. By default, the end point is the end of the **buffer~**'s allocated memory.

append — The word append, followed by a non-zero number, enables append mode. In this mode, when recording is turned on, it continues from where it was last stopped. append 0 disables append mode. In this case, recording always starts at the start point when it is turned on. Append mode is off initially by default.

loop — The word loop, followed by a non-zero number, enables loop recording mode. In loop mode, when recording reaches the end point of the recording (see above) it continues at the start point. loop 0 disables loop recording mode. In this case, recording stops when it reaches the end point. Loop mode is off initially by default.

set — The word set, followed by the name of a **buffer~**, changes the **buffer~** where **record~** will write the recorded samples.

(mouse) — Double-clicking on **record~** opens an editing window where you can view the contents of its associated **buffer~** object.

Arguments

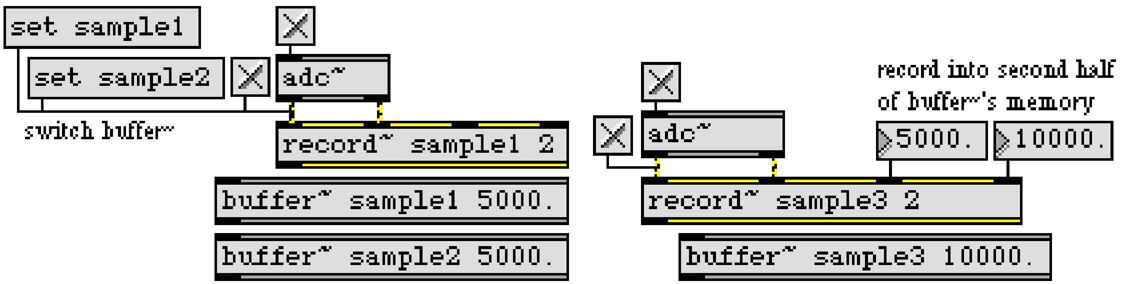
symbol — Obligatory. Names the **buffer~** where **record~** will write the recorded samples.

int — Optional, following the **buffer~** name argument. Specifies the number of input channels (1, 2, or 4). This determines the number of inlets **record~** has. The two rightmost inlets always set the record start and end points.

Output

signal — Sync output. During recording, this outlet outputs a signal that goes from 0 when recording at the start point to 1 when recording reaches the end point. When not recording, a zero signal is output.

Examples



Store a signal excerpt for future use

See Also

buffer~	Store a sound sample
groove~	Variable-rate looping sample playback
play~	Position-based sample playback
Tutorial 13	Sampling: Recording and playback

reson~

Input

signal — In left inlet: Any signal to be filtered.

In left-middle inlet: Sets the bandpass filter gain. This value should generally be less than 1.

In right-middle inlet: Sets the bandpass filter center frequency in hertz.

In right inlet: Sets the bandpass filter "Q"—roughly, the sharpness of the filter—where Q is defined as the filter bandwidth divided by the center frequency. Useful Q values are typically between 0.01 and 500.

int or float — An int or float can be sent in the three right inlets to change the filter gain, center frequency, and Q. If a signal is connected one of the inlets, a number received in that inlet is ignored.

list — The first number sets the filter gain. The second number sets the filter center frequency. The third number sets the filter Q. If any of the inlets corresponding to these parameters have signals connected, the corresponding value in the list is ignored.

clear — Clears the filter's memory. Since **reson~** is a recursive filter, this message may be necessary to recover from blowups.

Arguments

int or float — Optional. Numbers set the initial gain, center frequency, and Q. The default values are 0 for gain, 0 for center frequency, and 0.01 for Q.

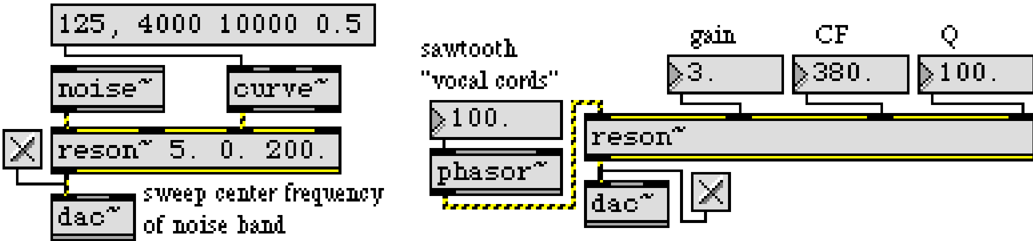
Output

signal — The filtered input signal. The equation of the filter is

$$y_n = gain * (x_n - r * x_{n-2}) + c1 * y_{n-1} + c2 * y_{n-2}$$

where r, $c1$, and $c2$ are parameters calculated from the center frequency and Q.

Examples



Control gain, center frequency, and Q of a bandpass filter to alter a rich signal

See Also

biquad~	Two-pole, two-zero filter
comb~	Comb filter

Sample and hold

Input

signal In left inlet: A signal to be sampled. When the control signal (in the right inlet) goes from being at or below the current trigger value to being above the trigger value, the signal in the left inlet is sampled and its value is sent out as a constant signal value.

In right inlet: The control signal. In order to cause a change in the output of **sah~**, the control signal must go from being at or below the trigger value to above the trigger value. When this transition occurs the signal in the left inlet is sampled and becomes the new output signal value.

int or float In left inlet: Sets the trigger value.

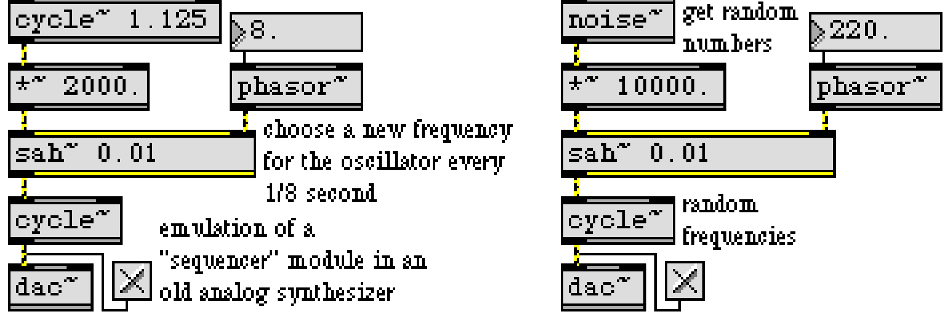
Arguments

int or float Optional. Sets the initial trigger value. The default is 0.

Output

signal When the control signal received in the right inlet goes from being at or below the trigger value to being above the trigger value, the output signal changes to the current value of the signal received in the left inlet. This signal value is sent out until the next time the trigger value is exceeded by the control signal.

Examples



Hold the signal value constant until the next trigger

See Also

phasor~ Sawtooth wave generator

Input

float or int — A value representing a number of samples received in the inlet is converted to milliseconds at the current sampling rate and sent out the object's right outlet. The input may contain a fractional number of samples. For example, at 44.1 kHz sampling rate, 322.45 samples is 7.31 milliseconds. (A float or int input triggers output even when audio is off.)

signal — Values in the signal represent a number of samples, and are converted to milliseconds at the current sampling rate and output as a signal out the left outlet. The input may contain a fractional number of samples.

Arguments

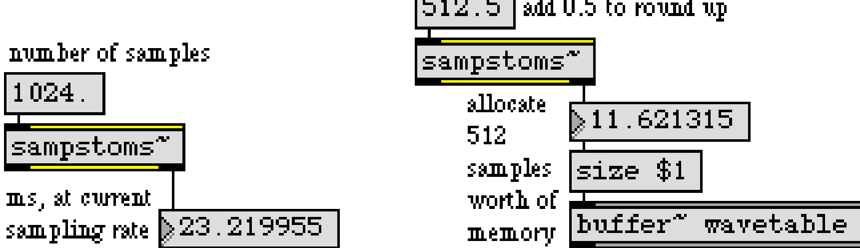
None.

Output

signal — Out left outlet: A signal consisting of the number of milliseconds corresponding to values representing a number of samples in the input signal.

float — Out right outlet: A number of milliseconds corresponding to a number of samples received in the inlet.

Examples



Some objects refer to time in samples, some in milliseconds

See Also

dspstate~	Report current DSP settings
mstosamps~	Convert milliseconds to samples

scope~

Input

signal — In left inlet: The input signal is displayed on the X axis of the oscilloscope.

In right inlet: The input signal is displayed on the Y axis of the oscilloscope.

