MSP

Getting Started
Tutorials and Topics
Reference

Table of Contents

Introduction 5
Digital Audio: *How Digital Audio Works* 8
How MSP Works: *Max Patches and the MSP Signal Network* 22
Audio I/O: *Audio input and output with MSP* 28
Tutorial 1: *Fundamentals: Test tone* 43
Tutorial 2: *Fundamentals: Adjustable oscillator* 48
Tutorial 3: *Fundamentals: Wavetable oscillator* 53
Tutorial 4: *Fundamentals: Routing signals* 59
Tutorial 5: *Fundamentals: Turning signals on and off* 68
Tutorial 6: *Fundamentals: Review* 76
Tutorial 7: *Synthesis: Additive synthesis* 81
Tutorial 8: *Synthesis: Tremolo and ring modulation* 85
Tutorial 9: *Synthesis: Amplitude modulation* 89
Tutorial 10: *Synthesis: Vibrato and FM* 93
Tutorial 11: *Synthesis: Frequency modulation* 95
Tutorial 12: *Synthesis: Waveshaping* 99
Tutorial 13: *Sampling: Recording and playback* 104
Tutorial 14: *Sampling: Playback with loops* 109
Tutorial 15: *Sampling: Variable-length wavetable* 112
Tutorial 16: *Sampling: Record and play audio files* 117
Tutorial 17: *Sampling: Review* 121
Tutorial 18: *MIDI control: Mapping MIDI to MSP* 125
Tutorial 19: *MIDI control: Synthesizer* 130
Tutorial 20: *MIDI control: Sampler* 137
Tutorial 21: *MIDI control: Using the* poly~ *object* 143
Tutorial 22: *MIDI control: Panning* 150
Tutorial 23: *Analysis: Viewing signal data* 157
Tutorial 24*: Analysis: Oscilloscope* 163
Tutorial 25: *Analysis: Using the FFT* 166
Tutorial 26: *Frequency Domain Signal Processing with* pfft~ 172
Tutorial 27: *Processing: Delay lines* 189

Table of Contents

Tutorial 28: *Processing: Delay lines with feedback* 192
Tutorial 29: *Processing: Flange* 196
Tutorial 30: *Processing: Chorus* 200
Tutorial 31: *Processing: Comb filter* 203
MSP Reference Manual 208
The dsp Object: *Controlling and Automating MSP* 543
MSP Object Thesaurus 545
Index 551

Copyright and Trademark Notices

This manual is copyright © 2000/2003 Cycling '74.

MSP is copyright © 1997-2003 Cycling '74—All rights reserved. Portions of MSP are based on Pd by Miller Puckette, © 1997 The Regents of the University of California. MSP and Pd are based on ideas in FTS, an advanced DSP platform © IRCAM.

Max is copyright © 1990-2003 Cycling '74/IRCAM, l'Institut de Récherche et Coordination Acoustique/Musique.

VST is a trademark of Steinberg Soft- und Hardware GmbH.

ReWire is a trademark of Propellerhead Software AS.

Credits

Original MSP Documentation: Chris Dobrian

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

MSP2 Reference: David Zicarelli, Gregory Taylor, Joshua Kit Clayton, jhno, Richard Dudas, R. Luke DuBois, Andrew Pask

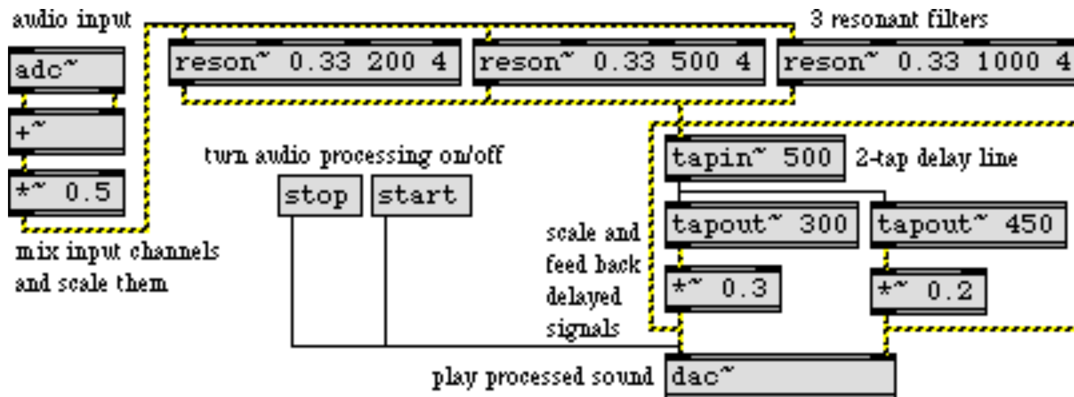
MSP2 Manual page example patches: R. Luke DuBois, Darwin Grosse, Ben Nevile, Joshua Kit Clayton, David Zicarelli

Cover Design: Lilli Wessling Hart

Graphic Design: Gregory Taylor

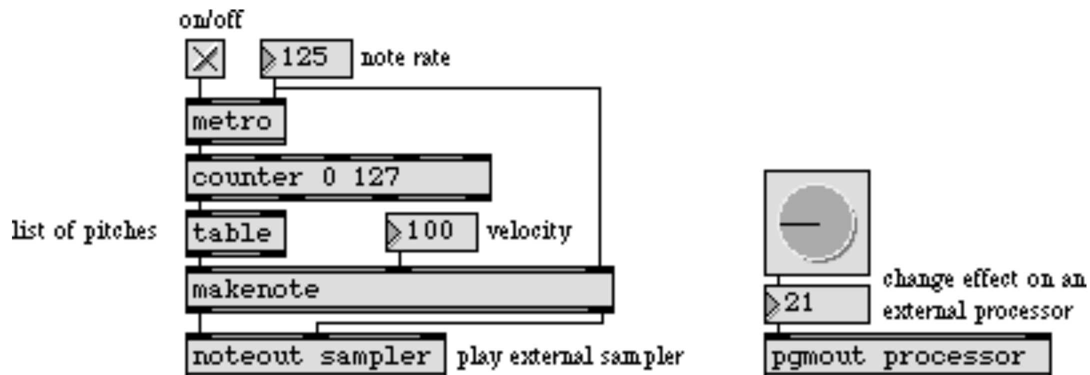
Signal processing in Max

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



A filter and delay effect processor in MSP

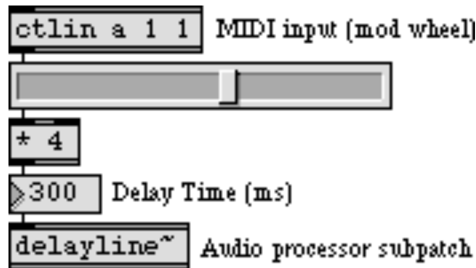
As you know, Max enables you to design your own programs for controlling MIDI synthesizers, samplers, and effects processors.



MIDI control with Max

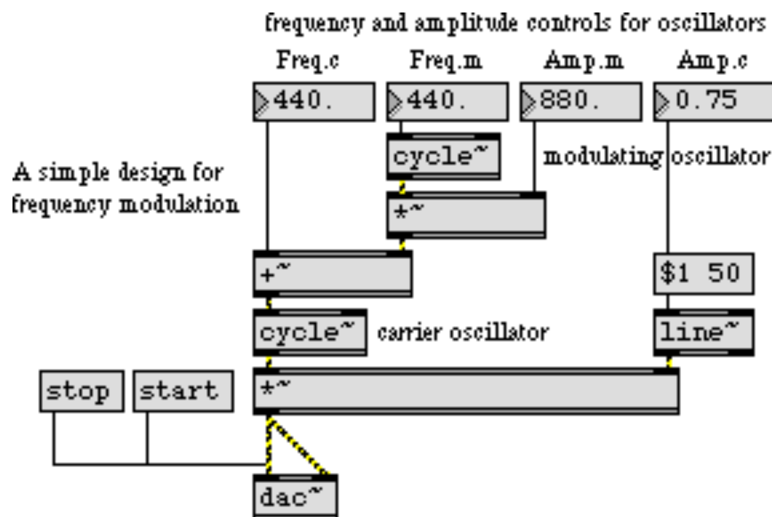
With the addition of the MSP objects, you can also create your own digital audio device designs—your own computer music *instruments*—and incorporate them directly into your Max programs.

You can specify exactly how you want your instruments to respond to MIDI control, and you can implement the entire system in a Max patch.



MIDI control of a parameter of an audio process

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



Signal network for an FM instrument

How To Use This Manual

The MSP Documentation contains the following sections:

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

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

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

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

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

Reading the manual online

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

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

Other Resources for MSP Users

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

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

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

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

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

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

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

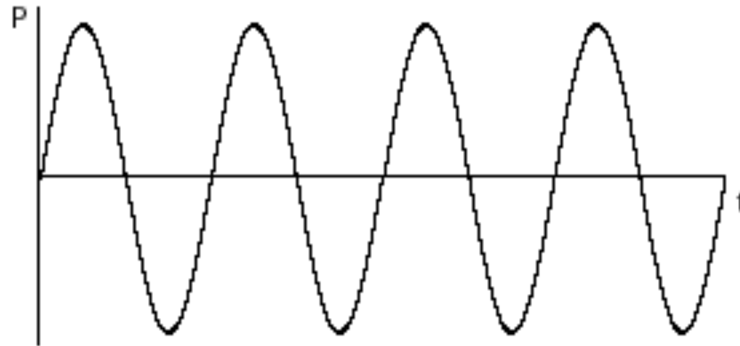
Sound

Simple harmonic motion

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

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

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



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

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

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

In order for us to hear such fluctuations of pressure:

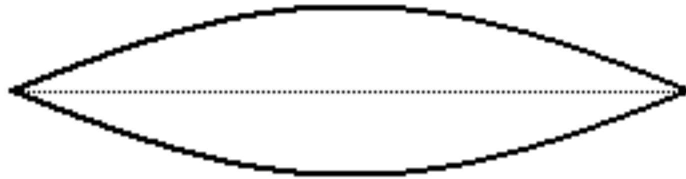
- The fluctuations must be substantial enough to affect our tympanic membrane (eardrum), yet not so substantial as to hurt us. In practice, the intensity of the changes in air pressure must be greater than about 10^{-9} times atmospheric pressure, but not greater than about 10^{-3} times atmospheric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

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

Complex tones

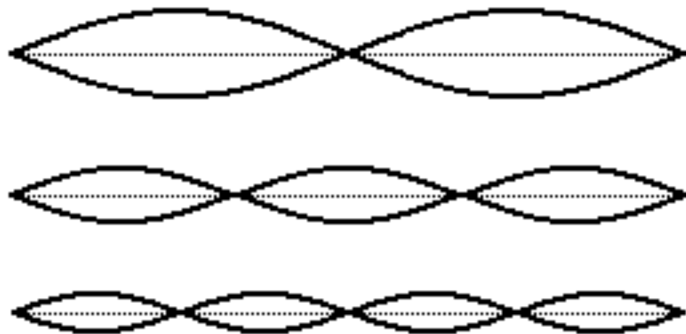
An object that vibrates in simple harmonic motion is said to have a resonant mode of vibration—a frequency at which it will naturally tend to vibrate when set in motion. However, most real-world objects have *several* resonant modes of vibration, and thus vibrate at many frequencies at once. Any sound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

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



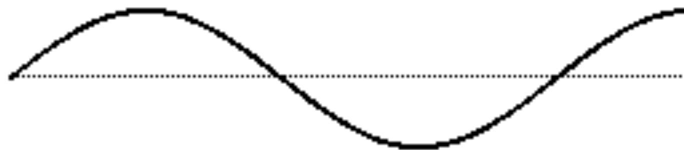
A plucked string vibrating in its fundamental resonant mode

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



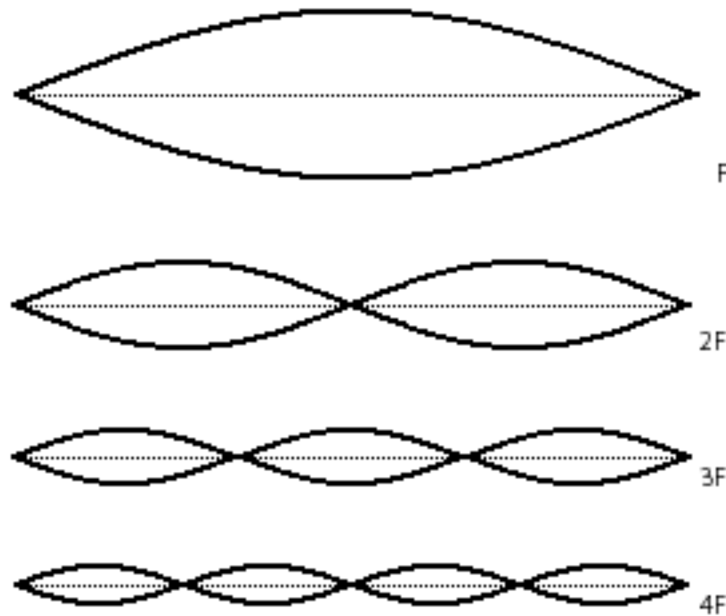
Some other resonant modes of a stretched string

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



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

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



Each resonant mode results in a different frequency

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

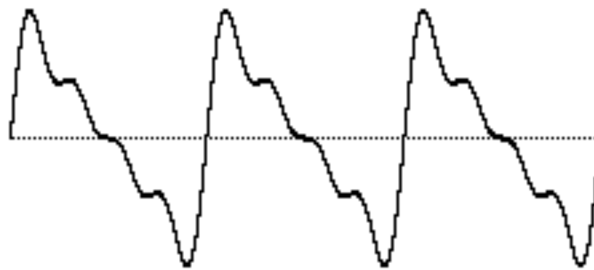
As the string's vibrations die away due to the damping force of the fixture at each end, each frequency may die away at a different rate. In fact, in many sounds the amplitudes of the different component frequencies may vary quite separately and differently from each other. This variety seems to be one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

Harmonic tones

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

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

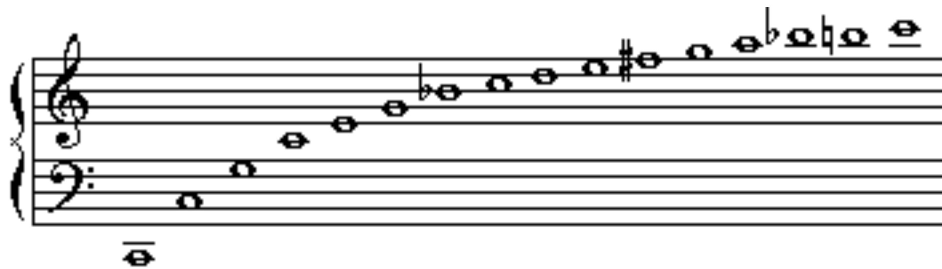
the fundamental frequency. The integer multiple frequencies thus fuse "harmoniously" into a single tone.