If signals are connected to both the left and right inlets, **scope~** operates in X-Y mode, plotting points whose horizontal position corresponds to the value of the signal coming into the left (X) inlet and whose vertical position corresponds to the value of the signal coming into the right (Y) inlet. If the two signals are identical and in phase, a straight line increasing from left to right will be seen. If the two signals are identical and 180 degrees out of phase, a straight line decreasing from left to right will be seen. Other combinations may produce circles, ellipses, and Lissajous figures.

int — In left inlet: Sets the number of samples collected for each value in the display buffer. Smaller numbers expand the image but make it scroll by on the screen faster. The minimum value is 2, the maximum is 8092, and the default initial value is 256. In X or Y mode, the most maximum or minimum value seen within this period is used. In X-Y mode, a representative sample from this period is used.

In right inlet: Sets the size of the display buffer. This controls the rate at which **scope~** redisplays new information as well as the scaling of that information. If the buffer size is larger, the signal image will stay on the screen longer and be visually compressed. If the buffer size is smaller, the signal image will stay on the screen a shorter time before it is refreshed and will be visually expanded.

It might appear that the samples per display buffer element and the display buffer size controls do the same thing but they have subtly different effects. You may need to experiment with both controls to find the optimum display parameters for your application.

range — The word range, followed by two numbers (float or int) sets the minimum and maximum displayed signal amplitudes. The default values are -1 to 1.

delay — The word delay, followed by a number, sets the number of milliseconds of delay before **scope~** begins collecting values. After a non-zero delay period, **scope~** enters a state in which it may wait for a trigger condition to be satisfied in the input signal based on the setting of the trigger state (set with the trigger message) and trigger level (set with the triglevel message). By default, the delay is 0.

trigger — Sets the trigger mode. After a non-zero delay period (set with the delay message), **scope~** begins to wait for a particular feature in the input signal before it begins collecting samples. trigger 1 sets an upward trigger in which the signal must go from being below the trigger level (default 0) to being equal to it or above it. trigger 2 sets a downward trigger in which the signal must go from being above the trigger level to being equal to it or below it. The default trigger mode is 0, which does not wait after

a non-zero delay period before collecting samples again. This is sometimes referred to as a “line” trigger mode.

triglevel — The word triglevel, followed by a number, sets the trigger level, used by trigger modes 1 and 2. The default trigger level is 0. If you are displaying a waveform, making slight changes to the trigger level will move the waveform to the left or right inside the **scope~**. It is possible to set the trigger level so that **scope~** will never change the display.

(mouse) — When you click on a **scope~**, its display freezes for as long as you hold the mouse button down.

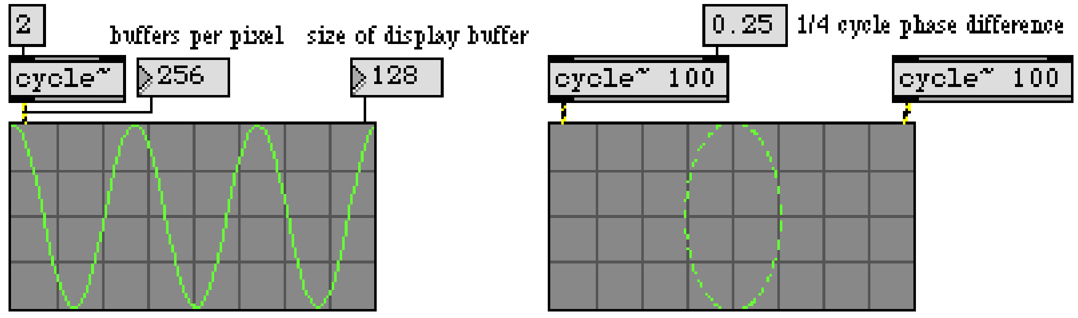
Arguments

None.

Output

None.

Examples



Display a signal, or plot two signals in X-Y mode

See Also

meter~	Visual peak level indicator
Tutorial 23	Analysis: Oscilloscope

selector~ *Assign one of several inputs to an outlet*

Input

int or float — In left inlet: If a signal is not connected to the left inlet, an int or float determines which input signal in the other inlets will be passed through to the outlet. If the value is 0 or negative, all inputs are shut off and a zero signal is sent out. If it is 1 but less than 2, the signal coming in the first inlet to the right of the leftmost inlet is passed to the outlet. If the number is 2 but less than 3, the signal coming into the next inlet to the right is used, and so on.

signal — In left inlet: If a signal is connected to the left inlet, **selector~** operates in a mode that uses signal values to determine which of its input signals is to be passed to its outlet. If the signal coming in the left inlet is 0 or negative, the output is shut off and a zero signal is sent out. If it is 1 but less than 2, the signal coming in the first inlet to the right of the leftmost inlet is passed to the outlet. If the signal is 2 but less than 3, the signal coming into the next inlet to the right is used, and so on.

In other inlets: Any signal, to be passed through to the **selector~** object's outlet depending on the value of the most recently received int or float in the left inlet, or the signal coming into the left inlet. The first signal inlet to the right of the leftmost inlet is considered input 1, the next to the right input 2, and so on.

If the signal network connected to one or more of the **selector~** signal inlets contains a **begin~** object, and a signal is not connected to the left inlet of the **selector~**, all processing between the **begin~** outlet and the **selector~** inlet is turned off when the input signal is not being passed to the **selector~** outlet.

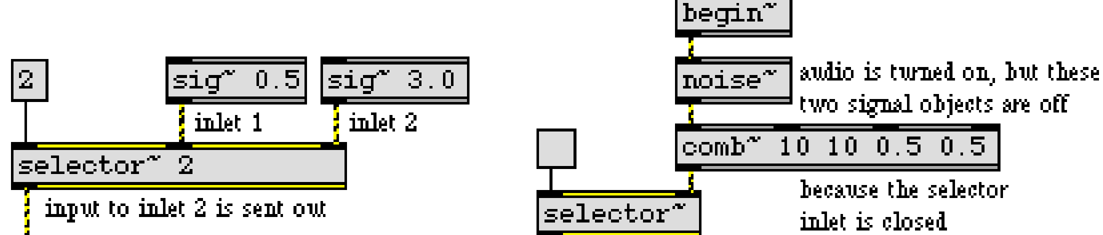
Arguments

int — Optional. The first argument specifies the number of input signals. The default is 1. The second argument specifies which signal inlet is initially open for its input to be passed through to the outlet. The default is 0, where all signals are shut off and a zero signal is sent out. If a signal is connected to the left inlet, the second argument is ignored.

Output

signal — The output is the signal coming in the "open" inlet, as specified by a number or signal in the left inlet. The output is a zero signal if all signal inlets are shut off.

Examples



Allow only one of several signals to pass; optionally turn off unneeded signal objects

See Also

gate~	Route a signal to one of several outlets
begin~	Define a switchable part of a signal network
Tutorial 5	Fundamentals: Turning signals on & off

Input

signal — The **send~** object sends its input signal to all **receive~** objects that share its name. The **send~** object need not be in the same patch as the corresponding **receive~** object(s).

set — The word set, followed by a symbol, changes the name of the **send~** so that it connects to different **receive~** objects that have the symbol as a name. (If no **receive~** objects with the same name exist, **send~** does nothing.)

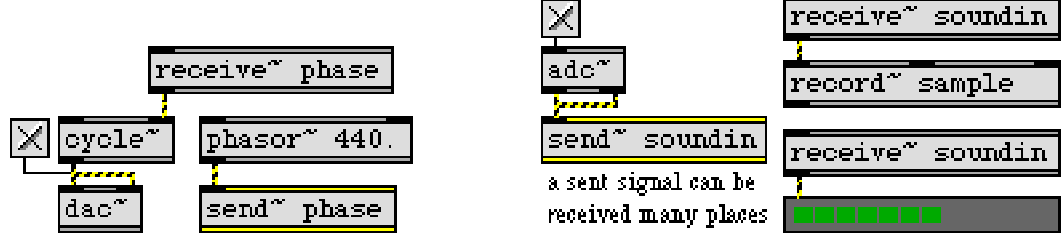
Arguments

symbol — Obligatory. Sets the name of the **send~** object.

Output

None.

Examples



Signal coming into **send~** *comes out any* **receive~** *object with the same name*

See Also

receive~	Receive signals without patch cords
Tutorial 4	Fundamentals: Routing signals

Input

open The word open, followed by a name of an AIFF or Sound Designer II file, opens the file if it exists in Max's search path. Without a filename, open brings up a standard open file dialog allowing you to choose a file. After the file is opened, **sfinfo~** interrogates the file and reports the number of channels, sample size, sample rate, and length in milliseconds out its outlets.

bang If a file has already been opened, either with the open message or specified by an argument to **sfinfo~**, bang reports the number of channels, sample size, sample rate, and length in milliseconds out the **sfinfo~** object's outlets.

Arguments

symbol Optional. Names a file that **sfinfo~** will report about when it receives a subsequent bang message. The file must exist in the Max search path.

Output

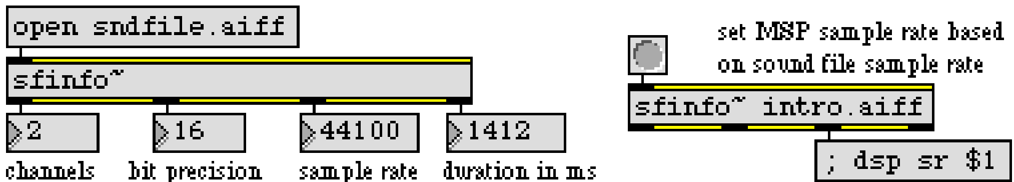
int Out left outlet: The number of channels in the sound file.

Out 2nd outlet: The sound file's sample size in bits (typically 16).

float Out 3rd outlet: The sound file's sampling rate.

Out right outlet: The duration of the sound file in milliseconds.

Examples



Report information about a specific sound file

See Also

info~ Report information about a sample
sflist~ Store sound file cues
sfplay~ Play sound file from disk

Input

open — The word open, followed by a name of an AIFF or Sound Designer II file, opens the file if it exists in Max's search path. Without a filename, open brings up a standard open file dialog allowing you to choose a file. When a file is opened, its beginning is read into memory, and until another file is opened, playing from the beginning the file is defined as cue 1. Subsequent cues can be defined referring to this file using the preload message without a filename argument. When the open message is received, the previous current file, if any, remains open and can be referred to by name when defining a cue with the preload message. If any cues were defined that used the previous current file, they are still valid even if the file is no longer current.

preload — Defines a cue—an integer greater than or equal to 2—to refer to a specific region of a file. When that cue number is subsequently received by an **sfplay~** object that is set to use cues from the **sflist~** object, the specified region of the file is played by **sfplay~**. Cue number 1 is always the beginning of the current file—the file last opened with the open message.—and cannot be modified with the preload message.

There are a number of forms for the preload message. The word preload is followed by an obligatory cue number between 2 and 32767. If the cue number is followed by a filename—a file that is currently open or one that is in Max's search path—that cue number will henceforth play the specified file . Note that a file need not have been explicitly opened with the open message in order to be used in a cue. If no filename is specified, the currently open file is used.

After the optional filename, an optional start time in milliseconds can be specified. If no start time is specified, the beginning of the file is used as the cue start point. After the start time, an end time in milliseconds can be specified. If no end time is specified, or the end time is 0, the cue will play to the end of the file. If the end time is less than the start time, the cue is defined but will not play. Eventually it may be possible to define cues that play in reverse.

Each cue that is defined requires 128K of memory per audio file channel at the default buffer size. This is the preloaded data for the start of the cue.

print — Prints a list of all the currently defined cues.

clear — The word clearwith no arguments clears all defined cues. After a clearmessage is received, only the number 1 will play anything (assuming there's an open file). The word clearfollowed by one or more cue numbers removes them from the **sflist~** object's cue list.

fclose — The word fclose, followed by the name of an open file, closes the file and removes all cues associated with it. The word fclose by itself closes the current file.

embed — The message embed, followed by any non-zero integer, causes **sflist~** to save all of its defined cues and the name of the current open file when the Patcher file is saved.

The message embed 0 keeps **sflist~** from saving this information when the Patcher is saved. By default, the current file name and the cue information is not saved in **sflist~** when the Patcher is saved. If an **sflist~** object is saved with stored cues, they will all be preloaded when the Patcher containing the object is loaded.

Arguments

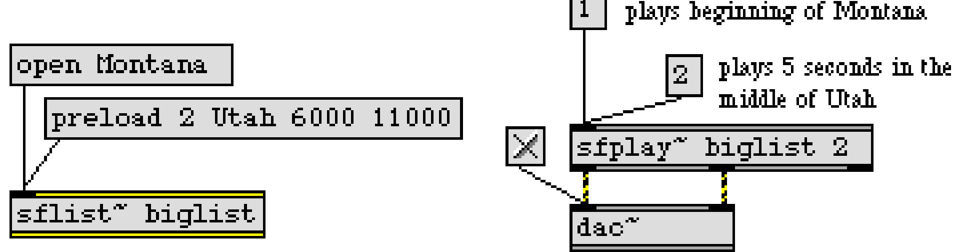
symbol Obligatory. Names the **sflist~**. **sfplay~** objects use this name to refer to cues stored inside the object.

int Optional. Sets the buffer size used to preload sound files. The default and minimum is 16384. Preloaded buffers are 4 times the buffer size per channel of the sound file.

Output

None.

Examples



Store a global list of cues that can be used by one or more **sfplay~** *objects.*

See Also

buffer~	Store a sound sample
groove~	Variable-rate looping sample playback
play~	Position-based sample playback
sfinfo~	Report sound file information
sfplay~	Play sound file from disk
sfrecord~	Record to sound file on disk
Tutorial 16	Sampling: Record and play sound files

Input

open Opens an AIFF or Sound Designer II file for playback and makes it the current file. The word open, followed by a filename, opens the file if it exists in Max's search path. Without a filename, open brings up a standard open file dialog allowing you to choose a file. When a file is opened, its beginning is read into memory, and until another file is opened, you can play the file from the beginning by sending **sfplay~** the message 1. When the open message is received, the previous current file, if any, remains open and can be referred to by name when defining a cue with the preload message. If any cues were defined that used the previous current file, they are still valid even if the file is no longer current.

int If a file has been opened with the open message, 1 begins playback (of the most recently opened file), and 0 stops playback. Numbers greater than 1 trigger cues that have been defined with the preload message, or that were defined based on the saved state of the **sfplay~** object. When the file is played, the audio data in the file is sent out the signal outlets according to the number of channels the object has. When the cue is completed or **sfplay~** is stopped with a 0, a bang is sent out the right outlet.

If the object is currently assigned to an **sflist~** object (using the set message or with a typed-in argument), an int will trigger cues stored in the **sflist~** object rather than inside the **sfplay~**. To reset **sfplay~** to use its own cues, send it the set message with no arguments.

set The message set, followed by a name of an **sflist~** object, will cause **sfplay~** to play cues stored in the **sflist~** when it receives an int or list. The message set with no arguments resets **sfplay~** to use its own internally defined cues when receiving an int or list.

list Gives a set of cues for **sfplay~** to play, one after the other. The maximum number of cues is in a list is 128. If a cue number in a list has not been defined, it is skipped and the next cue, if any, is tried. If the object is currently assigned to an **sflist~** object, a list uses cues stored in the **sflist~** object. Otherwise, cues stored inside the **sfplay~** object are used.

anything If the name of an **sflist~** object is sent to **sfplay~**, followed by a number, the numbered cue from the **sflist~** is played if it exists.

pause Pauses soundfile playback. You can continue playback at the paused point with the resume message, or at a different location if you send a cue number or the seek message.

resume If playback was paused, playback resumes from the paused point in the file.

preload Defines a cue—an integer greater than or equal to 2—to refer to a specific region of a file. When that cue number is subsequently received, **sfplay~** plays that region of

that file. Cue number 1 is always the beginning of the current file—the file last opened with the open message.—and cannot be modified with the preload message.

There are a number of forms for the preload message. The word preload is followed by an obligatory cue number between 2 and 32767. If the cue number is followed by a filename—a file that is currently open or one that is in Max's search path—that cue number will henceforth play the specified file . Note that a file need not have been explicitly opened with the open message in order to be used in a cue. If no filename is specified, the currently open file is used.

After the optional filename, an optional start time in milliseconds can be specified. If no start time is specified, the beginning of the file is used as the cue start point. After the start time, an end time in milliseconds can be specified. If no end time is specified, or the end time is 0, the cue will play to the end of the file. If the end time is less than the start time, the cue is defined but will not play. Eventually it may be possible to define cues that play in reverse.

Each cue that is defined requires 128K of memory per sound file channel at the default buffer size. This is the preloaded data for the start of the cue.

print — Prints information about the state of the object, plus a list of all the currently defined cues.

clear — The word clearwith no arguments clears all defined cues. After a clearmessage is received, only the number 1 will play anything (assuming there's an open file). The word clearfollowed by one or more cue numbers removes them from the **sfplay~** object's cue list.

fclose — The word fclose, followed by the name of an open file, closes the file and removes all cues associated with it. The word fclose by itself closes the current file.

seek — The word seek, followed by a start time in milliseconds, moves to the specified position in the current file and begins playing. After the start time, an optional end time can be specified, which will set a point for playback to stop. The seek message is intended to allow you to preview and adjust the start and end points of a cue.

embed — The message embed, followed by any non-zero integer, causes **sfplay~** to save all of its defined cues and the name of the current open file when the Patcher file is saved. The message embed 0 keeps **sfplay~** from saving this information when the Patcher is saved. By default, the current file name and the cue information is not saved in **sfplay~** when the Patcher is saved. If an **sfplay~** object is saved with stored cues, they will all be preloaded when the Patcher containing the object is loaded.