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

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

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



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

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

Inharmonic tones and noise

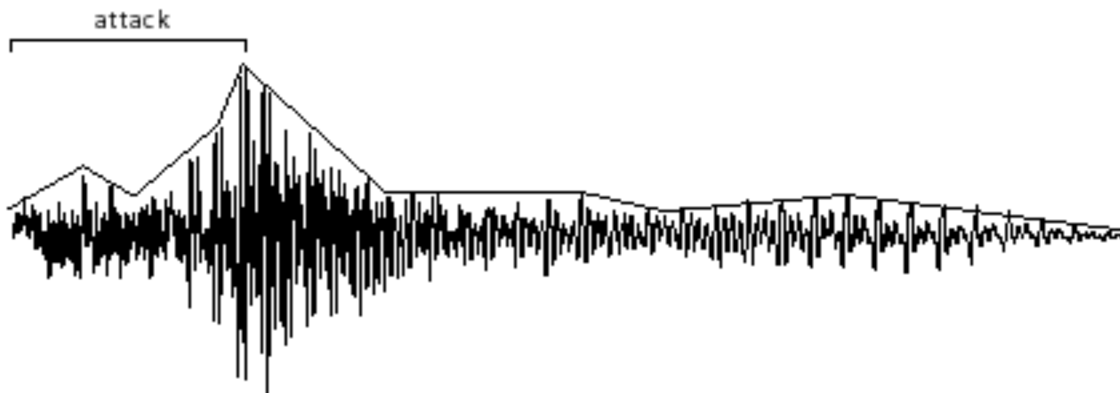
Some objects—such as a bell, for instance—vibrate in even more complex ways, with many different modes of vibrations which may not produce a harmonically related set of partials. If the frequencies present in a tone are not integer multiples of a single fundamental frequency, the wave does not repeat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

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

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

Amplitude envelope

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



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

Amplitude and loudness

The relationship between the objectively measured amplitude of a sound and our subjective impression of its loudness is very complicated and depends on many factors. Without trying to explain all of those factors, we can at least point out that our sense of the relative loudness of two sounds is related to the ratio of their intensities, rather than the mathematical difference in their intensities. For example, on an arbitrary scale of measurement, the relationship between a sound

of amplitude 1 and a sound of amplitude 0.5 is the same to us as the relationship between a sound of amplitude 0.25 and a sound of amplitude 0.125. The subtractive difference between amplitudes is 0.5 in the first case and 0.125 in the second case, but what concerns us perceptually is the ratio, which is 2:1 in both cases.

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

Nevertheless, there is a correlation—even if not perfectly linear—between amplitude and loudness, so it’s certainly informative to know the relative amplitude of two sounds. As mentioned earlier, the softest sound we can hear has about one millionth the amplitude of the loudest sound we can bear. Rather than discuss amplitude using such a wide range of numbers from 0 to 1,000,000, it is more common to compare amplitudes on a logarithmic scale.

The ratio between two amplitudes is commonly discussed in terms of ***decibels*** (abbreviated dB). A ***level*** expressed in terms of decibels is a statement of a ratio relationship between two values—not an absolute measurement. If we consider one amplitude as a reference which we call 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

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

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

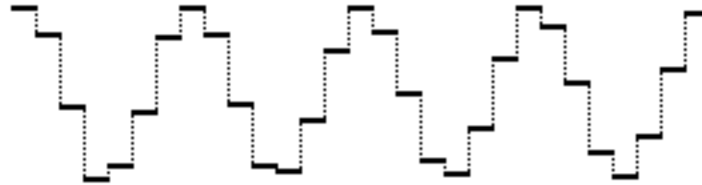
A theoretical understanding of sine waves, harmonic tones, inharmonic complex tones, and noise, as discussed here, is useful to understanding the nature of sound. However, most sounds are actually complicated combinations of these theoretical descriptions, changing from one instant to another. For example,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

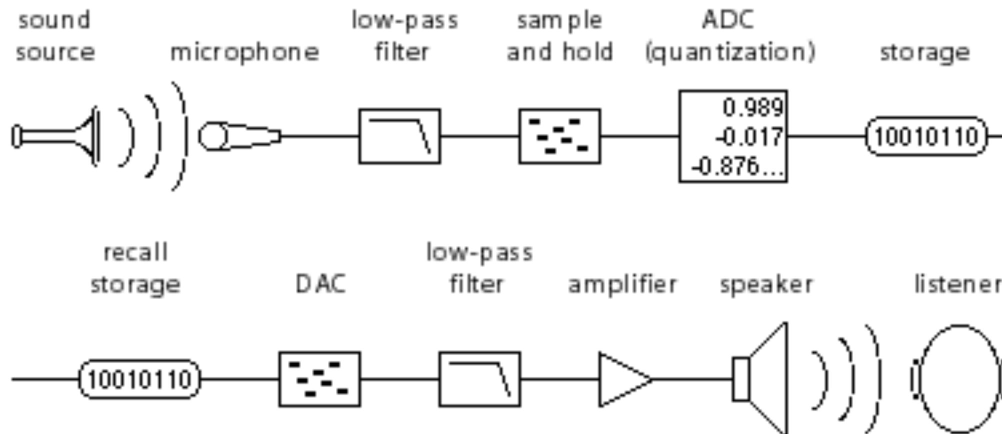
A device known as an *analog-to-digital converter* (ADC) receives the discrete voltages from the sample and hold device, and ascribes a numerical value to each amplitude. This process of converting voltages to numbers is known as *quantization*. Those numbers are expressed in the computer as a string 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 samples must be taken per second— many more than are necessary for filming a visual image. In fact, we need to take more than twice as many samples as the highest frequency we wish to record. (For an explanation of why this is so, see *Limitations of Digital Audio* on the next page.) If we want to record frequencies as high as 20,000 Hz, we need to sample the sound at least 40,000 times per second. The standard for compact disc recordings (and for "CD-quality" computer audio) is to take 44,100 samples per second for each channel of audio. The number of samples taken per second is known as the *sampling rate*.

This means the computer can only accurately represent frequencies up to half the sampling rate. Any frequencies in the sound that exceed half the sampling rate must be filtered out before the sampling process takes place. This is accomplished by sending the electrical signal through a *low-pass filter* which removes any frequencies above a certain threshold. Also, when the digital signal (the stream of binary digits representing the quantized samples) is sent to the DAC to be re-converted into a continuous electrical signal, the sound coming out of the DAC will contain spurious

high frequencies that were created by the sample and hold process itself. (These are due to the "sharp edges" created by the discrete samples, as seen in the above example.) Therefore, we need to send the output signal through a low-pass filter, as well.

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



Digital recording and playback process

Limitations of digital audio

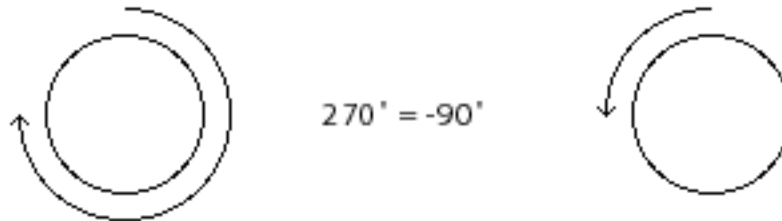
Sampling rate and Nyquist rate

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

If we take, for example, 16,000 samples of an audio signal per second, we can only capture frequencies up to 8,000 Hz. Any frequencies higher than the Nyquist rate are perceptually "folded" back down into the range below the Nyquist frequency. So, if the sound we were trying to sample contained energy at 9,000 Hz, the sampling process would misrepresent that frequency as 7,000 Hz—a frequency that might not have been present at all in the original sound. This effect is known as *foldover* or *aliasing*. The main problem with aliasing is that it can add frequencies to the digitized sound that were not present in the original sound, and unless we know the exact spectrum of the original sound there is no way to know which frequencies truly belong in the digitized sound and which are the result of aliasing. That's why it's essential to use the low-pass filter before the sample and hold process, to remove any frequencies above the Nyquist frequency.

To understand why this aliasing phenomenon occurs, think back to the example of a film camera, which shoots 24 frames per second. If we're shooting a movie of a car, and the car wheel spins at a rate greater than 12 revolutions per second, it's exceeding half the "sampling rate" of the camera. The wheel completes more than $^1/_2$ revolution per frame. If, for example it actually completes $^{18}/_{24}$ of a revolution per frame, it will appear to be going backward at a rate of 6 revolutions per second. In other words, if we don't witness what happens between samples, a 270° revolution of the wheel

is indistinguishable from a -90° revolution. The samples we obtain in the two cases are precisely the same.



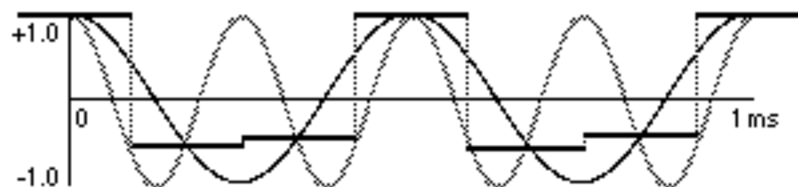
For the camera, a revolution of $^{18}/_{24}$ is no different from a revolution of $-^{6}/_{24}$

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

For a demonstration, consider the next two examples. The following example shows a graph of a 4,000 Hz cosine wave (energy only at 4,000 Hz) being sampled at a rate of 22,050 Hz. 22,050 Hz is half the CD sampling rate, and is an acceptable sampling rate for sounds that do not have much energy in the top octave of our hearing range. In this case the sampling rate is quite adequate because the maximum frequency we are trying to record is well below the Nyquist frequency.

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

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



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

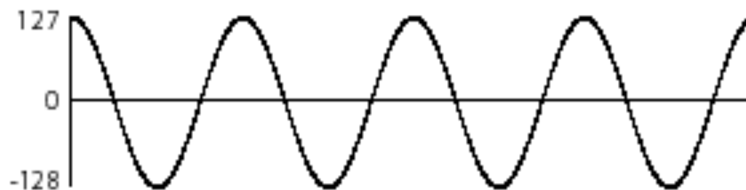
The simple lesson to be learned from the Nyquist theorem is that digital audio cannot accurately represent any frequency greater than half the sampling rate. Any such frequency will be misrepresented by being folded over into the range below half the sampling rate.

Precision of quantization

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

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

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



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

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

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

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

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

As a rule of thumb, each bit of precision used in quantization adds 6 dB to the SQNR. Therefore, sound quantized with 8-bit numerical precision will have a best case SQNR of about 48 dB. This is adequate for cases where fidelity is not important, but is certainly not desirable for music or other critical purposes. Sound sampled with 16-bit precision ("CD-quality") has a SQNR of 96 dB, which is quite good—much better than traditional tape recording.

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

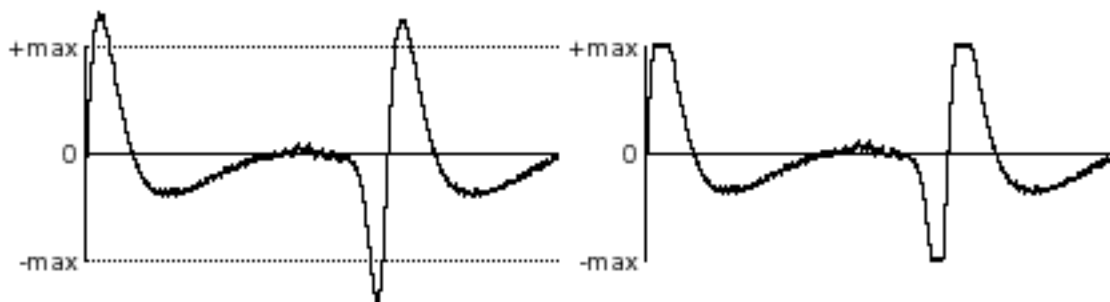
Memory and storage

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

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

Clipping

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



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

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

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

Advantages of digital audio

Synthesizing digital audio

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

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

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

$$y = A \sin(2_fn/R+ø)$$

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

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

$$y = A_1 \sin(2_f_1n/R+ø_1) + A_2 \sin(2_f_2n/R+ø_2) + \dots$$

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.

How MSP Works

Max Patches and the MSP Signal Network

Introduction

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

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

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

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

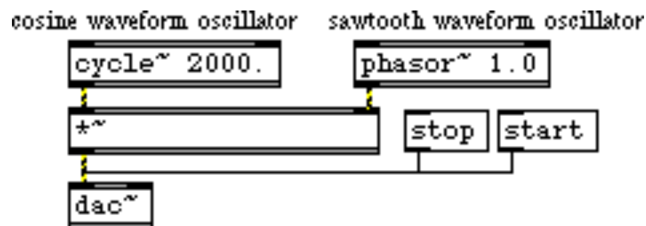
In the Max example on the left, the **number box** doesn't know about the number 0.75 stored in the **float** object. When the user clicks on the **button**, the **float** object sends out its stored value. Only then does the **number box** receive, display, and send out the number 0.75. In the MSP example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

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

Audio rate and control rate

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

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



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

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

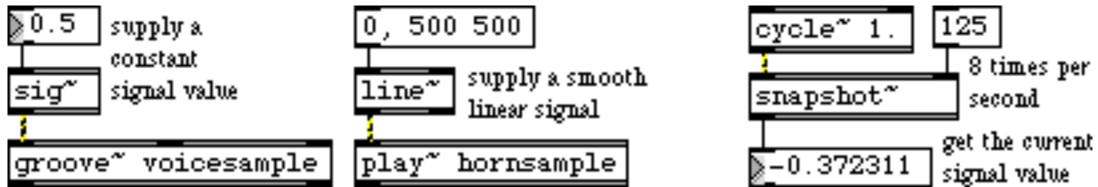
Even though many MSP objects accept input values expressed in milliseconds, they calculate samples at an audio sampling rate. Max messages travel much more slowly, at what is often referred to as a *control* rate. It is perhaps useful to think of there being effectively two different rates of activity: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

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

The link between Max and MSP

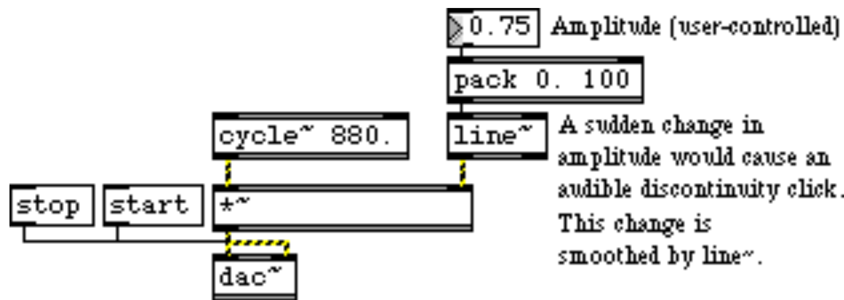
Some MSP objects exist specifically to provide a link between Max and MSP—and to translate between the control rate and the audio rate. These objects (such as **sig~** and **line~**) take Max messages in their inlets, but their outlets connect to the signal network; or conversely, some objects

(such as **snapshot~**) connect to the signal network and can peek (but only as frequently as once per millisecond) at the value(s) present at a particular point in the signal network.



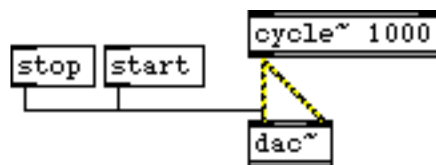
Supply a Max message to the signal network, or get a Max message from a signal

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



User interface control over the signal's amplitude

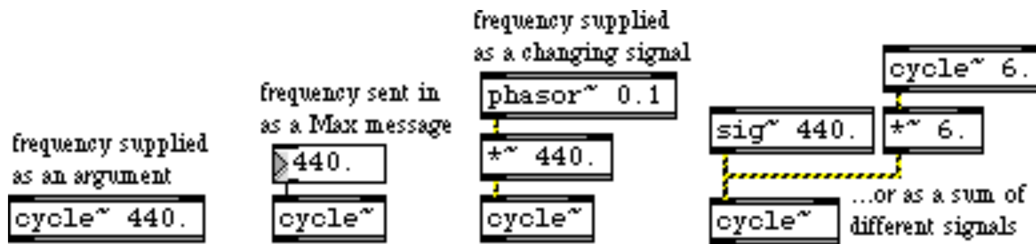
Some MSP object inlets accept both signal input and Max messages. They can be connected as part of a signal network, and they can also receive instructions or modifications via Max messages. For example the **dac~** (digital-to-analog converter) object, for playing the audio signal, can be turned on and off with the Max messages start and stop.



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

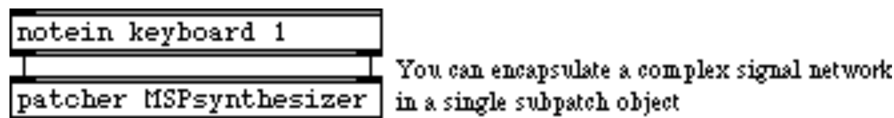
And the **cycle~** (oscillator) object can receive its frequency as a Max float or int message, or it can receive its frequency from another MSP object (although it can't do both at the same time, because

the audio input can be thought of as constantly supplying values that would immediately override the effect of the float or int message).