Play sound file from disk

Arguments

symbol — Optional. If the first argument is a symbol, it names an **sflist~** that the **sfplay~** object will use for playing cues. If no symbol argument is given, **sfplay~** plays its own internally defined cues.

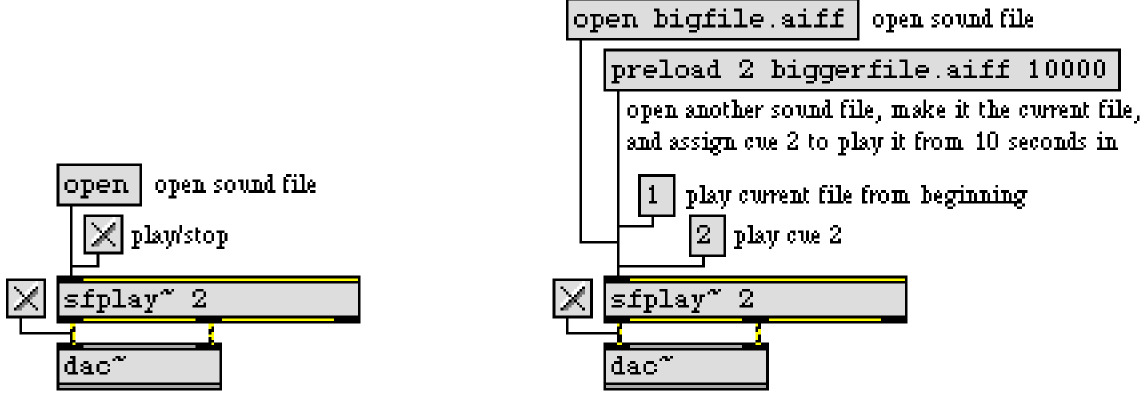
int — Optional. Sets the number of output channels, which determines the number of outlets that the **sfplay~** object will have. The maximum number of channels is 8. The default is 1. If the sound file being played has more output channels than the **sfplay~** object, higher-numbered channels will not be played. If the sound file has fewer channels, the signals coming from the extra outlets of **sfplay~** will be 0.

Output

signal — Each outlet except the right outlet sends out the audio data of the corresponding channel of the sound file when a cue number is received in the inlet. (The left outlet plays channel 1, and so on.)

bang — Out right outlet: When the file is done playing, or when playback is stopped with a 0 message, a bang is sent out.

Examples



Sound files can be played from the hard disk, without loading the whole file into memory

See Also

buffer~	Store a sound sample
groove~	Variable-rate looping sample playback
play~	Position-based sample playback
sfinfo~	Report sound file information
sflist~	Store sound file cues
sfrecord~	Record to sound file on disk
Tutorial 16	Sampling: Record and play sound files

Input

open — In left inlet: Opens an AIFF file for recording. The word open, followed by a symbol filename, creates (or opens, if it already exists) the specified file in the current default volume. Without a symbol, open brings up a standard save file dialog allowing you to name a file for recording. An existing file with the same name will be overwritten.

opensd2 — Opens a Sound Designer II file for recording. The word opensd2, followed by a symbol filename, creates (or opens, if it already exists) the specified file in the current default volume. Without a symbol, opensd2 brings up a standard save file dialog allowing you to name a file for recording. An existing file with the same name will be overwritten.

int — In left inlet: If a file has been opened with the open or opensd2 message, a non-zero value begins recording, and 0 stops recording and closes the file. **sfrecord~** requires another open or opensd2 message to record again if a 0 has been sent.

Recording may also stop spontaneously if there is an error, such as running out of space on your hard disk.

signal — Each inlet of **sfrecord~** accepts a signal which is recorded to a channel of a sound file when recording is turned on.

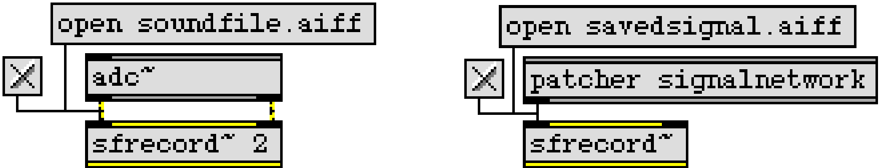
print — Outputs cryptic status information about the progress of the recording.

record — In left inlet: If a file has been opened with the open or opensd2 message, the word record, followed by a time in milliseconds, begins recording for the specified amount of time. The recording can be stopped before it reaches the end by sending **sfrecord~** a 0 in its left init.

Arguments

int — Optional. Sets the number of input channels, which determines the number of inlets that the **sfrecord~** object will have. The maximum number of channels is 8, and the default is 1. The AIFF or Sound Designer II file created will have the same number of channels as this argument. Whether you can actually record the maximum number of channels is dependent on the speed of your processor and hard disk.

Examples



Save a sound file containing "real world" sound and/or sound created in MSP

See Also

sfplay~	Play sound file from disk
Tutorial 16	Sampling: Record and play sound files

Input

int or float The number is sent out as a constant signal.

signal Any signal input is ignored. You can connect a **begin~** object to the **sig~** inlet to define the beginning of a switchable signal network.

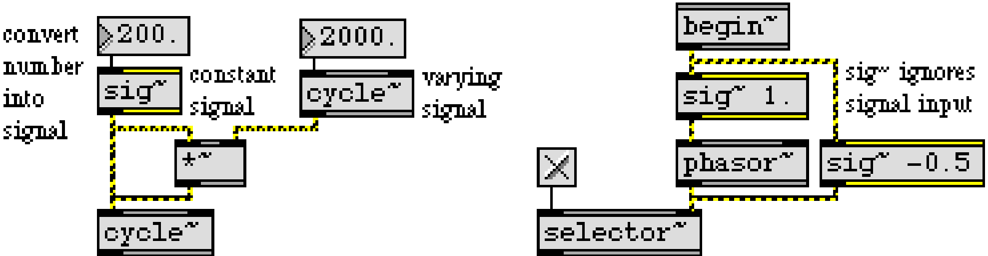
Arguments

int or float Optional. Sets an initial signal output value.

Output

signal **sig~** outputs a constant signal consisting of the value of its argument or the most recently received int or float in its inlet.

Examples



Provide constant numerical values to a signal network with **sig~**

See Also

+~ Add signals
begin~ Define a switchable part of a signal network
line~ Ramp generator
Tutorial 4 Fundamentals: Routing signals

snapshot~ *Convert signal values to numbers*

Input

signal In left inlet: The signal whose values will be sampled and sent out the outlet.

int or float In left inlet: Any non-zero number turns on the object's internal clock, 0 turns it off. The internal clock is on initially by default, if a positive clock interval has been provided.

In right inlet: Sets the interval in milliseconds for the internal clock that triggers the automatic output of values from the input signal. If the interval is 0, the clock stops. If it is a positive integer, the interval changes the rate of data output.

bang Sends out a report of the current signal value.

offset The word offset, followed by a number, sets the number of the sample within a signal vector that will be reported when **snapshot~** sends its output. The number is constrained between 0 (the default) and the current signal vector size minus one.

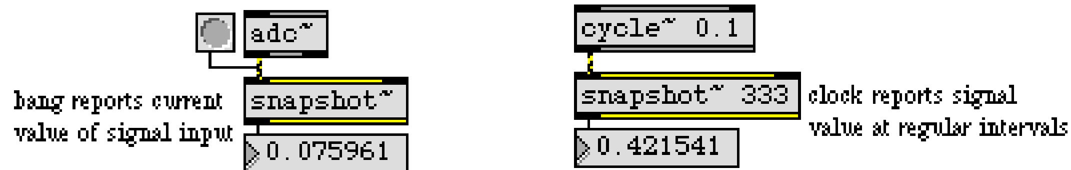
Arguments

int Optional. The first argument sets the internal clock interval. If it is 0, the internal clock is not used, so **snapshot~** will only output data when it receives a bang message. By default, the interval is 0. The second argument sets the sample number within a signal vector that is reported.

Output

float When **snapshot~** receives a bang, or its internal clock is on, sample values from the input signal are sent out its outlet.

Examples



See a sample of a signal at a given moment

See Also

capture~ Store a signal to view as text
sig~ Constant signal of a number
Tutorial 22 Analysis: Viewing signal data

sqrt~

Input

signal **sqrt~** outputs a signal that is the square root of the input signal. A negative input has no real solution, so it causes an output of 0.