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

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



Encapsulation can clarify relationships in a Max patch

Limitations of MSP

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

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

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

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

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

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

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

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

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

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

Advantages of MSP

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

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

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

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

4. *Integrate audio processing into your composition or performance programs.* If your musical work consists of devising automated composition programs or computer-assisted performances in

Max, now you can incorporate audio processing into those programs. Need to do a hands-free crossfade between your voice and a pre-recorded sample at a specific point in a performance? You can write a Max patch with MSP objects that does it for you, triggered by a single MIDI message.

Some of these ideas are demonstrated in the MSP tutorials.

See Also

Audio I/O	Audio input and output with MSP

Audio I/O

Audio input and output with MSP

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

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

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

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

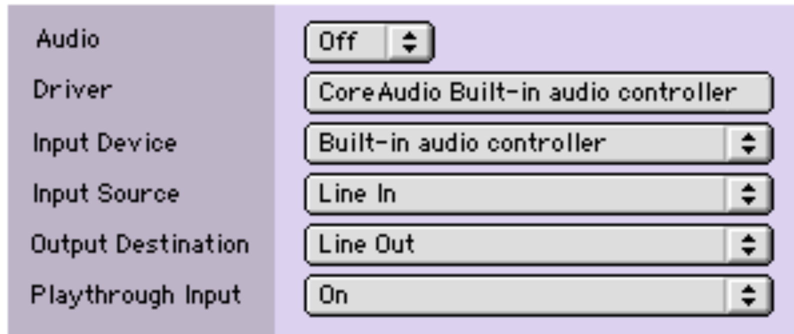
The DSP Status Window

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

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

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

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



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

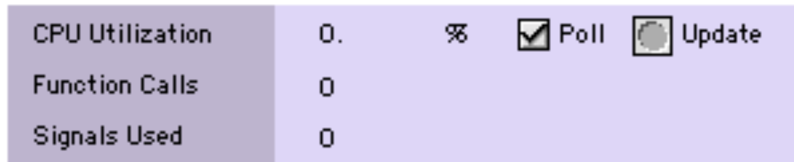
None	This setting shuts off MSP audio processing.
Core Audio	This is the default audio driver for MSP on Macintosh. It interfaces with the system's built-in Core Audio system and can be used with the built-in audio of the computer, or, with the proper software support, a third-party hardware interface, such as ASIO.
MME or DirectSound	(Windows only) On Windows, MSP loads the MME driver by default. If you have correctly installed external hardware and it also supports DirectSound, it should also appear as an option on the pop-up menu.
ASIO	(Windows only) If you have a third-party audio interface which supports ASIO (a cross-platform audio hardware standard developed by Steinberg), and it is installed correctly, it will be found by the MSP ASIO driver. You may have as many ASIO devices as you wish; they will all be found by the driver and will appear in the Driver pull-down menu in the DSP Status Window preceded by the word ASIO.
NonRealTime	This driver enables MSP to work in non real-time mode, allowing you to synthesize and process audio without any real-time processor performance limitations. Real-time audio input and output are disabled under this driver.

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

The next two pop-up menus are active only when using the Core Audio driver on Macintosh or ASIO drivers. When the Core Audio driver or either the MME or DirectSound drivers on Windows are selected, the pop-up menus allow you to change the audio input source. On Macintosh only, an additional pop-up menu lets you choose whether or not audio playthrough is enabled. These settings can also be changed using the Audio MIDI Setup application on Macintosh or the

Sounds and Audio Devices Properties window (Start – Settings – Control Panel – Sounds and Audio Devices) on Windows, but only with these menus while MSP is running.

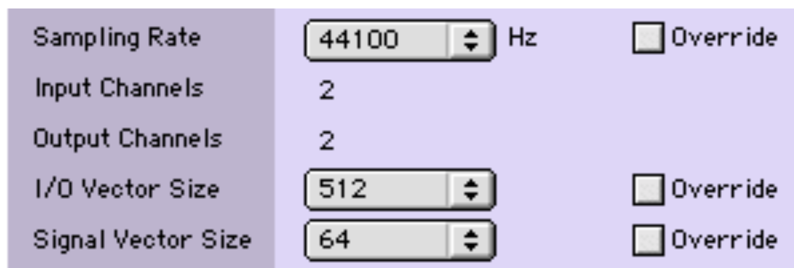
When ASIO is in use, the pop-up menus allow you to set the clock source for your audio hardware and whether or not to prioritize MIDI input and output over audio I/O.



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

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

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



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

he *I/O Vector Size* may have an effect on latency and overall performance. A smaller vector size may reduce the inherent delay between audio input and audio output, because MSP has to perform calculations for a smaller chunk of time. On the other hand, there is an additional computational burden each time MSP prepares to calculate another vector (the next chunk of audio), so it

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

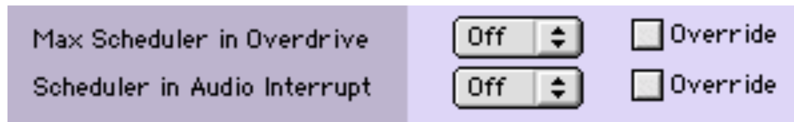
Note: Some audio interface cards do not provide a choice of I/O Vector Sizes. There are also some ASIO drivers whose selection of I/O Vector Sizes may not conform to the multiple-of-a-power-of-2 limitation currently imposed by MSP's ASIO support. In some cases, this limitation can be remedied by using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

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

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

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

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



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

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

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

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

Mappings window). In addition, you can use the **adstatus** object from within your patch to map any of the 512 logical audio I/O channels.

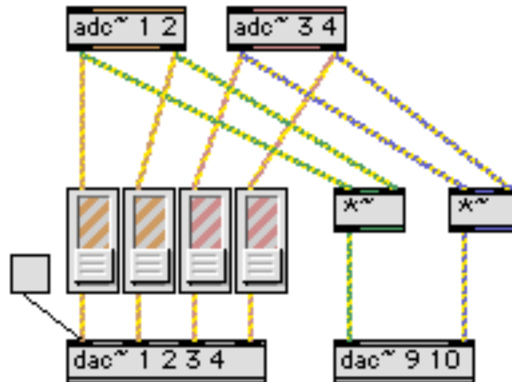
The *Optimize* pop-up menu is found only on the Macintosh version of MSP. It allows you to select whether G4 (Altivec) vector optimization will be used by MSP when computing audio. Vector optimization allows four samples to be processed within the space of a single instruction. However, not all audio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

The *CPU Limit* option allows you to set a limit (expressed in terms of a percentage of your computer's CPU) to how much signal processing MSP is allowed to do. MSP will not go above the set CPU limit for a sustained period, allowing your computer to perform other tasks without MSP locking them out. The trade-off, however, is that you'll hear clicks in the audio output when the CPU goes over the specified limit. Setting this value to either '0' or '100' will disable CPU limiting.

About Logical Input and Output Channels

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

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

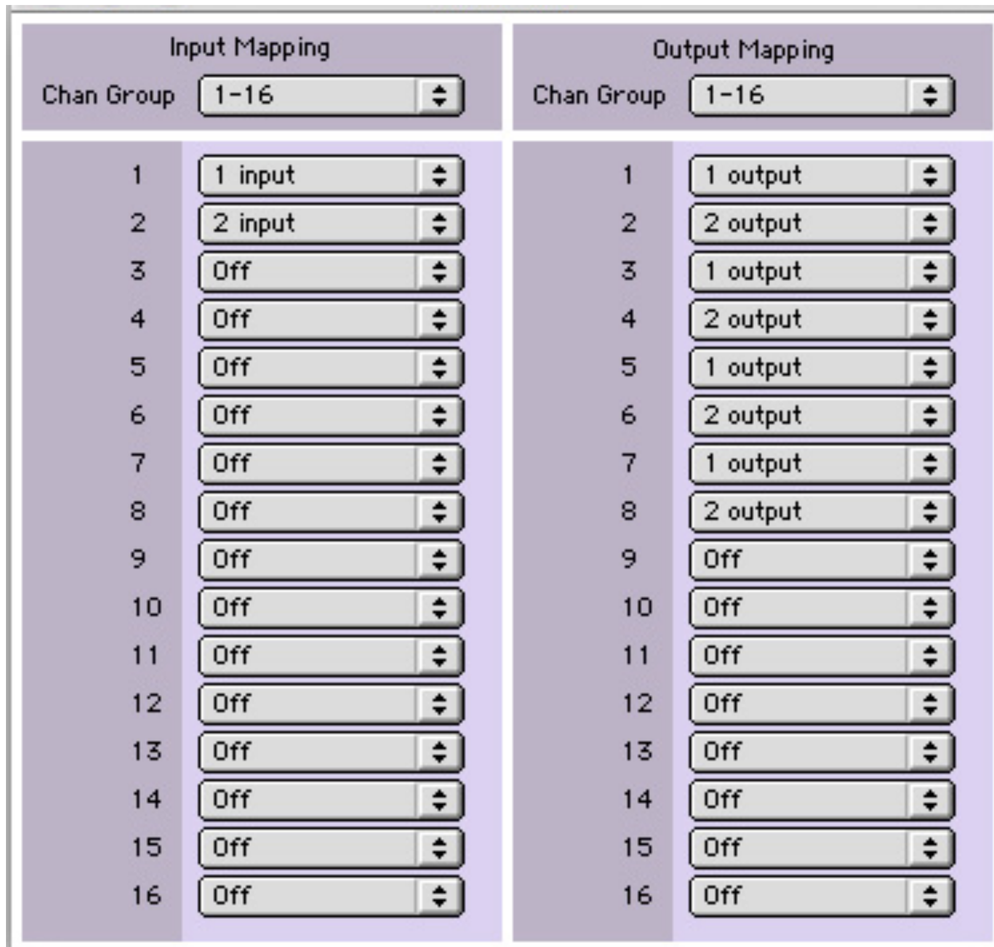


In this example, two separate **adc~** objects output audio signals from logical channel pairs 1/2 and 3/4, respectively. The output of all four channels is sent to **gain~** objects which attenuate the incoming signals and send them to the first four logical output channels, as specified by the first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

The purpose of having both logical channels and physical device channels is to allow you to create patches that use as many channels as you need without regard to the particular hardware configuration you're using. For instance, some audio interfaces use physical device channels 1 and 2 for S/PDIF input and output. If you don't happen to have a S/PDIF-compatible audio interface, you may wish to use channels 8 and 9 instead. With MSP 1.x, you would have been required to change all instances of **dac~** and/or **adc~** objects with arguments 1 and 2 to have arguments 8 and 9. With MSP 2, this is no longer necessary. You can simply go to the DSP Status window and choose the eighth and ninth physical channels listed in the Input and Output pop-up menus.

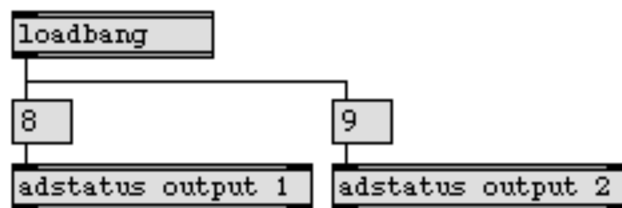
Logical channels in MSP are only created if there is a **dac~** or **adc~** object using them. In other words, if you're only using logical outputs 1 and 2, there aren't 510 unused audio streams sitting around hogging your CPU. However, since you can mix any number of logical channels to a single physical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

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



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

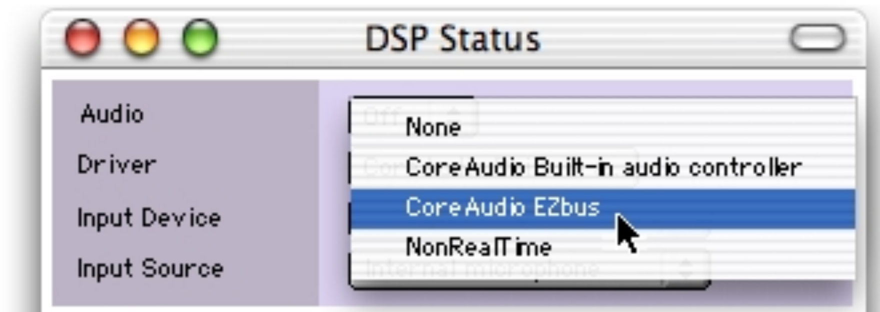
I/O Mappings are saved for each audio driver. You can also create I/O mappings within your patch using the **adstatus** object. The example patch below accomplishes the same remapping as that shown in the I/O Mapping window above, but does so automatically when the patch is loaded.



Using Core Audio on Macintosh

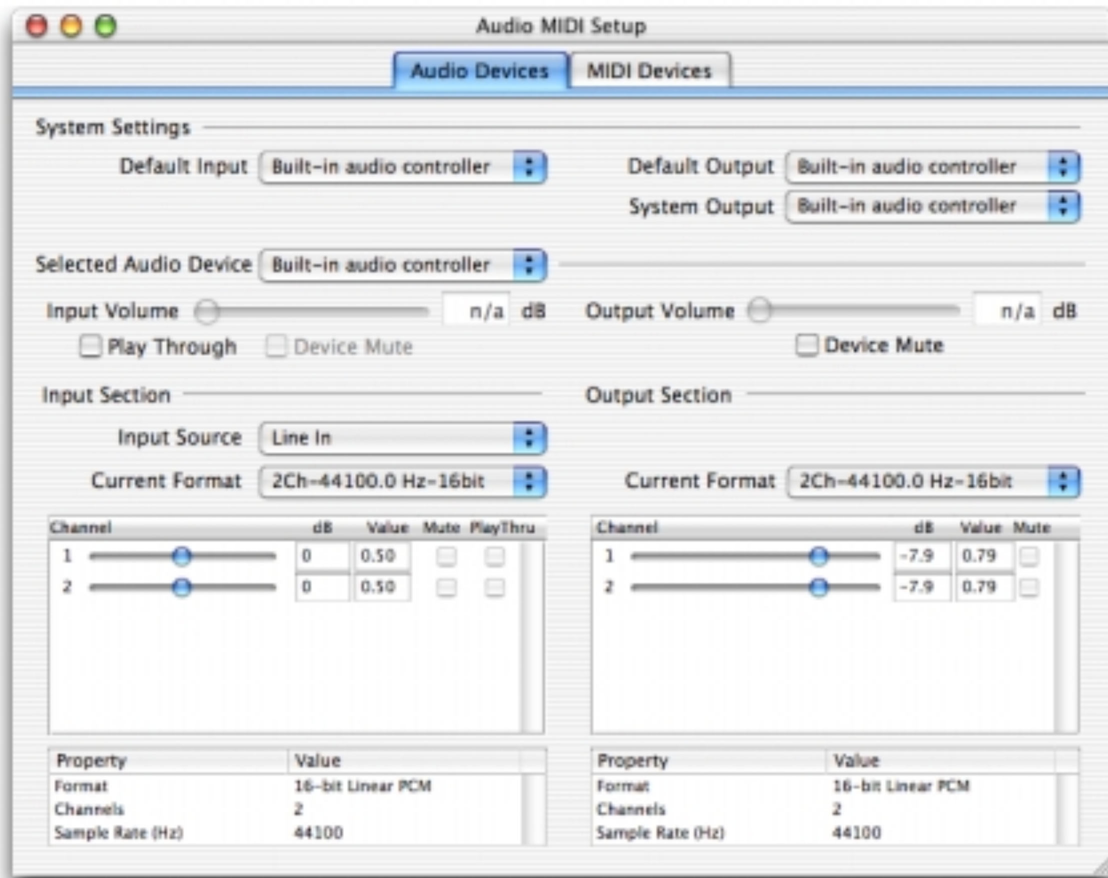
On Macintosh, MSP uses the Core Audio driver by default. As with all audio drivers, the Core Audio object file must be located in a folder called ad which is placed in /Library/Application Support/Cycling '74/. Core Audio is available on all Macintoshes running Mac OS X 10.2 or later, and provides Audio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

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



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

You can open the Audio MIDI Setup application by clicking on the Open Audio Control Panel button in the lower left corner of the DSP Status Window.



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

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

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

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

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

```
; dsp driver playthrough 0
```

Using an argument of 1 will turn it on.

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

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

Using MME Audio and DirectSound on Windows

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

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

DirectSound drivers, built on Microsoft's DirectX technology, have become commonplace for most sound cards, and provide much better latency and performance than MME drivers. Whenever possible, a DirectSound driver (ad_directsound) should be used in preference to an MME driver. Occasionally, (and especially in the case of motherboard-based audio systems) you will find the DirectSound driver performs more poorly than the MME driver. This can happen when a hardware-specific DirectSound driver is not available, and the system is emulating DirectSound while using the MME driver. In these cases, it is best to use MME directly, or find an ASIO driver for your system.

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

Using MME and DirectSound Drivers on with MSP on Windows

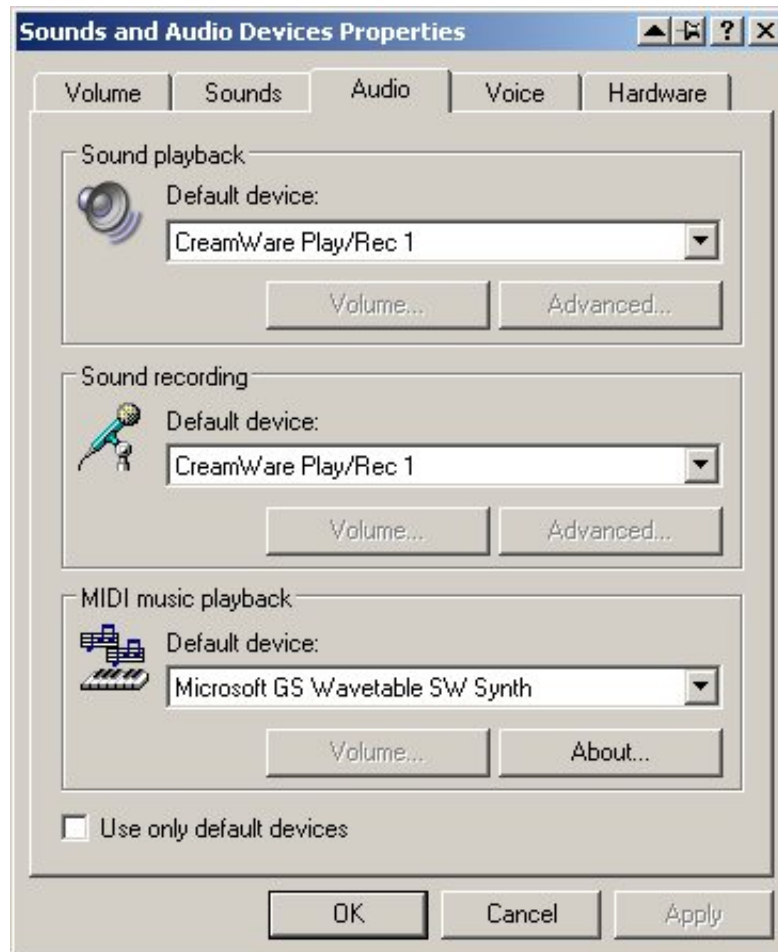
On Windows, MSP loads the MME driver by default. The MSP MME and DirectSound drivers are located in C:\Program Files\Common Files\Cycling '74\ad\.

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

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

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

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



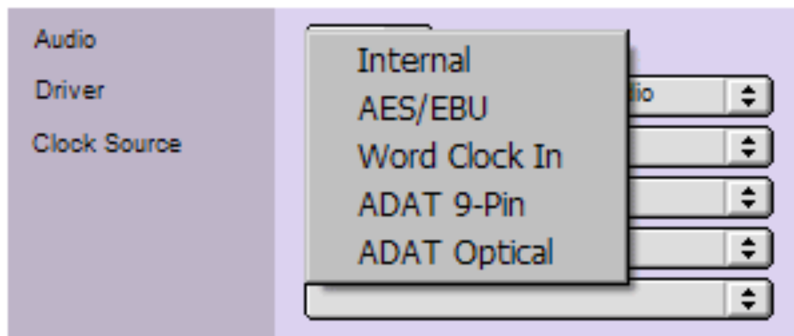
MSP supports the use of different input and output devices with MME and DirectSound drivers. Use the DSP Status Window to choose input and output devices.

Using ASIO on Windows

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

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

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



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

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

Controlling ASIO Drivers with Messages to the dsp Object on Windows

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

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

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

Here are some examples of the monitor message:

; dsp driver monitor 1 1	Patches input 1 to output 1 at unity gain with center pan.
; dsp driver monitor 1 0	Turns off input 1
; dsp driver monitor 1 4 2. -1.	patches input 1 to output 4 with 6dB gain panned to the left

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

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

; dsp driver timecode 1

To stop timecode reading, send the following message:

; dsp driver timecode 0

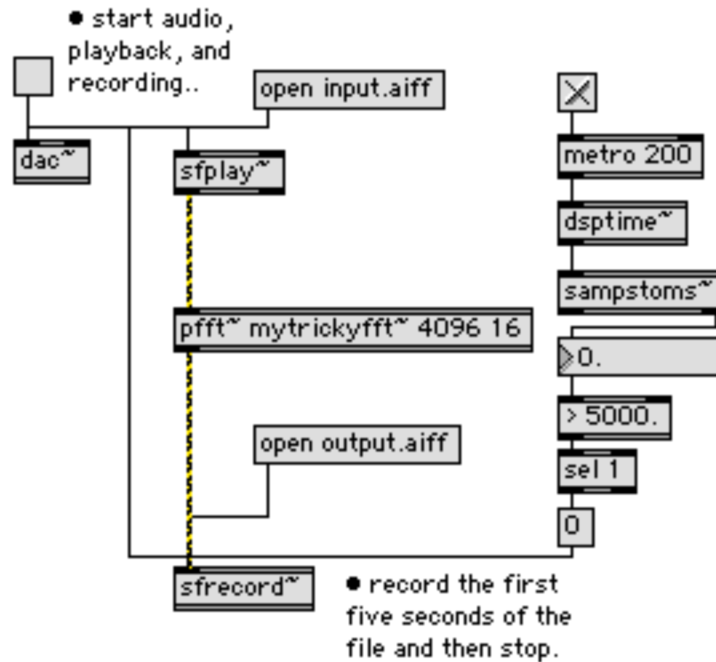
The **plugsync~** object reports the sample position reported by the audio interface when you enable timecode, but there isn't currently an object that reports the timecode of the interface.

Working in Non-Real Time with MSP

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

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

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



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

See Also

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

Tutorial 1

Fundamentals: Test tone

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

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

MSP objects are pretty much like Max objects

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

...but they're a little different

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

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

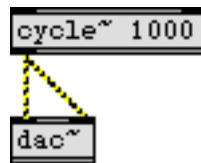
Here's another helpful way to think of the difference. Think of a patch of MSP objects not as a program in which events occur at specific instants (as in a standard Max patch), but rather as a description of an instrument design—a synthesizer, sampler, or effect processor. It's like a mathematical formula,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

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

...so they look a little different

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

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

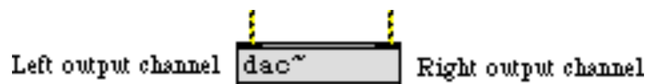


MSP objects are connected by striped patch cords

Digital-to-analog converter: dac~

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

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



dac~ *plays the signal*

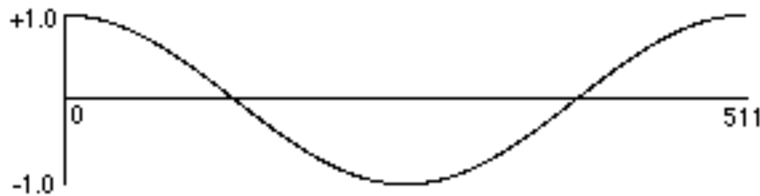
Important! dac~ must be receiving a signal of non-zero amplitude in order for you to hear anything. **dac~** expects to receive signal values in the range -1.0 to 1.0. Numbers that exceed that range will cause distortion when the sound is played.

Wavetable synthesis: cycle~

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

You can direct **cycle~** to read from a list of values that you supply (in the form of an audio file), or if you don't supply one, it will read through its own table which represents a cycle of a cosine wave

with an amplitude of 1. We'll show you how to supply your own waveform in *Tutorial 3*. For now we'll use the cosine waveform.



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

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

Technical detail: To figure out how far to step through the list for each consecutive sample, **cycle~** uses the basic formula

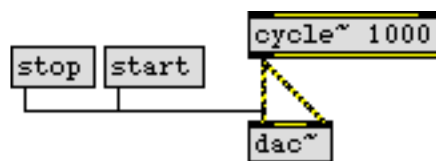
$$I=fL/R$$

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

By default **cycle~** has a frequency of 0 Hz. So in order to hear the signal, we need to supply an audible frequency value. This can be done with a number argument as in the example patch, or by sending a number in the left inlet, or by connecting another MSP object to the left inlet.

Starting and stopping signal processing

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



The simplest possible signal network

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

Listening to the Test Tone

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

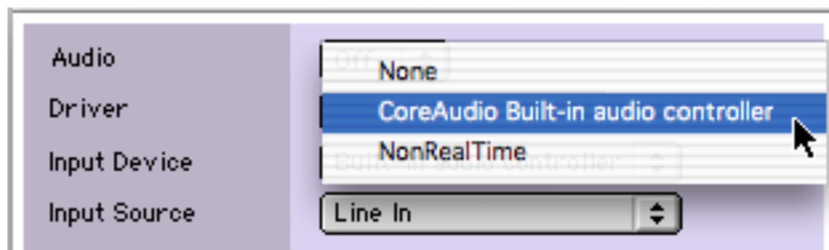
In order to get MSP up and running properly, we recommend that you start the tutorials using your computer's built-in sound hardware. If you want to use an external audio interface or sound care, please refer to the Audio Input and Output chapter for details on configuring MSP to work with audio hardware.

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

Troubleshooting

If you don't hear any sound coming from your computer when you start the **dac~** in this example, check the level setting on your amplifier or mixer, and check all your audio connections. Check that the sound output isn't currently muted. On Macintosh, the sound output level is set using the Sound preferences in the System Preferences application. On Windows, the sound output level is set using the Sounds and Audio Devices setup (Start - Control Panels - Sounds and Audio Devices).

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



Summary

A *signal network* of connected MSP objects describes an audio instrument. The digital-to-analog converter of the instrument is represented by the **dac~** object; **dac~** must be receiving a signal of non-zero amplitude (in the range -1.0 to 1.0) in order for you to hear anything. The **cycle~** object is a wavetable oscillator which reads cyclically through a list of 512 amplitude values, at a rate

determined by the supplied frequency value. Signal processing is turned on and off by sending a start or stop message to any **dac~** or **adc~** object.

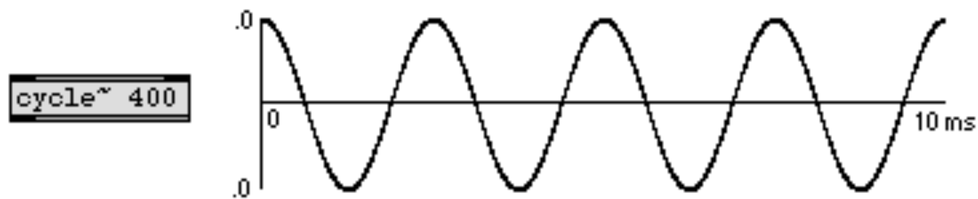
- Close the Patcher window before proceeding to the next chapter.

See Also

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

Amplifier: *~

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



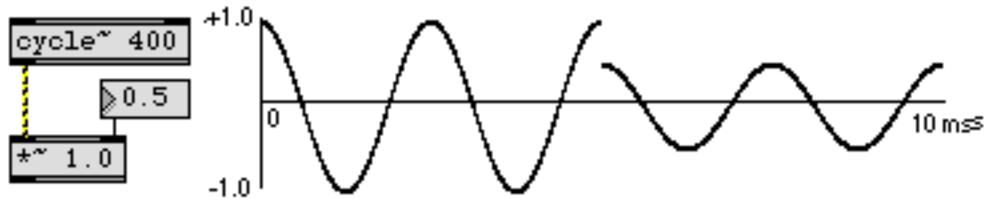
The default output of **cycle~** *has amplitude of 1*

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

Amplitude adjusted by multiplication

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

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

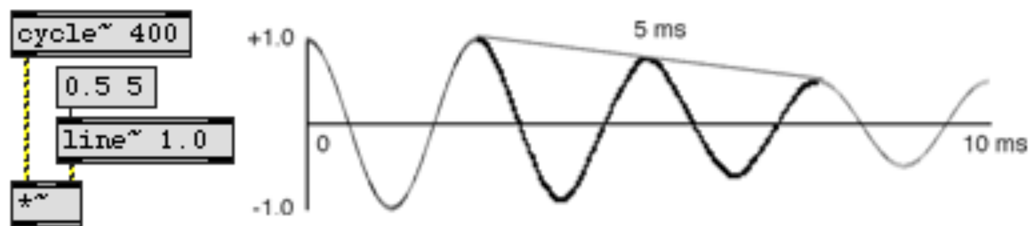


Instantaneous change of amplitude causes a noisy distortion of the signal

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

Line segment generator: line~

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

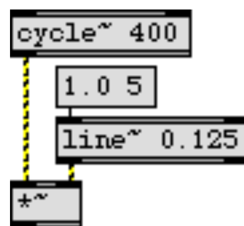


Linear amplitude change over 5 milliseconds using **line~**

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

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

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

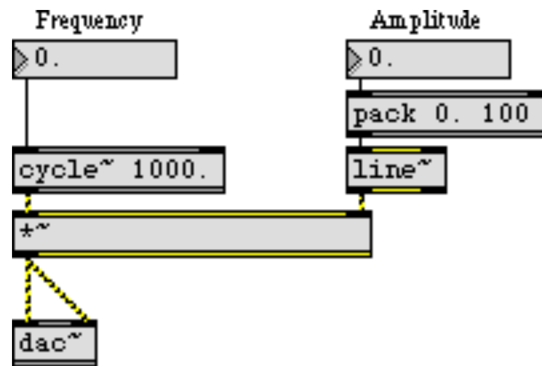


An eightfold (18 dB) increase in 5 ms creates a percussive effect

Adjustable oscillator

The example patch uses this combination of ***~** and **line~** to make an adjustable amplifier for scaling the amplitude of the oscillator. The **pack** object appends a transition time to the target ampli-

tude value, so every change of amplitude will take 100 milliseconds. A **number box** for changing the frequency of the oscillator has also been included.



Oscillator with adjustable frequency and amplitude

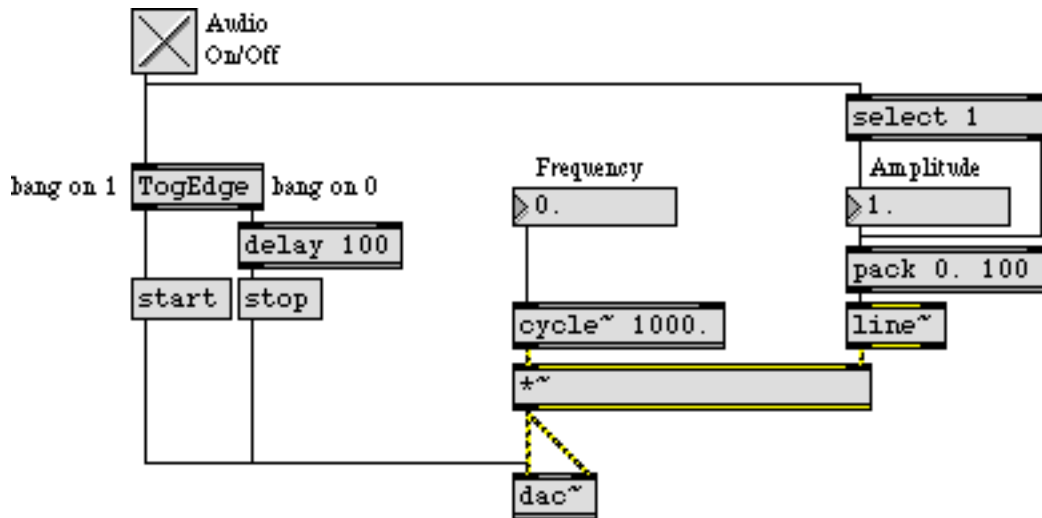
Notice that the signal network already has default values before any Max message is sent to it. The **cycle~** object has a specified frequency of 1000 Hz, and the **line~** object has a default initial value of 0. Even if the *~ had a typed-in argument for initializing its right inlet, its right operand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

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

Fade In and Fade Out

The combination of **line~** and *~ also helps to avoid the clicks that can occur when the audio is turned on and off. The 1 and 0 "on" and "off" messages from the **toggle** are used to fade the volume

up to the desired amplitude, or down to 0, just as the start or stop message is sent to **dac~**. In that way, the sound is faded in and out gently rather than being turned on instantaneously.