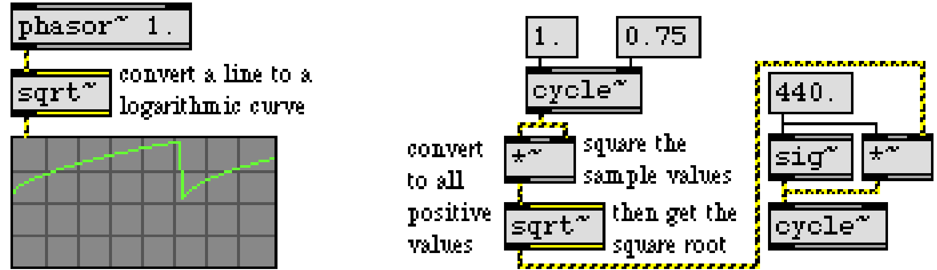
Arguments

None.

Output

signal The square root of the input signal.

Examples



Output signal is the square root of the input signal

See Also

curve~	Exponential ramp generator
log~	Logarithm of a signal
pow~	Signal power function

tapin~

Input

signal — The signal is written into a delay line that can be read by the **tapout~** object.

clear — Clears the memory of the delay line., which may produce a click in the output.

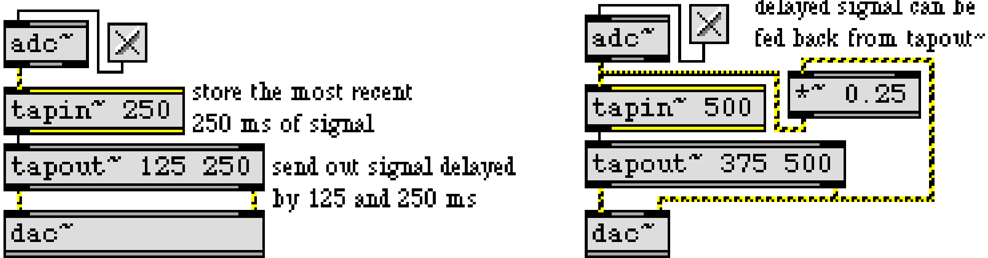
Arguments

float or int — Optional. The maximum delay time in milliseconds. This determines the size of the delay line memory. If the sampling rate is increased after the object has been created, **tapin~** will attempt to resize the delay line. If no argument is present, the default maximum delay time is 100 milliseconds.

Output

tap — In order for the delay line to function, the outlet of **tapin~** must be connected to the left inlet of **tapout~**. It cannot be connected to any other object.

Examples



tapin~ *creates a delay buffer from which to tap delayed signal*

See Also

delay~	Delay line specified in samples
tapout~	Output from a delay line
Tutorial 25	Processing: Delay lines

Input

tap — In left inlet: The outlet of a **tapin~** object must be connected to the left inlet of **tapout~** in order for the delay line to function.

The **tapout~** object has one or more inlets and one or more outlets. A delay time signal or number received in an inlet affects the output signal coming out of the outlet directly below the inlet.

signal — If a signal is connected to an inlet of **tapout~**, the signal coming out of the outlet below it will use a continuous delay algorithm. Incoming signal values represent the delay time in milliseconds. If the signal increases slowly enough, the pitch of the output will decrease, while if the signal decreases slowly, the pitch of the output will increase. The continuous delay algorithm is more computationally expensive than the fixed delay algorithm that is used when a signal is not connected to a **tapout~** inlet.

float or int — If a signal is not connected to an inlet of **tapout~**, a fixed delay algorithm is used, and a float or int received in the inlet sets the delay time of the signal coming out of the corresponding outlet. This may cause clicks to appear in the output when the delay time is changed. However, fixed delay is suitable for many applications such as reverberation where delay times do not change dynamically, and it is computationally less expensive than the continuous delay algorithm.

list — In left inlet: Allows several fixed delay times to be changed at the same time. The first number in the list sets the delay time for the first outlet, and so on. If any inlets corresponding to list values have signals connected to them, the values are skipped.

Arguments

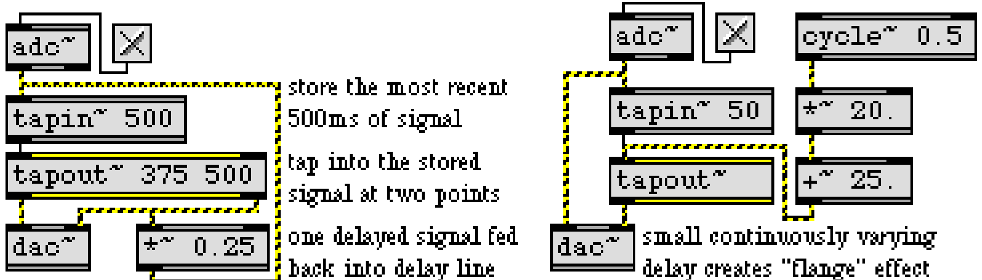
float or int — Optional. One or more initial delay times in milliseconds, one for each delay “tap” inlet-outlet pair desired. For example, the arguments 50 100 300 would create a **tapout~** object with three independent “taps” corresponding to three inlets and three outlets. If a signal is connected to an inlet, the initial delay time corresponding to that inlet-outlet pair is ignored.

Output

signal — Each outlet of **tapout~** corresponds to an individually controlled “tap” of a delay line written by the **tapin~** object. The output signal coming out of a **tapout~** outlet is the input to **tapin~** delayed by the number of milliseconds specified by the numerical or signal control received in the inlet directly above the outlet.

Output from a delay line

Examples



tapout~ *sends out the signal* **tapin~** *receives, delayed by some amount of time*

See Also

delay~	Delay line specified in samples
tapin~	Input to a delay line
Tutorial 25	Processing: Delay lines

Detect signal above a set level

Input

signal In left inlet: A signal whose level you want to detect.

float In middle inlet: Sets the lower ("reset") threshold level for the input signal. When a sample in the input signal is greater than or equal to the upper ("set") level, **thresh~** sends out a signal of 1 until a sample in the input signal is less than or equal to this reset level.

In right inlet: Sets the upper ("set") threshold level for the input signal. When the input is equal to or greater than this value, **thresh~** sends out a signal of 1.

Arguments

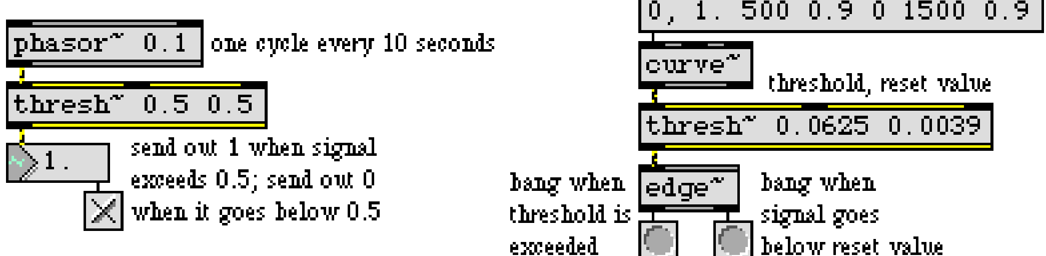
float The first argument specifies the *reset* or low threshold level. If no argument is present, the reset level is 0. The second argument specifies the *set* or high threshold level. If no argument is present, the set level is 0.

If only one argument is present, it specifies the reset level, and the set level is 0.

Output

signal When a sample in the input signal is greater than or equal to the upper threshold level, the output is 1. The output continues to be 1 until a sample in the input signal is equal to or less than the reset level. If the set level and the reset level are the same, the output is 1 until a sample in the input signal is less than the reset level.

Examples



Detect when signal exceeds a certain level

See Also

>~	*Is greater than*, comparison of two signals
change~	Report signal direction
edge~	Detect logical signal transitions

train~ *Pulse train generator*

Input

signal — In left inlet: Specifies the period (time interval between pulse cycles), in milliseconds, of a pulse train sent out the left outlet.

In middle inlet: Controls the pulse width or duty cycle. The signal values represent a fraction of the pulse interval that will be devoted to the "on" part of the pulse (signal value of 1). A value of 0 has the smallest "on" pulse size (usually a single sample), while a value of 1 has the largest (usually the entire interval except a single sample). A value of .5 makes a pulse with half the time at 1 and half the time at 0.

In right inlet: Sets the phase of the onset of the "on" portion of the pulse. A value of 0 places the "on" portion at the beginning of the interval, while other values (up to 1, which is the same as 0) delay the "on" portion by a fraction of the total inter-pulse interval.

float or int — Numbers can be used instead of signals to control period, pulse width, and phase. If a signal is also connected to the inlet, floats and ints are ignored.