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

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

Summary

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

See Also

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

Tutorial 3

Fundamentals: Wavetable oscillator

Audio on/off switch: ezdac~

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

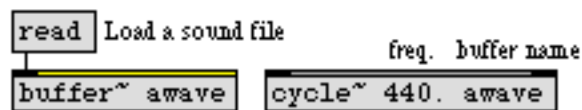
ezdac~ *is an on/off button for audio, available in the object palette*

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

A stored sound: buffer~

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

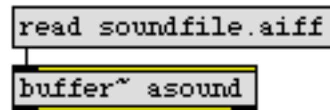
A **buffer~** object requires a unique name typed in as an argument. A **cycle~** object can then be made to read from that buffer by typing the same name in as its argument. (The initial frequency value for **cycle~**, just before the buffer name, is optional.)



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

To get the sound into the **buffer~**, send it a read message. That opens an Open Document dialog box, allowing you to select an audio file to load. The word read can optionally be followed by a spe-

cific file name, to read a file in without selecting it from the dialog box, provided that the audio file is in Max's search path.



Read in a specific sound immediately

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

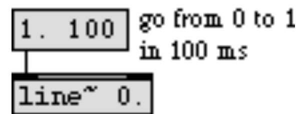
> **Technical detail:** In fact, **cycle~** uses 513 samples. The 513th sample is used only for interpolation from the 512th sample. When **cycle~** is being used to create a periodic waveform, as in this example patch, the 513th sample should be the same as the 1st sample. If the **buffer~** contains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

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

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

Create a breakpoint line segment function with line~

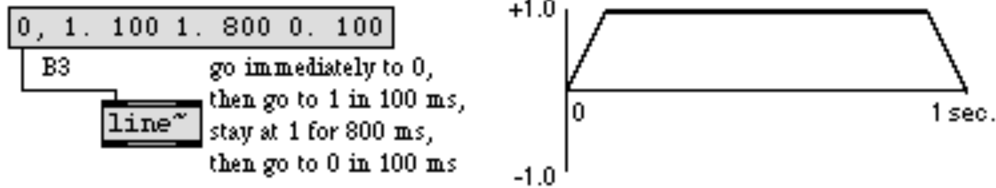
In the previous example patch, we used **line~** to make a linearly changing signal by sending it a list of two numbers. The first number in the list was a target value and the second was the amount of time, in milliseconds, for **line~** to arrive at the target value.



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

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

immediately toward the next target value in the list, taking the specified amount of time to get there.

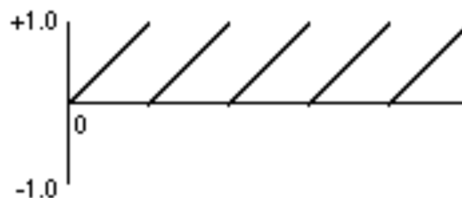


A function made up of line segments

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

Other signal generators: phasor~ and noise~

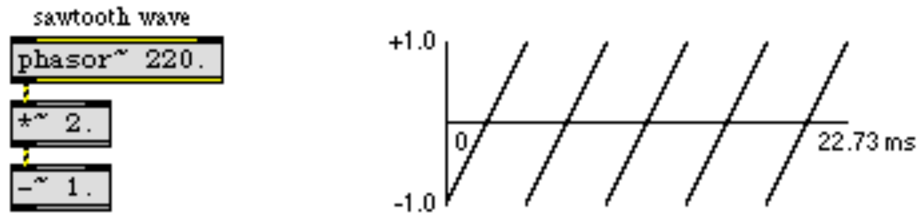
The **phasor~** object produces a signal that ramps repeatedly from 0 to 1.



Signal produced by **phasor~**

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

high frequency, of course, it is audible as a sawtooth waveform. In the example patch, the **phasor~** is pitched an octave above **cycle~**, and its output is scaled and offset so that it ramps from -1 to +1.



220 Hz sawtooth wave

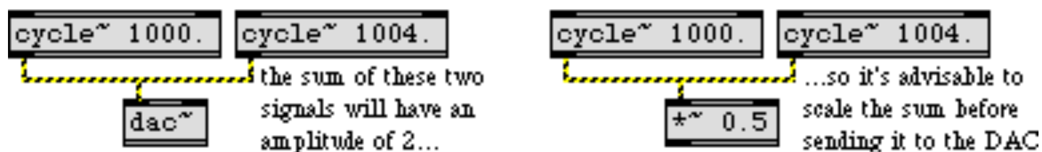
> **Technical detail:** A sawtooth waveform produces a harmonic spectrum, with the amplitude of each harmonic inversely proportional to the harmonic number. Thus, if the waveform has amplitude A, the fundamental (first harmonic) has amplitude A, the second harmonic has amplitude A/2, the third harmonic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

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

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

Add signals to produce a composite sound

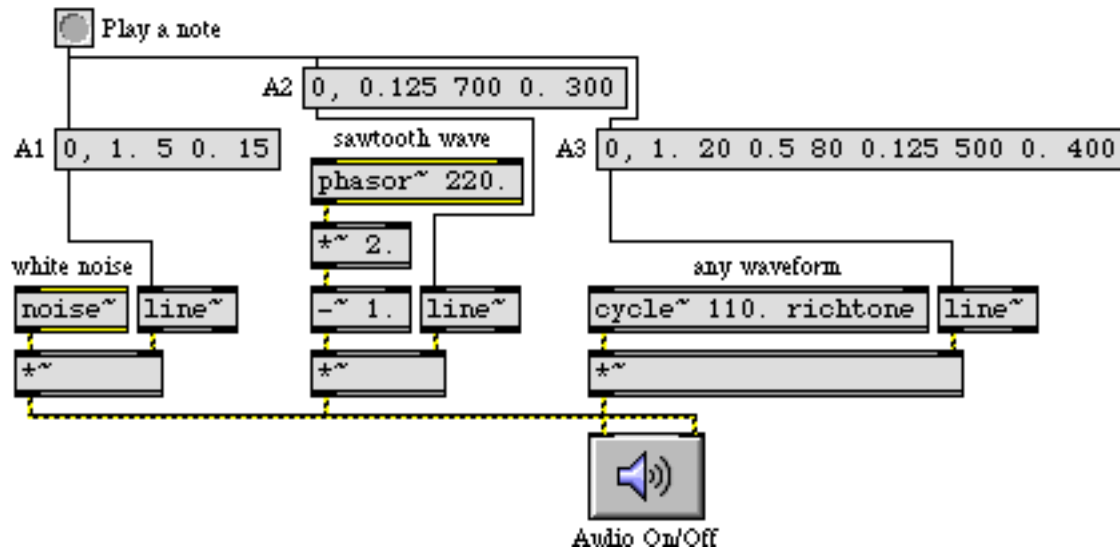
Any time two or more signals are connected to the same signal inlet, those signals are added together and their sum is used by the receiving object.



Multiple signals are added (mixed) in a signal inlet

Addition of digital signals is equivalent to unity gain mixing in analog audio. It is important to note that even if all your signals have amplitude less than or equal to 1, the sum of such signals can easily exceed 1. In MSP it's fine to have a signal with an amplitude that exceeds 1, but before sending the signal to **dac~** you must scale it (usually with a ***~** object) to keep its amplitude less than or equal to 1. A signal with amplitude greater than 1 will be distorted by **dac~**.

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



Three signals mixed to make a composite instrument sound

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

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

Summary

The **ezdac~** object is a button for switching the audio on and off. The **buffer~** object stores a sound. You can load an audio file into **buffer~** with a read message, which opens an Open Document dialog box for choosing the file to load in. If a **cycle~** object has a typed-in argument which gives it the same name as a **buffer~** object has, the **cycle~** will use 512 samples from that buffered sound as its waveform, instead of the default cosine wave.

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

Whenever you connect more than one signal to a given signal inlet, the receiving object adds those signals together and uses the sum as its input in that inlet. Exercise care when mixing (adding)

audio signals, to avoid distortion caused by sending a signal with amplitude greater than 1 to the DAC; signals must be kept in the range -1 to +1 when sent to **dac~** or **ezdac~**.

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

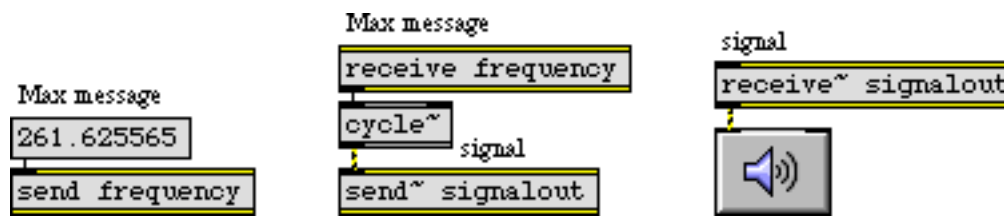
See Also

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

Remote signal connections: send~ and receive~

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

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



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

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

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

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

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

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

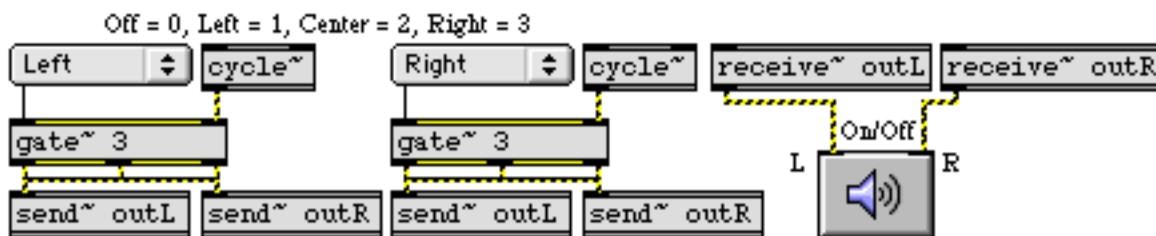
4. Changing the name of a **receive** object with a set message is a useful way of dynamically redirecting audio signals. Changing the name of **receive~**, however, does not redirect the signal until you turn audio off and back on again.

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

Routing a signal: gate~

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

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



gate~ *sends a signal to a chosen location*

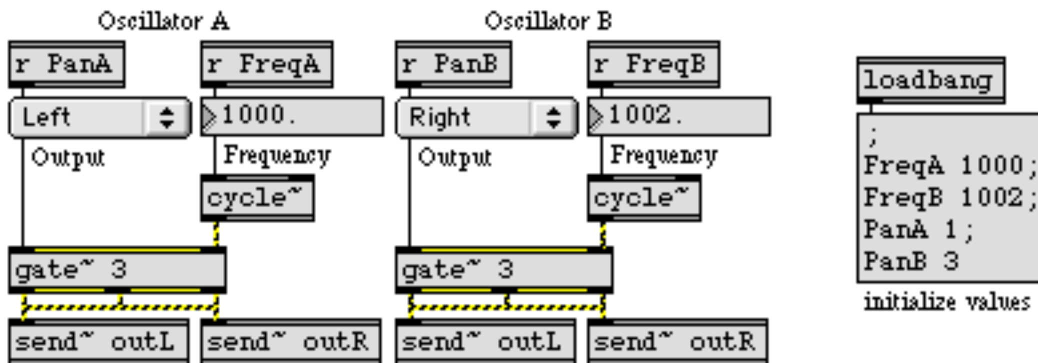
It is worth noting that changing the chosen outlet of a **gate~** while an audio signal is playing through it can cause an audible click because the signal shifts abruptly from one outlet to another. To avoid this, you should generally design your patch in such a way that the **gate~** object's outlet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

Wave interference

It's a fundamental physical fact that when we add together two sinusoidal waves with different frequencies we create *interference* between the two waves. Since they have different frequencies, they will usually not be exactly in phase with each other; so, at some times they will be sufficiently in phase that they add together constructively, but at other times they add together destructively, canceling each other out to some extent. They only arrive precisely in phase with each other at a rate equal to the difference in their frequencies. For example, a sinusoid at 1000 Hz and another at 1002 Hz come into phase exactly 2 times per second. In this case, they are sufficiently close in frequency that we don't hear them as two separate tones. Instead, we hear their recurring pattern of constructive and destructive interference as *beats* occurring at a sub-audio rate of 2 Hz, a rate known as the *difference frequency* or *beat frequency.* (Interestingly, we hear the two waves as a single tone with a sub-audio beat frequency of 2 Hz and an audio frequency of 1001 Hz.)

When the example patch is opened, a **loadbang** object sends initial frequency values to the **cycle~** objects—1000 Hz and 1002 Hz—so we expect that these two tones sounded together will cause a beat frequency of 2 Hz. It also sends initial values to the **gate~** objects (passing through the

umenus on the way) which will direct one tone to the left audio output and one to the right audio output.



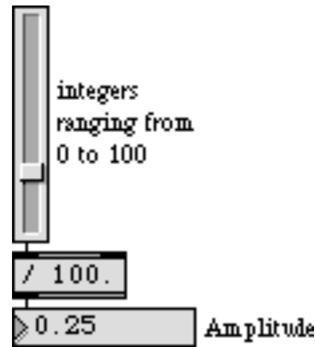
The two waves interfere at a rate of 2 Hz

- 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.

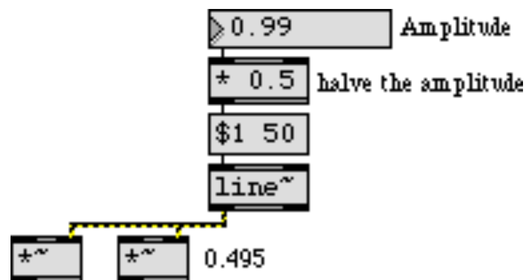
Amplitude and relative amplitude

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.



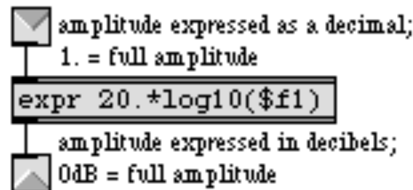
The amplitude is halved in case both oscillators are going to the same output channel

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

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

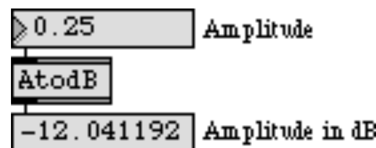
where A is the amplitude being measured and A_{ref} is a fixed reference amplitude.

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



The contents of the subpatch **AtodB**

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



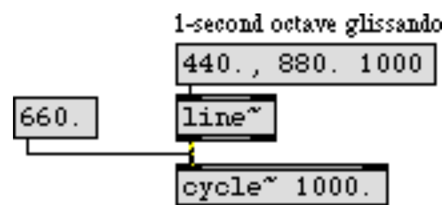
AtodB *reports amplitude in dB, relative to a reference amplitude of 1*

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

Constant signal value: sig~

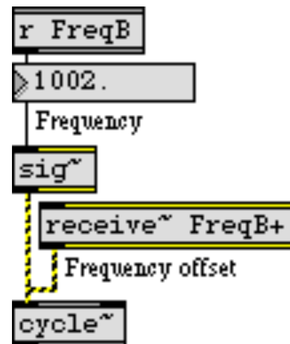
Most signal networks require some changing values (such as an amplitude envelope to vary the amplitude over time) and some constant values (such as a frequency value to keep an oscillator at a steady pitch). In general, one provides a constant value to an MSP object in the form of a float message, as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

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



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

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

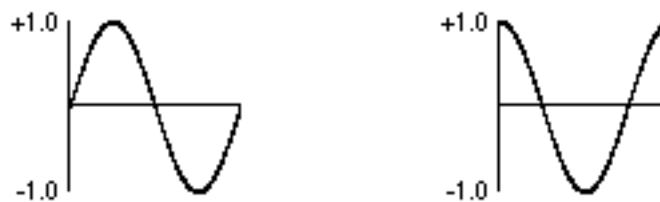


sig~ *converts a* float *to a* signal *so it can be combined with another* signal

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

Changing the phase of a waveform

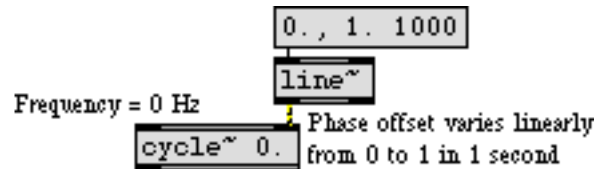
For the most part, the phase offset of an isolated audio wave doesn't have a substantial effect perceptually. For example, a sine wave in the audio range sounds exactly like a cosine wave, even though there is a theoretical phase difference of a quarter cycle. For that reason, we have not been concerned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



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

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

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

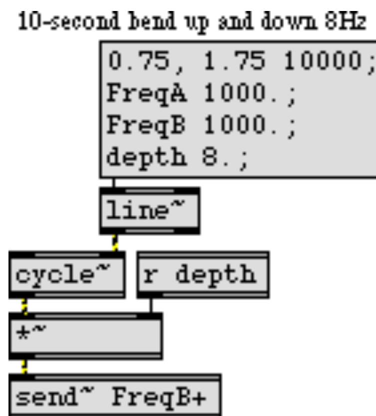


The resulting output is a cosine wave with a frequency of 1 Hz

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

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

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



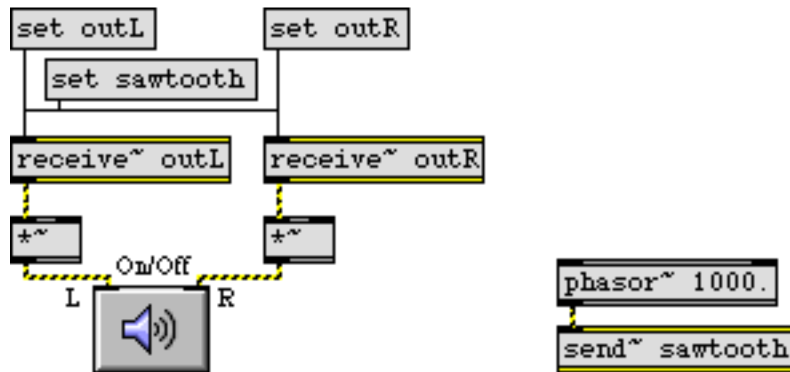
The result is one cycle of a sine wave. The sine wave is multiplied by a "depth" factor to scale its amplitude up to 8. This sub-audio sine wave, varying slowly from 0 up to 8, down to -8 and back to 0, is added to the frequency of Oscillator B. This causes the frequency of Oscillator B to fluctuate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

- Click on the **message** box in the lower-left part of the window, and notice how the beat frequency varies sinusoidally over the course of 10 seconds, from 0 Hz up to 8 Hz (as the fre-

quency of Oscillator B approaches 1008 Hz), back to 0 Hz, back up to 8 Hz (as the frequency of Oscillator B approaches 992 Hz), and back to 0 Hz.

Receiving a different signal

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



Giving **receive~** *a new name changes its input*

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

Summary

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

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

The **sig~** object converts a number to a constant signal; it receives a number in its inlet and sends out a signal of that value. This is useful for combining constant values with varying signals. Mixing together tones with slightly different frequencies creates interference between waves, which can create beats and other timbral effects.

See Also

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

Tutorial 5

Fundamentals: Turning signals on and off

Turning audio on and off selectively

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

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

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

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

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

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

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

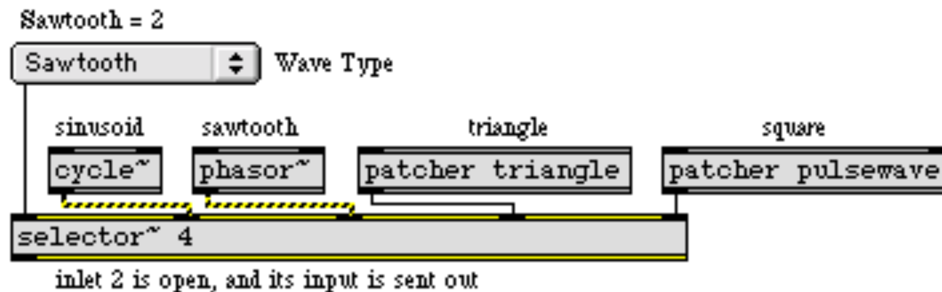
Sending the message startwindow—instead of start—to a **dac~**, **adc~**, **ezdac~**, or **ezadc~** object turns the audio on *only* in the Patcher window that contains that object, and in its subpatches. It turns audio off in all other Patchers. The startwindow message is very useful because it allows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

In some cases startwindow *is more appropriate than* start

Selecting one of several signals: selector~

In the previous chapter, we saw the **gate~** object used to route a signal to one of several possible destinations. Another useful object for routing signals is **selector~**, which is comparable to the Max object **switch**. Several different signals can be sent into **selector~**, but it will pass only one of

them—or none at all—out its outlet. The left inlet of **selector~** receives an int specifying which of the other inlets should be opened. Only the signal coming in the opened inlet gets passed on out the outlet.



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

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

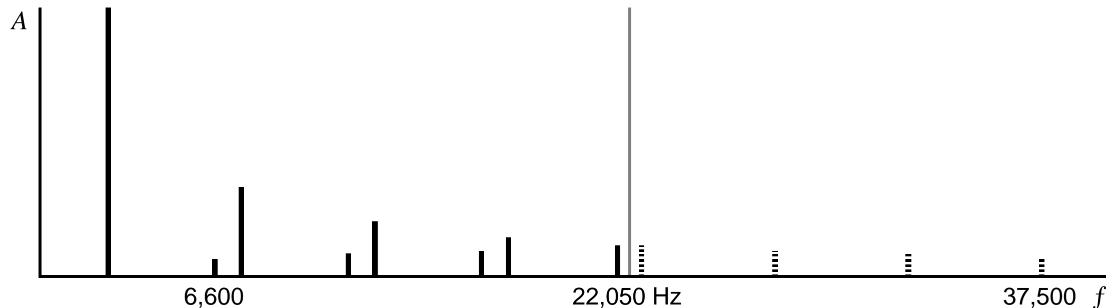
In the tutorial patch, **selector~** is used to choose one of four different classic synthesizer wave types: sine, sawtooth, triangle, or square. The **umenu** contains the names of the wave types, and sends the correct number to the control inlet of **selector~** to open the desired inlet.

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

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

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

amplitude is only about 24 dB less than that of the fundamental.) Other partials of the square wave will be similarly folded over.



Partials that exceed the Nyquist frequency are folded over

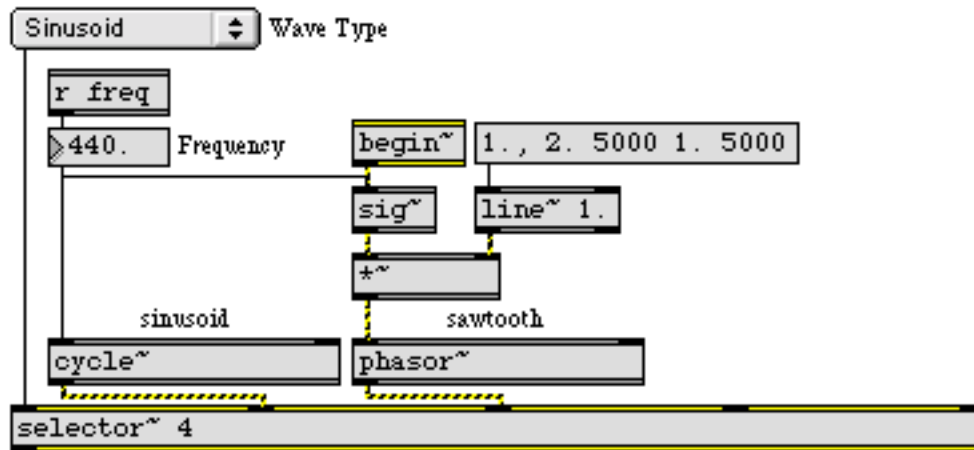
- Choose the square wave from the pop-up menu, and set the frequency to 2500 Hz. Turn audio on. Notice that some of the partials you hear are not harmonically related to the fundamental. If you move the frequency up further, the folded-over partials will go down by the same amount. Turn audio off.

Turning off part of a signal network: begin~

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

If the **begin~** object is placed at the beginning of a portion of a signal network, and that portion goes to the inlet of a **selector~** or **gate~** object, all calculations for that portion of the network will

be completely shut down when the **selector~** or **gate~** is ignoring that signal. This is illustrated by comparing the sinusoid and sawtooth signals in the tutorial patch.



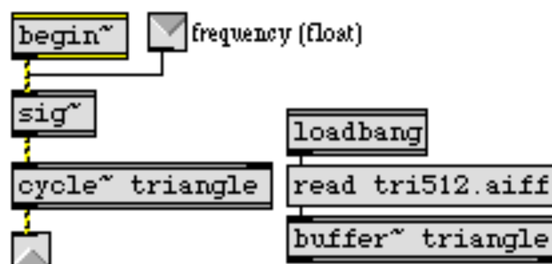
When the sinusoid is chosen, processing for the sawtooth is turned off entirely

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

- Choose "Sawtooth" from the pop-up menu, set the frequency back to 440 Hz, and turn audio on. Click on the **message** box above the **line~** object. The **line~** makes a glissando up an octave and back down over the course of ten seconds. Now click on it again, let the glissando get underway for two seconds, then use the pop-up menu to switch the **selector~** off. Wait five seconds, then switch back to the sawtooth. The glissando is on its way back down, indicating that the **line~** object continued to work even though the **sig~**, ***~**, and **phasor~** objects were shut down. When the glissando has finished, turn audio off.

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

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



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

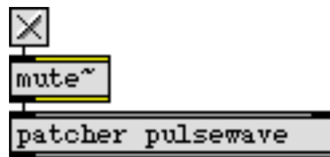
Here the **begin~** object is inside a subpatch, and the **selector~** is in the main patch, but the combination still works to stop audio processing in the objects that are between them. There is no MSP

object for making a triangle wave, so **cycle~** reads a single cycle of a triangle wave from an AIFF file loaded into a **buffer~**.

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

Disabling audio in a Patcher: mute~ and pcontrol

You have seen that the startwindow message to **dac~** turns audio on in a single Patcher and its subpatches, and turns audio off in all other patches. There are also a couple of ways to turn audio off in a specific subpatch, while leaving audio on elsewhere. One way is to connect a **mute~** object to the inlet of the subpatch you want to control.



Stopping audio processing in a specific subpatch

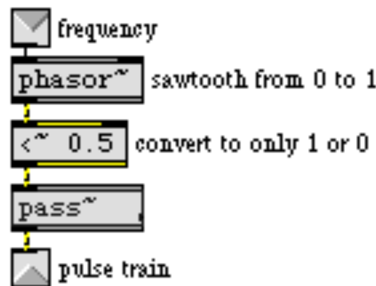
To mute a subpatch, connect a **mute~** object to the inlet of the subpatch, as shown. When **mute~** receives a non-zero int in its inlet, it stops audio processing for all MSP objects in the subpatch. Sending 0 to **mute~** object's inlet unmutes the subpatch.

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

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

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

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

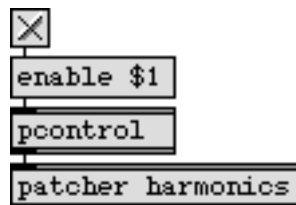


Output is 1 *for half the cycle, and* 0 *for half the cycle*

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

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

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

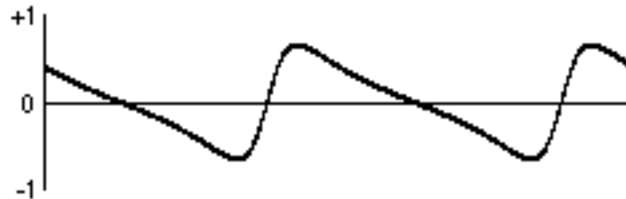


pcontrol *can disable and re-enable all MIDI and audio objects in a subpatch*

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

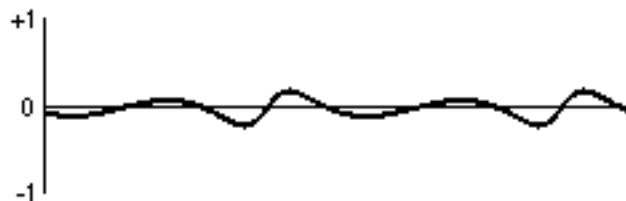
- Turn audio off, click on the **toggle** above the **patcher harmonics** object to enable it, then double-click on the **patcher harmonics** object to see its contents.

This subpatch combines 8 harmonically related sinusoids to create a complex tone in which the amplitude of each harmonic (harmonic number n) is proportional to $^1/_2{}^n$. Because the tones are harmonically related, their sum is a periodic wave at the fundamental frequency.



Wave produced by the **patcher harmonics** *subpatch*

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



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

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

Summary

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

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

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

See Also

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

Tutorial 6

Fundamentals: Review

Exercises in the fundamentals of MSP

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

Exercise 1

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

Here are a few hints:

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

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

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

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

Exercise 2

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

Hints:

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

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

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

Exercise 3

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

Hints:

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

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

Solution to Exercise 1

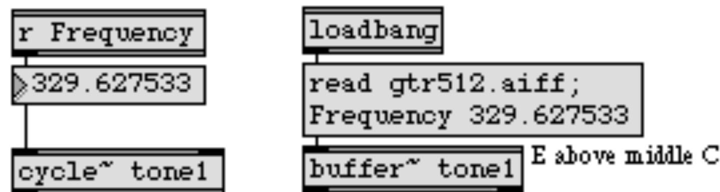
- Scroll the example Patcher window all the way to the right to see one possible solution to these exercises.

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

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

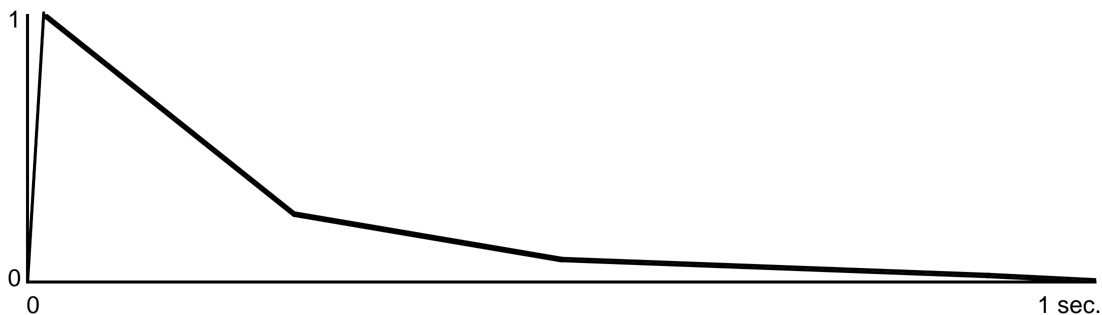
Note that there is a limit to the precision with which Max can represent decimal numbers. When you save your patch, Max may change float values slightly. In this case, you won't hear the difference.