Arguments

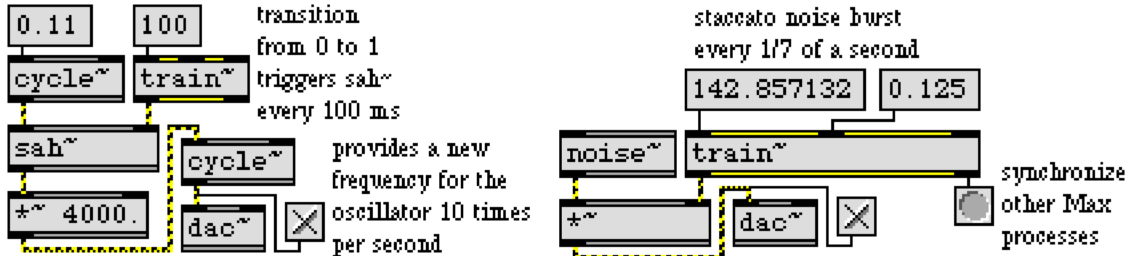
float or int — Optional. Initial values for inter-pulse interval in milliseconds (default 1000), pulse width (default 0.5), and phase (default 0). If signals are connected to any of the **train~** object's inlets, the corresponding initial argument value is ignored.

Output

signal — Out left outlet: A pulse (square) wave train having the specified interval, width, and phase.

bang — Out right outlet: When the "on" portion of the pulse begins, a bang is sent out the right outlet. Using this outlet, you can use **train~** as a signal-synchronized metronome with an interval specifiable as a floating-point (or signal) value. However, there is an unpredictable delay between the "on" portion of the pulse and the actual output of the bang message, which depends in part on the current Max scheduler interval. The delay is guaranteed to be a millisecond or less if the scheduler interval is set to 1 millisecond.

Examples



Provide an accurate pulse for rhythmic changes in signal

See Also

<~	*Is less than*, comparison of two signals
>~	*Is greater than*, comparison of two signals
clip~	Limit signal amplitude
phasor~	Sawtooth (phase) wave generator

Input

signal — In left inlet: The left channel audio input to the plug-in. If the plug-in has a mono input, use the **vst~** object's left inlet.

In right inlet: The right channel audio input to the plug-in.

int — In left inlet: Changes the current program of the plug-in. A program is a collection of settings for all of the plug-in's parameters. This message will cause them all to change at once.

list — In left inlet: A list consisting of an int followed by a float between 0 and 1 changes the parameter specified by the first number to the value of the second number.

anything — In left inlet: A symbol specifying a parameter name defined by the object (see the params message) followed by a float between 0 and 1 sets the named parameter to the specified value.

open — In left inlet: Opens the plug-in's editing window. If the plug-in does not contain its own editor, a default editing window is displayed. The default window allows the plug-ins parameters to be edited with horizontal sliders, as well as banks of programs to be loaded and saved. These functions are typically available in a different form in plug-in specific editor windows.

read — In left inlet: Opens a dialog box for choosing a file containing a bank of programs for the plug-in. It is possible to open a VST program file that doesn't contain programs for a particular plug-in; in this case, the read operation will do nothing.

write — In left inlet: Opens a dialog box allowing you to name a file where you want to save the current set of programs for the plug-in, and writes out a file.

mix — In left inlet: The word mix, followed by a 1, turns on mixing of the input with the plug-in's output. The word mix, followed by a value of 0, turns off mixing of the input with the plug-in's output. By default, input mixing is off, although the mix argument, if present, will turn it on initially.

set — In left inlet: The word set, followed by a symbol, renames the current program to the name specified by the symbol.

params — In left inlet: Outputs a series of symbols containing the names of all the plug-in's parameters.

plug — In left inlet: The word plug, followed by a symbol filename, looks for a VST plug-in file by that name. If one is found, the currently loaded plug-in is closed and the one specified by the filename is opened. If no filename is specified, the plug message will open a dialog box asking for the name of a VST plug-in to load .

(mouse) — Double-clicking on **vst~** opens the plug-in's editing window.

Arguments

symbol Obligatory. The name of the VST plug-in file or none if no file is to be loaded. If the file cannot be found, the default plug-in is loaded, which acts as a simple stereo gain control.

Optional. Following the plug-in name, you can specify the name of a VST program file to load for the plug-in.

Optional. Following the plug-in name or program filename, the word mix turns on mixing the input with the plug-in's output.

Output

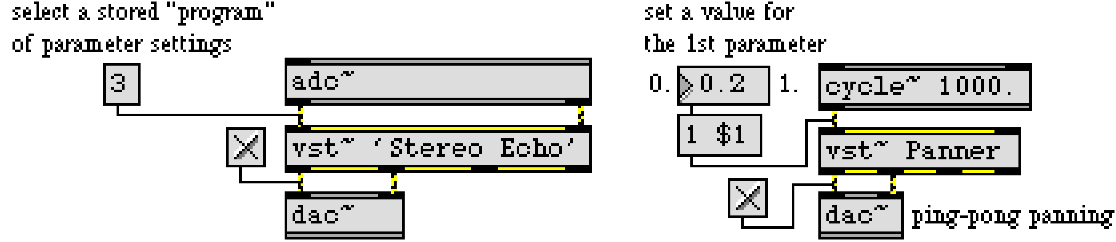
signal Out left outlet: The left channel of the audio output produced by the plug-in.

Out 2nd outlet: The right channel of the audio output produced by the plug-in.

symbol Out 3rd outlet: When **vst~** receives the params message , it sends out a series of messages containing the parameter names of each parameter (starting at the first parameter, which is number 1) defined by the plug-in. Some plug-ins, especially those with their own editors, fail to name the parameters. In this case, the output of the params message may be relatively useless.

list Out right outlet: When a plug-in parameter is changed, either with a list as input or by using the plug-in editor window, a list containing the parameter number followed by the new parameter value is sent out this outlet.

Examples



Process an audio signal with a VST plug-in

See Also

gain~ Exponential scaling volume slider.

Input

signal — In left inlet: Input signal values progressing from 0 to 1 are used to scan a specified range of samples in a **buffer~** object. The output of a **phasor~** can be used to control **wave~** as an oscillator, treating the range of samples in the **buffer~** as a repeating waveform. However, note that when changing the frequency of a **phasor~** connected to the left inlet of **wave~**, the perceived pitch of the signal coming out of **wave~** may not correspond exactly to the frequency of **phasor~** itself if the stored waveform contains multiple or partial repetitions of a waveform. You can invert the **phasor~** to play the waveform backwards.

In middle inlet: The start of the waveform as a millisecond offset from the beginning of a **buffer~** object's sample memory.

In right inlet: The end of the waveform as a millisecond offset from the beginning of a **buffer~** object's sample memory.

float or int — In middle or right inlets: Numbers can be used instead of signals to control the start and end points of the waveform, provided a signal is not connected to the inlet that receives the number.

set — The word set, followed by a symbol, sets the **buffer~** used by **wave~** for its stored waveform. The symbol can optionally be followed by two values setting new waveform start and end points. If the values are not present, the default start and end points (the start and end of the sample) are used. If signals are connected to the start and/or end point inlets, the start and/or end point values are ignored.

Arguments

symbol — Obligatory. Names the **buffer~** object whose sample memory is used by **wave~** for its stored waveform. Note that if the underlying data in a **buffer~** changes, the signal output of **wave~** will change, since it does not copy the sample data in a **buffer~**. **wave~** always uses the first channel of a multi-channel **buffer~**.

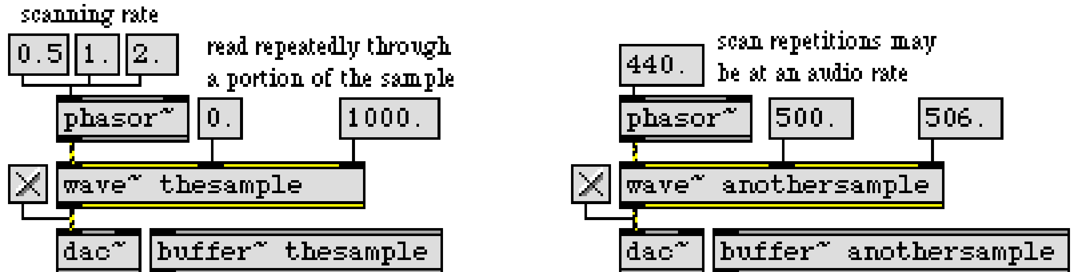
float or int — Optional. After the **buffer~** name argument, you can type in values for the start and end points of the waveform, as millisecond offsets from the beginning of a **buffer~** object's sample memory. By default the start point is 0 and the end point is the end of the sample. If you want to set a non-zero start point but retain the sample end as the waveform end point, use only a single typed-in argument after the **buffer~** name. If a signal is connected to the start point (middle) inlet, the initial waveform start point argument is ignored. If a signal is connected to the end point (right) inlet, the initial waveform end point is ignored.