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



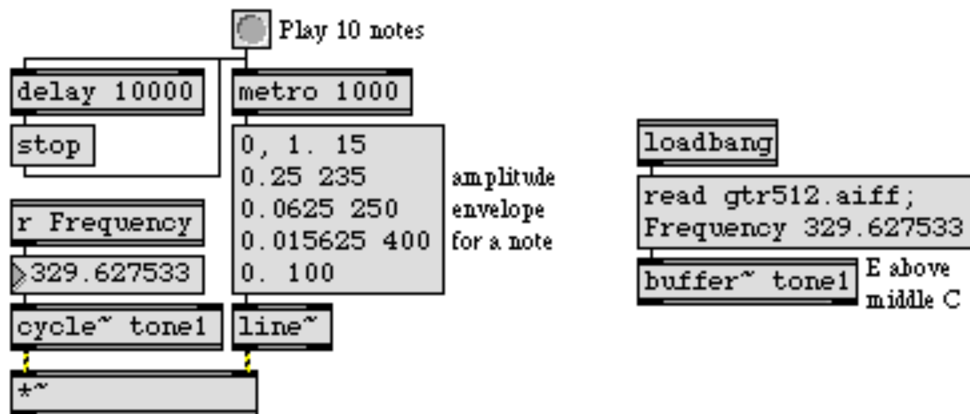
loadbang *is used to initialize the contents of* **buffer~** *and the frequency of* **cycle~**

Now that we have an oscillator producing the desired tone, we need to provide an amplitude envelope to shape a note. We chose the envelope shown below, composed of straight line segments. (See *Tutorial 3*.)



"Guitar-like" amplitude envelope

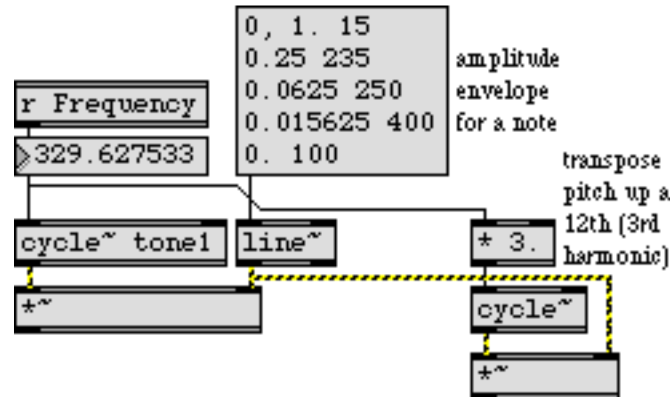
This amplitude envelope is imposed on the output of **cycle~** with a combination of **line~** and ***~**. A **metro** is used to trigger the envelope once per second, and the **metro** gets turned off after a 10-second delay.



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

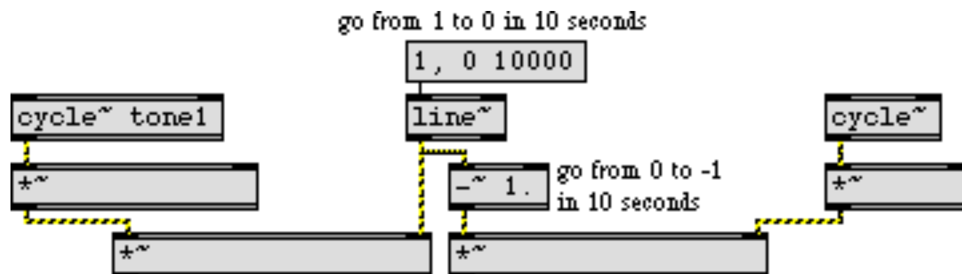
Solution to Exercise 2

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



Two oscillators with the same amplitude envelope and related frequencies

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



Linear crossfade of two tones

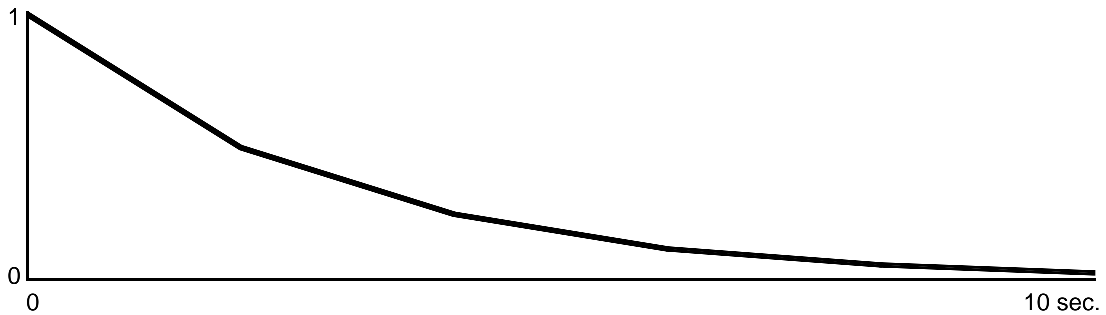
We used a little trick to economize. Rather than use a separate **line**~ object to fade the second tone from 0 to 1, we just subtract 1 from the output of the existing **line**~, which gives us a ramp from 0 to -1. Perceptually this will have the same effect.

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

Solution to Exercise 3

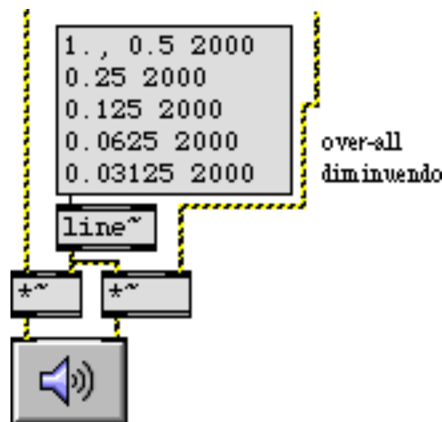
Finally, we need to use one more amplitude envelope to create a global *diminuendo*. The two tones go to yet another *~ object, controlled by another **line**~. As noted earlier, a straight line decrease in

amplitude will not give the perception of constant diminuendo in loudness. Therefore, we used five line segments to simulate a curve that decreases by half every two seconds.



Global amplitude envelope decreasing by half every two seconds

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



Both tones are scaled by the same global envelope

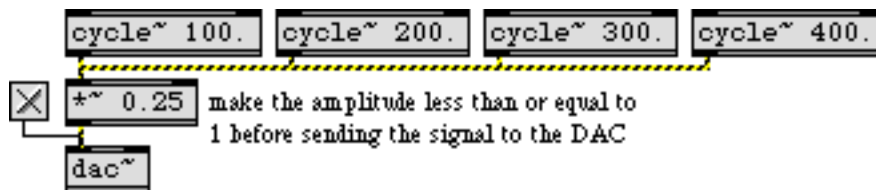
Tutorial 7

Synthesis: Additive synthesis

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

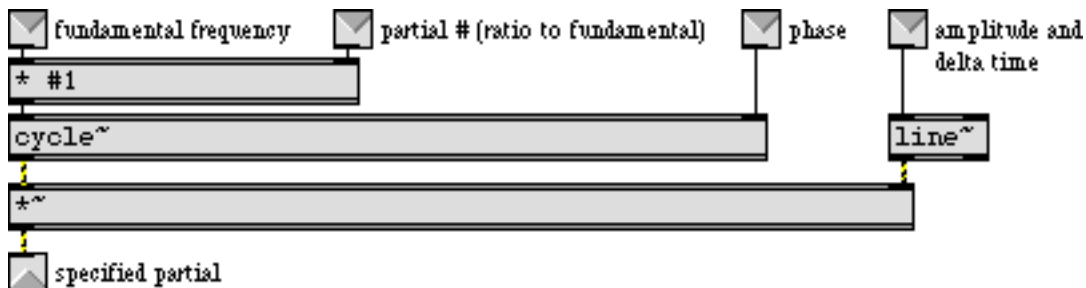
A sine wave contains energy at a single frequency. Since complex tones, by definition, are composed of energy at several (or many) different frequencies, one obvious way to synthesize complex tones is to use multiple sine wave oscillators and add them together.



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

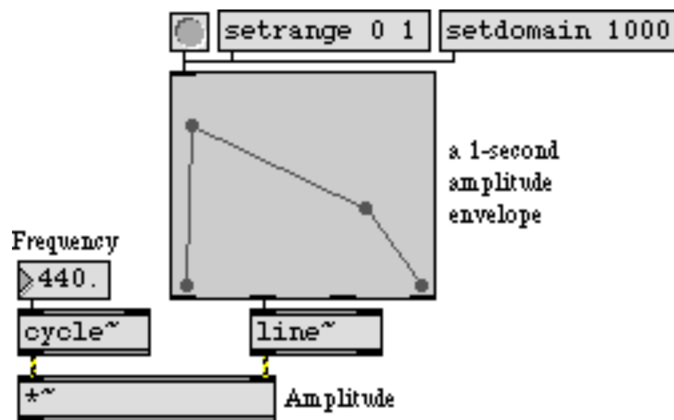
their fundamental frequency value (in the left inlet) from the same source, their relative frequencies will stay the same even when the value of the fundamental frequency changes.

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.

See Also

function	Graphical function breakpoint editor

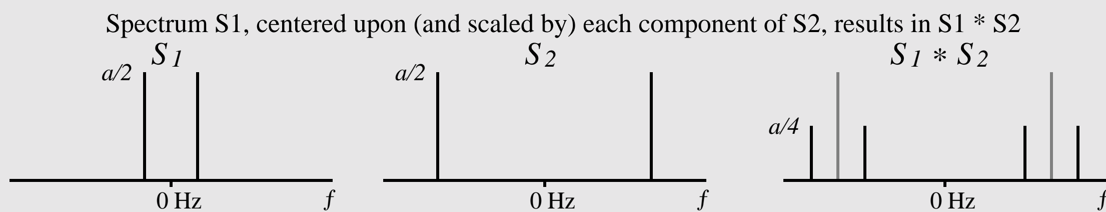
Tutorial 8

Synthesis: Tremolo and ring modulation

Multiplying signals

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

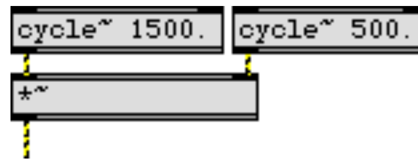
> **Technical detail:** Multiplication of waveforms in the time domain is equivalent to convolution of waveforms in the frequency domain. One way to understand convolution is as the superimposition of one spectrum on every frequency of another spectrum. Given two spectra S_1 and S_2, each of which contains 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

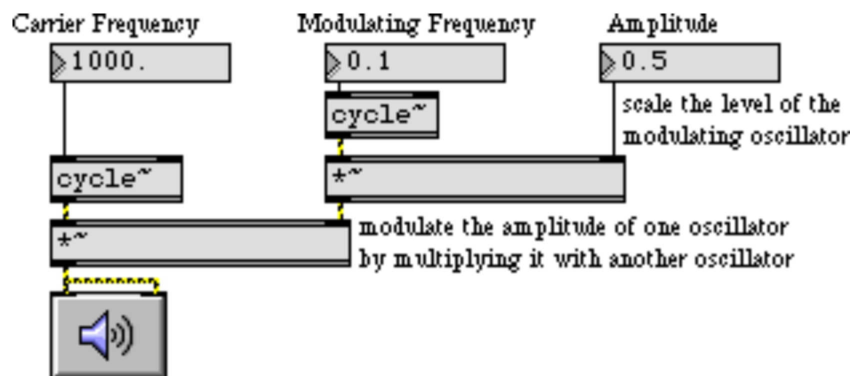
In our example patch, we multiply two sinusoidal tones. Ring modulation (multiplication) can be performed with any signals, and in fact the most sonically interesting uses of ring modulation

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



Simple multiplication of two cosine waves

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



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

Tremolo

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

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

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

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

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

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

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

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

Sidebands

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

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

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

- Listen to the remaining modulation rates.

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

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

Summary

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

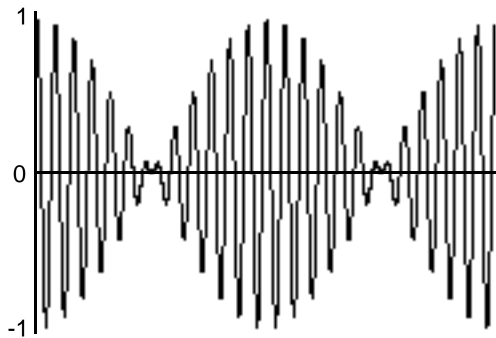
Multiplying an audio signal by a sub-audio signal results in regular fluctuations of amplitude known as *tremolo*. Multiplication of signals creates *sidebands*—additional frequencies not present in the original tones. Multiplying two sinusoidal tones produces energy at the sum and difference of the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

Tutorial 9

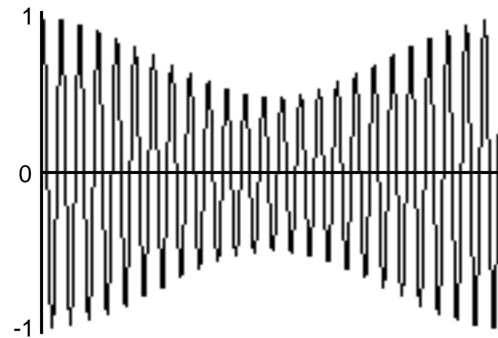
Synthesis: Amplitude modulation

Ring modulation and amplitude modulation

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



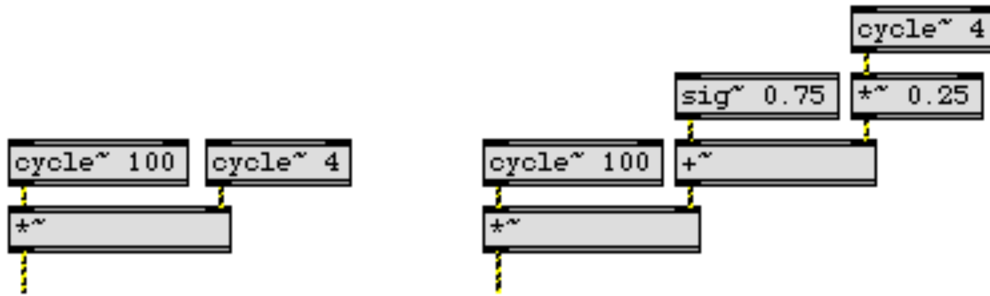
Ring modulation



Amplitude modulation

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

AM is perceived as a single cosine fluctuation per modulation period. The two MSP patches that made these examples are shown below.