Output

signal — The portion of the **buffer~** specified by **wave~**'s start and end points is scanned by signal values ranging from 0 to 1 in **wave~**'s inlet, and the corresponding sample

value from the **buffer~** is sent out **wave~**'s outlet. If the signal received in wave's inlet is a repeating signal such as a sawtooth wave from a **phasor~**, the resulting output will be a waveform (excerpted from the **buffer~**) repeating at the frequency corresponding to the repetition of the input signal.

Examples



Loop through part of a sample, treating it as a variable-size wavetable

See Also

buffer~	Store a sound sample
groove~	Variable-rate looping sample playback
phasor~	Sawtooth (phase) wave generator
play~	Position-based sample playback
Tutorial 15	Sampling: Variable-length wavetable

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 in Max 3.5 (refer to the Max 3.5 manual for a list of messages understood by the max object). 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 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, Debugging on; N = 0, Debugging off
; dsp takeover N	N = 1, Scheduler in Audio Interrupt On; N = 0, Scheduler in Audio Interrupt Off

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.

Absolute value of all samples in a signal ..... abs~

Adding signals together ..... +~

Additive synthesis ..... +~, cycle~

AIFF saving and playing ..... buffer~, info~, sfplay~, sfrecord~

Aliasing ..... dspstate~

Amplification ..... *~, /~, gain~, normalize~

Amplitude indicator ..... avg~, meter~

Amplitude modulation ..... *~

Analog-to-digital converter ..... adc~, ezadc~

Analysis of a signal ..... capture~, fft~, scope~

Average signal amplitude ..... avg~

Backward sample playback ..... groove~, play~

Bandpass filter ..... noise~, rand~, reson~

Bypassing a signal ..... gate~, mute~, pass~, selector~

Chorusing ..... cycle~, tapout~

Clipping ..... clip~, dac~, normalize~

Comb filter ..... comb~

Comparing signals ..... <~, ==~, >~, change~, meter~, scope~, snapshot~

Constant signal value ..... sig~

Control function ..... curve~, function~, line~

Convert Max messages to signals ..... curve~, line~, peek~, poke~, sig~

Convert signals to Max messages ..... avg~, meter~, peek~, snapshot~

Cosine wave ..... cos~, cycle~

DC offset ..... +~, -~, number~, sig~

Delay ..... allpass~, comb~, delay~, tapin~, tapout~

Difference between samples ..... change~, delta~

Difference between signals ..... -~, scope~

Digital-to-analog converter ..... dac~, ezdac~

Disabling part of a signal network ..... gate~, mute~, pass~, selector~

Display signal value ..... capture~, meter~, number~, scope~, snapshot~

Downsampling ..... avg~, number~, sah~, snapshot~

Duty cycle of a pulse wave ..... <~, >~, train~

Editing an audio sample ..... record~, peek~, poke~

Envelope following ..... adc~, ezadc~, function~, line~

Envelope generator ..... curve~, function~, line~

Equalization ..... allpass~, biquad~, comb~, lores~, reson~

Exponential curve function ..... curve~, gain~, linedrive~, pow~

Feedback delayed signal ..... allpass~, biquad~, comb~, lores~, reson~, tapin~, tapout~

Filter ..... allpass~, biquad~, comb~, lores~, noise~, reson~, vst~

Flanging ..... cycle~, tapout~

Fourier analysis and synthesis ..... fft~, ifft~

Frequency modulation ..... +~, cycle~, phasor~

Frequency-to-pitch conversion ..... ftom

Function generator ..... curve~, function~, line~, peek~, poke~

Global signal values ..... receive~, send~

Hertz equivalent of a MIDI key number ..... ftom, mtof

IIR filter .... **allpass~**, **biquad~**, **comb**, **lores~**, **reson~**
Input received in audio input jack .... **adc~**, **ezadc~**
Inverting signals .... ***~**, **-~**
Level control .... ***~**, **/~**, **gain~**, **normalize~**
Level meter .... **meter~**, **number~**
Limiter .... **clip~**, **lookup~**
Logical operations using signal values .... **<~**, **==~**, **>~**, **edge~**
Logarithmic curve function .... **curve~**, **gain~**, **linedrive~**, **log~**, **pow~**, **sqrt~**
Lookup table .... **buffer~**, **cycle~**, **function~**, **index~**, **lookup~**, **peek~**, **wave~**
Loop points in a sound file .... **info~**
Looping a sample .... **groove~**, **info~**, **wave~**
Lowpass filter .... **lores~**, **noise~**, **rand~**
Max messages converted to signals .... **curve~**, **line~**, **peek~**, **poke~**, **sig~**
Max messages derived from signals .... **avg~**, **edge~**, **meter~**, **number~**, **peek~**, **snapshot~**
MIDI control from MSP .... **avg~**, **ftom**, **function~**, **number~**, **snapshot~**
MIDI control of MSP .... **curve~**, **line~**, **mtof**, **sig~**
Millisecond calculations .... **mstosamps~**, **sampstoms~**
Mixing signals .... **+~**
Multiplying signals .... ***~**
Noise .... **noise~**, **rand~**
Noise gate .... **gate~**
Normalization .... ***~**, **/~**, **normalize~**
Numerical display of a signal .... **capture~**, **number~**, **snapshot~**
On/off audio switch .... **adc~**, **dac~**, **dspstate~**, **ezadc~**, **ezdac~**
Oscillator .... **cycle~**, **phasor~**, **wave~**
Oscilloscope .... **scope~**
Output audio jack .... **dac~**, **ezdac~**
Peak amplitude .... **meter~**
Periodic waves .... **cycle~**, **phasor~**, **wave~**
Phase distortion synthesis .... **kink~**, **phasor~**
Phase modulation .... **phasor~**
Pitch bend .... **ftom**, **mtof**
Pitch-to-frequency conversion .... **mtof**
Playing audio .... **dac~**, **ezdac~**
Playing samples .... **buffer~**, **groove~**, **index~**, **play~**, **sfplay~**, **wave~**
Plug-in in VST format used in MSP .... **vst~**
Pulse wave .... **<~**, **>~**, **clip~**, **train~**
Ramp signal .... **curve~**, **line~**
Random signal values .... **noise~**, **rand~**
Recording audio samples .... **adc~**, **ezadc~**, **poke~**, **record~**, **sfrecord~**
Repetition at sub-audio rates .... **cycle~**, **phasor~**, **train~**
Resonant filter .... **allpass~**, **biquad~**, **comb~**, **lores~**, **reson~**
Reverberation .... **allpass~**, **comb~**, **tapin~**, **tapout~**
Reversed sample playback .... **groove~**, **play~**
Ring modulation .... ***~**
Sample and hold .... **sah~**

Sample index in a buffer .... **count~**, **index~**
Sample playback .... **buffer~**, **groove~**, **index~**, **play~**, **sfplay~**, **wave~**
Sample storage .... **buffer~**, **record~**, **sfrecord~**
Sampling rate .... **adc~**, **buffer~**, **count~**, **dac~**, **dspstate~**, **mstosamps~**, **sampstoms~**
Sawtooth oscillator .... **phasor~**
Sine wave .... **cos~**, **cycle~**
Sound Designer II saving and playing .... **buffer~**, **info~**, **sfplay~**, **sfrecord~**
Spectrum measurement .... **fft~**, **ifft~**
Start and end point of a sample .... **groove~**, **index~**, **play~**, **wave~**
Subpatch control .... **mute~**, **receive~**, **send~**
Subtractive synthesis .... **allpass~**, **biquad~**, **comb~**, **lores~**, **noise~**, **rand~**, **reson~**
Switching signal flow on and off .... **gate~**, **mute~**, **pass~**, **selector~**
Table lookup .... **buffer~**, **cycle~**, **function~**, **index~**, **lookup~**, **peek~**, **wave~**
Text file of signal samples .... **capture~**
Transfer function .... **cycle~**, **lookup~**
Triggering a Max message with an audio signal .... **edge~**, **thresh~**
Varispeed sample playback .... **groove~**, **play~**
Vector size .... **adc~**, **dac~**, **dspstate~**
Velocity (MIDI) control of a signal .... **curve~**, **gain~**, **line~**, **sig~**
View a signal .... **buffer~**, **capture~**, **number~**, **scope~**, **snapshot~**
Waveshaping .... **lookup~**
Wavetable synthesis .... **buffer~**, **cycle~**, **wave~**
White noise .... **noise~**
Windowing a portion of a signal .... **index~**, **cycle~**, **gate~**, **lookup~**, **wave~**

Index

*~ 34, 177
+~ 178
-~ 179
/~ 180
<~ 181
==~ 182
>~ 183
abs~ 184
absolute value 184
absorption of sound waves 144
access the hard disk 95
ADAT 172
adc~ 84, 185
adding signals together 41, 178
additive synthesis 21, 63
AIFF 38, 85, 95, 272, 275
aliasing 17, 52, 156
allpass~ 186
amplification 177, 221, 244
amplitude 9, 126
amplitude adjustment 34
amplitude envelope 13, 60, 64, 99, 234
amplitude modulation 67, 71, 130
analog-to-digital conversion 15, 84, 185, 211
AppleTalk 26
ASCII 97
AtodB subpatch 46
attack, velocity control of 110
audio input 164, 185, 211
audio input and output 159
audio interface card 167
audio output 164, 206, 212
audio processing off for some objects 53, 189, 223, 266
Audiodriver files 159, 168
audiodrivers folder 167
Audiomedia II 170
AV connector 165
avg~ 188
balance between stereo channels 120
band-limited noise 256
band-limited pulse 156
bandpass filter 260
beats 44, 133
begin~ 53, 189
bell-like tone 65
biquad~ 190
buffer~ 38, 85, 192
capture~ 130, 195
cards, audio interface 167
carrier oscillator 68
change~ 196
chorus 151
clip~ 197
clipping 20, 34
comb~ 154, 198
comb filter 142, 154, 198
comparing signal values 181, 182, 183, 283
complex tone 9, 63
composite instrument sound 41
control rate 24
convolution 67
cos~ 200
cosine wave 31, 200, 204
count~ 86, 201
critical band 69
crossfade 41
    constant intensity 122
    linear 122
    speaker-to-speaker 124
Csound 5
cue sample for playback 96
current file for playback 96
curve~ 202
cycle~ 31, 204
dac~ 206
DAE 170
dBtoA subpatch 105
DC offset 72, 178, 179, 245, 277
decibels 14, 46, 105, 221
default values 36
delay~ 207
delay 141, 207
delay line 207, 280, 281
delay line with feedback 144, 152, 154
delay time modulation 148
delta~ 208
difference frequency 44, 69, 133
Digidesign 165, 170
DigiSystem™ INIT 170
digital audio in and out 171

Index

digital-to-analog converter 16, 31, 206, 212
diminuendo 61
Direct I/O MSP Audiodriver 170
disable audio of a subpatch 55
disk, soundfiles on 95
display signal 264
display signal amplitude 239, 278
display signal as text 195
display signal graphically 132
display the value of a signal 126, 245
divide one signal by another 180
Doppler effect 147
dsp object 173, 175
DSP Status window 159, 169
dspstate~ 132, 209
duty cycle 284
echo 141
edge~ 210
envelope 39
envelope generator 64, 217, 234
equal to comparison 182
equalization 186, 190, 198, 238, 260
exponent in a power function 101
exponential curve 105, 106, 111, 202, 221, 233, 255
Extensions folder 169
ezadc~ 84, 211
ezdac~ 38, 212
fade volume in or out 36
feedback in a delay line 144, 152, 154
fft~ 135, 214
file search path of Max 269, 270, 272
file, play AIFF 272
file, play sound 95
file, record AIFF 95, 275
filter
    allpass 186
    comb 198
    lowpass 238
    resonant bandpass 260
    two-pole two-zero 190
flange 198
flanging 148
float-to-signal conversion 245, 277
FM 74, 76
foldover 17, 52, 156
Fourier synthesis 227
Fourier transform 12, 135, 214
frequency 9, 32
frequency domain 67, 135
frequency modulation 74, 76
frequency-to-MIDI conversion 216
ftom 216
function object 64, 217
gain~ 156, 221
gate~ 44, 223
greater than comparison 183
groove~ 88, 99, 114, 225
hard disk, soundfiles on 95
harmonically related sinusoids 11, 56
harmonicity ratio 76
hertz 9
ifft~ 135, 227
index~ 86, 229
info~ 89, 230
input 185, 211
input source 84, 160, 164
interference between waves 44, 133
interpolation 32, 39, 86, 128, 204, 219, 234, 245
interrupt 161
inverse fast Fourier transform 135, 227
key region 114
kink~ 232
Korg 1212I/O 174
LED display 126
less than comparison 181
level meter 239
level of a signal 34
LFO 106
limiting amplitude of a signal 197, 244
line~ 35, 234
line segment function 39
linear crossfade 122
linear mapping 104
linedrive 233
localization 120
log~ 235
logarithmic curve 202, 221, 233, 235
logarithmic scale 14, 46
logical signal transitions 210
lookup~ 80, 236

Index

lookup table 80, 92, 236, 249
loop an audio sample 88, 225
lores~ 238
loudness 14, 105
low-frequency oscillator 106
lowpass filter 238
lowpass filtered noise 256
Lucid PCI24 171
map subpatch 105
mapping a range of numbers 104
Max messages 32
meter~ 126, 239
metronome 284
MIDI 5, 103, 108
MIDI controller 29
MIDI note value 216
MIDI panning 120
MIDI-to-amplitude conversion 149, 156, 221
MIDI-to-frequency conversion 110, 241
millisecond scheduler of Max 23, 30
millisecond-to-sample conversion 240
mixing 41
mixing signals 178
modulation
  amplitude 71
  delay time 148
  frequency 74, 76
  ring 67
modulation index 76
modulation wheel 103, 108
modulator 68
Monitors & Sound control panel 163
MSP Tutorial 28
mstosamps~ 240
mtof 110, 241
multiply one signal by another 67, 177
mute~ 54, 242
mute audio of a subpatch 54, 242, 248
noise 13, 40, 151, 243, 256
noise~ 40, 243
normalize~ 146, 244
number~ 126, 245
number-to-signal conversion 245, 277
Nyquist rate 17, 52, 92, 155
on and off, turning audio 185, 206, 211, 212
oscillator 32, 204
oscilloscope 132, 264
output 164, 206, 212
Overdrive 162
panning 120
partial 11, 63
pass~ 248
Patcher, audio on in one 51
PCI24 171
PCI24 Driver API 172
pcontrol to mute a subpatch 55
peak amplitude 9, 128, 146, 239
peek~ 249
period of a wave 9
phase distortion synthesis 232
phase modulation 232, 251
phase offset 48
phasor~ 40, 251
pitch bend 106, 108
pitch-to-frequency conversion 100, 106, 241
play~ 86, 252
play AIFF 272
play audio sample 86, 88, 225, 229, 252
play audio sample as waveform 288
play sound 95
plug-in 286
poke~ 253
polyphony 108, 114
pow~ 101, 255
PowerPC 4, 26
precision of floating point numbers 59
pulse train 284
Q of a filter 238, 260
RAM 95
RAM allocation 26
rand~ 151, 256
random signal 40, 243, 256
receive~ 43, 257
record~ 85, 258
record audio 85, 258
record soundfile 95, 275
reflection of sound waves 144
reson~ 260
resonance of a filter 238, 260
ring modulation 67

Index

Roads, Curtis 8
routing a signal 44, 223
S/PDIF 171
sah~ 262
sample and hold 15, 262
sample number 201
sample stored in memory 192
sample, read single 229, 249
sample, write single 249, 253
sample-to-millisecond conversion 263
sampler 114
sampling rate 16, 24, 161, 164, 209
    of AIFF file 116, 230
sampstoms~ 263
save a sound file 86
sawtooth wave 40, 52, 251
Scheduler in Audio Interrupt 162
scope~ 132, 264
search path 269, 270, 272
selector~ 51, 266
semitone 100
send~ 43, 268
sfinfo~ 269
sflist~ 270
sfplay~ 272
sfrecord~ 275
sidebands 69, 73, 76
sig~ 47, 277
signal network 5, 23, 30
signal of constant value 245, 277
signal-to-float conversion 245, 278
simple harmonic motion 9
sine wave 8, 48, 204
slapback echo 141
snapshot~ 129, 278
Sonorus StudI/O 172
sound 8, 230
Sound control panel 163
Sound Designer II 38, 85, 272, 275
sound input 84, 160, 164, 185, 211
Sound Manager 160, 163
Sound Monitoring Source 165
sound output 164, 206, 212
spectrum 11, 67, 135
sqrt~ 279
square root of signal value 279
StudI/O 172
subpatch, mute audio of 54, 242
support, technical 170
sustain 217
switch 51, 266
synthesis techniques 63
synthesis, additive 63
tapin~ 141, 280
tapout~ 141, 281
technical support 170
temperament, equal 241
test tone 29
text, viewing a signal as 195
thresh~ 283
threshold detection 283
timbre 11
train~ 284
transfer function 80, 236
tremolo 68, 72, 130
tuning, equal temperament 241
Tutorial, MSP 28
variable speed sample playback 86, 88, 225, 252
vector size 161, 209
velocity sensitivity 108, 156
velocity-to-amplitude conversion 221
vibrato 68, 74, 100, 106
vst~ 286
VST plug-in file 286
wave~ 90, 288
waveshaping synthesis 80, 94
wavetable synthesis 31, 38, 90, 204, 288
white noise 13, 40, 243
windowing 138