Ring modulation *Amplitude modulation*

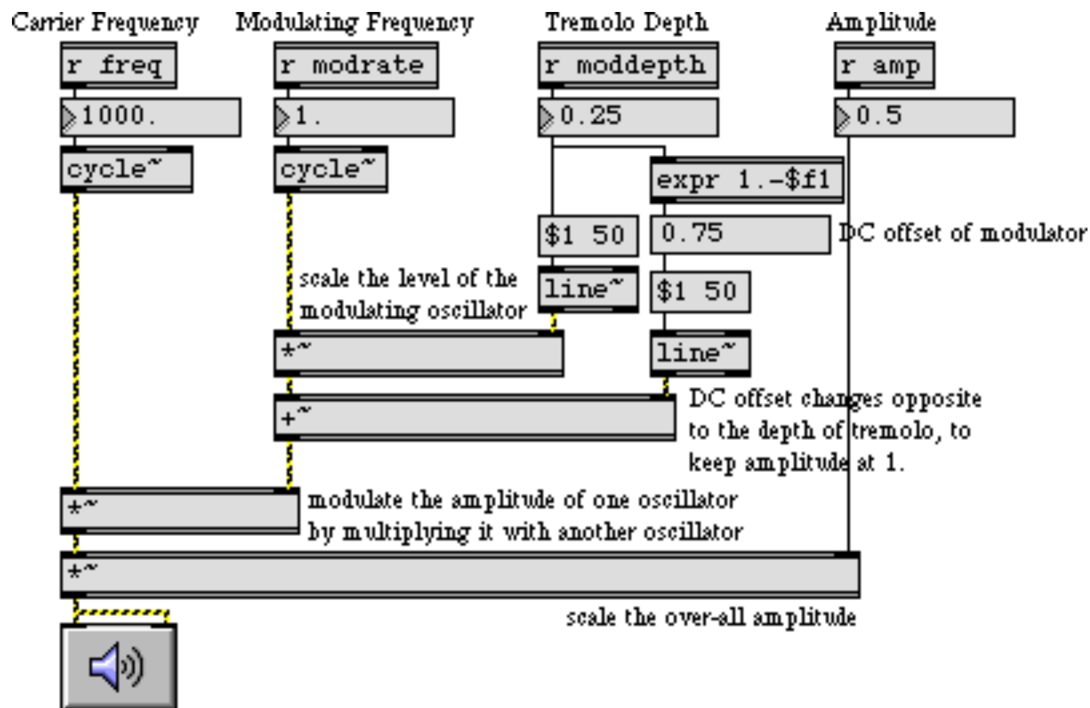
The difference in effect is due to the constant value of 0.75 in the AM patch, which is varied by a modulator of lesser amplitude. This constant value can be thought of as the carrier's amplitude, which is varied by the instantaneous amplitude of the modulator. The amplitude still varies according to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

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

Implementing AM in MSP

The tutorial patch is designed in such a way that the DC offset of the modulator is always 1 minus the amplitude of its sinusoidal variation. That way, the peak amplitude of the modulator is always

1, so the product of carrier and modulator is always 1. A separate *~ object is used to control the over-all amplitude of the sound.



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

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

Achieving different AM effects

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

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

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

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

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

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

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

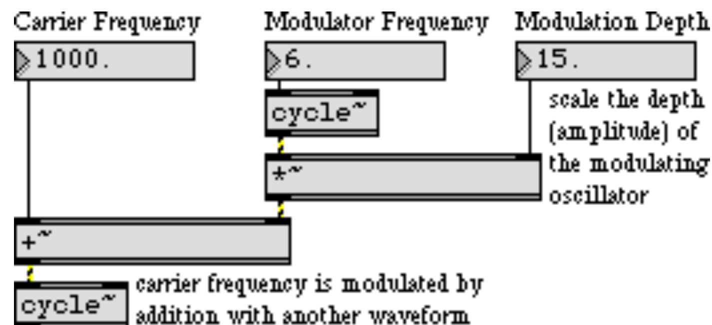
It is worth noting that any audio signals can be used as the carrier and modulator tones, and in fact many interesting results can be obtained by amplitude modulation with complex tones. (Tutorial 23 allows you to perform amplitude modulation on the sound coming into the computer.)

Summary

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

Basic FM in MSP

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



Simple frequency modulation

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

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

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

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

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

Note that 500 Hz is an octave below 1000 Hz, while 1500 Hz is only a perfect fifth above 1000 Hz. The interval between 500 Hz and 1500 Hz is thus a perfect 12th (as one would expect, given their 1:3 ratio). So you can see that a vibrato of equal frequency variation around a central frequency does not produce equal pitch variation above and below the central pitch. (In *Tutorial 17* we demonstrate how to make a vibrato that is equal in pitch up and down.)

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

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

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

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

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

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

Summary

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

Tutorial 11

Synthesis: Frequency modulation

Elements of FM synthesis

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

Modulating the frequency of one wave with another wave generates many sidebands, resulting in many more frequencies in the output sound than were present in the carrier and modulator waves themselves. As was mentioned briefly in the previous chapter, the frequencies of the sidebands are determined by the relationship between the carrier frequency (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.

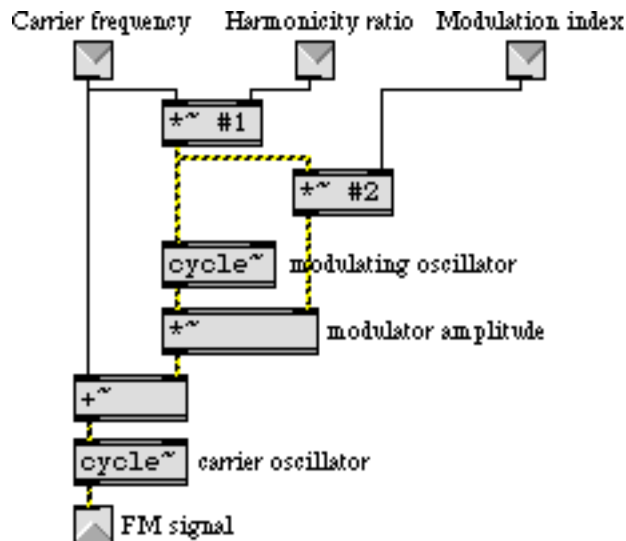
Technical detail: In John Chowning's article "Synthesis of Complex Audio Spectra by Means of Frequency Modulation" and in Curtis Roads' Computer Music Tutorial, they write about the ratio F_c/F_m. However, in F.R. Moore's Elements of Computer Music he defines the term harmonicity ratio as F_m/F_c. The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



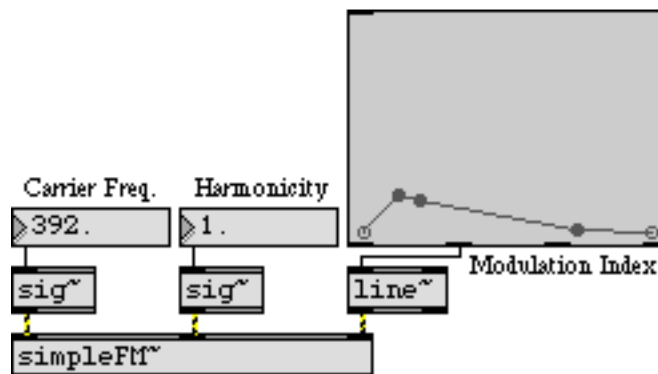
The **simpleFM~** *subpatch*

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.

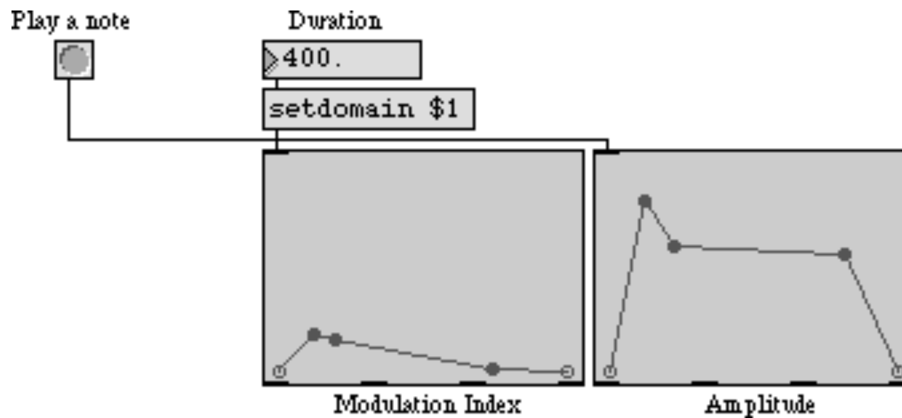
Producing different FM tones

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 slow attack and little decay. The pitch is G above middle C, and the harmonicity ratio is 1 for a fully harmonic spectrum.

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

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

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

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

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

As with amplitude modulation, frequency modulation can also be performed using complex tones. Sinusoids have traditionally been used most because they give the most predictable results, but many other interesting sounds can be obtained by using complex tones for the carrier and modulator signals.

Summary

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

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

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

Tutorial 12

Synthesis: Waveshaping

Using a stored wavetable

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

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

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

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

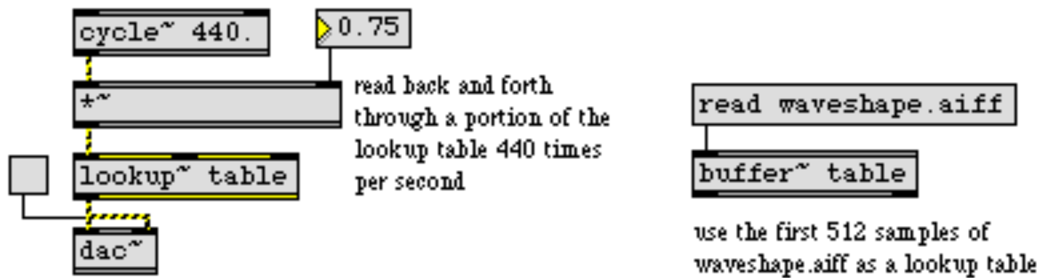
This type of synthesis allows you to use any waveform for **cycle~**, but the timbre is static and somewhat lifeless because the waveform is unchanging. This tutorial presents a new way to obtain dynamically changing timbres, using a technique known as *waveshaping*.

Table lookup: lookup~

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

The MSP object **lookup~** allows you to use samples stored in a **buffer~** as a lookup table which can be accessed by a signal in the range -1 to 1. By default, **lookup~** uses the first 512 samples in a **buffer~**, but you can type in arguments to specify any excerpt of the **buffer~** object's contents for

use as a lookup table. If 512 samples are used, input values ranging from -1 to 0 are mapped to the first 256 samples, and input values from 0 to 1 are mapped to the next 256 samples; **lookup~** interpolates between two stored values as necessary.



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

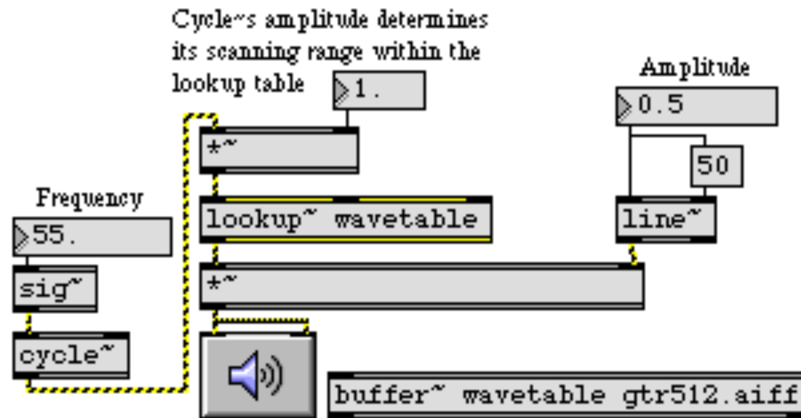
The most commonly used input signal for indexing the lookup table is a sine wave—it's a reasonable choice because it reads smoothly back and forth through the table—but any audio signal can be used as input to **lookup~**.

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

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

Varying timbre with waveshaping

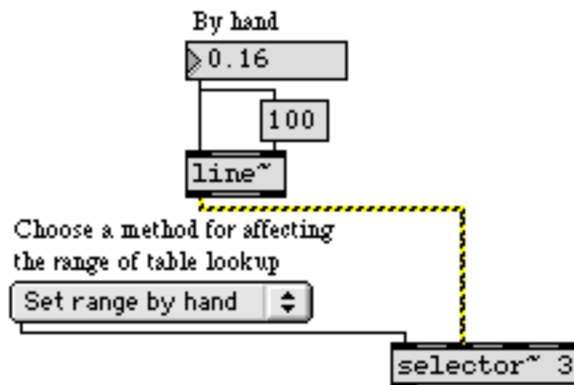
The waveshaping part of the tutorial patch is in the lower left portion of the Patcher window. It's very similar to the example shown above. The lookup table consists of the 512 samples in the **buffer~**, and it is read by a cosine wave from a **cycle~** object.



Lookup table used for waveshaping

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

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



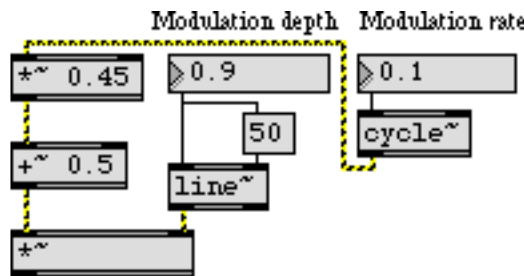
Set the amplitude of the input signal to change the timbre of the output

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

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

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

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



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

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

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

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

Summary

Waveshaping is the nonlinear distortion of a signal to create a new timbre. The sample values of the original signal are used to address a lookup table, and the corresponding value from the lookup table is sent out. The **lookup**~ object treats samples from a **buffer~** as such a lookup table, and uses the input range -1 to 1 to address those samples. A sine wave is commonly used as the input signal for waveshaping synthesis. The amplitude of the input signal determines how much of the lookup table gets used. As the amplitude of the input signal increases, more of the table gets used, and consequently more frequencies are generally introduced into the output. Thus, you can change the timbre of a waveshaped signal dynamically by continuously altering the amplitude of the input signal, using a control function or a modulating signal.

See Also

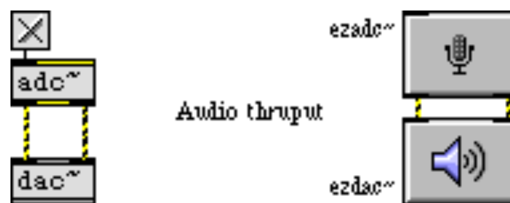
buffer~	Store audio samples
cycle~	Table lookup oscillator
lookup~	Transfer function lookup table

Tutorial 13

Sampling: Recording and playback

Sound input: adc~

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

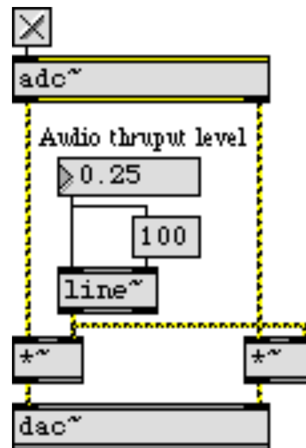


adc~ *and* **ezadc~** *get sound from the audio input jacks and send it out as a signal*

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

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

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

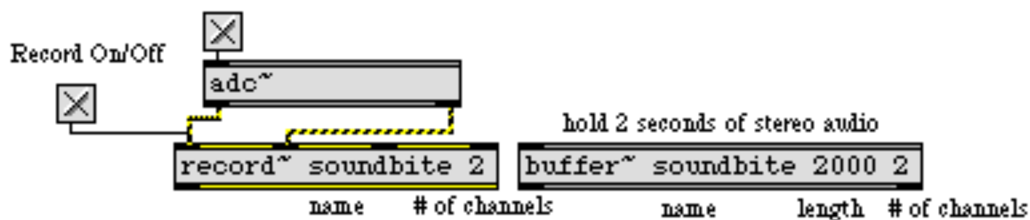


Adjust the audio throughput to a comfortable listening level

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

Recording a sound: record~

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

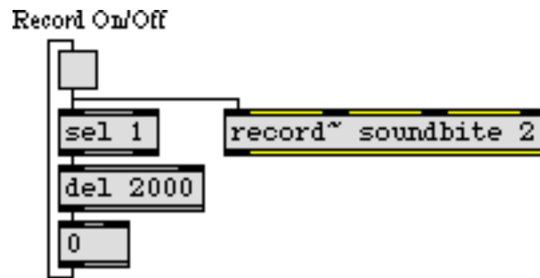


Record two seconds of stereo sound into the **buffer~** *named* soundbite

When **record~** receives a non-zero int in its left inlet, it begins recording the signals connected to its record inlets; 0 stops the recording. You can specify recording start and end points within the **buffer~** by sending numbers in the two right inlets of **record~**. If you don't specify start and end points, recording will fill the entire **buffer~**. Notice that the length of the recording is limited by the length of the **buffer~**. If this were not the case, there would be the risk that **record~** might be left on accidentally and fill the entire application memory.

In the tutorial patch, **record~** will stop recording after 2 seconds (2000 ms). We have included a delayed bang to turn off the **toggle** after two seconds, but this is just to make the **toggle** accurately

display the state of **record~**. It is not necessary to stop **record~** explicitly, because it will stop automatically when it reaches its end point or the end of the **buffer~**.



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

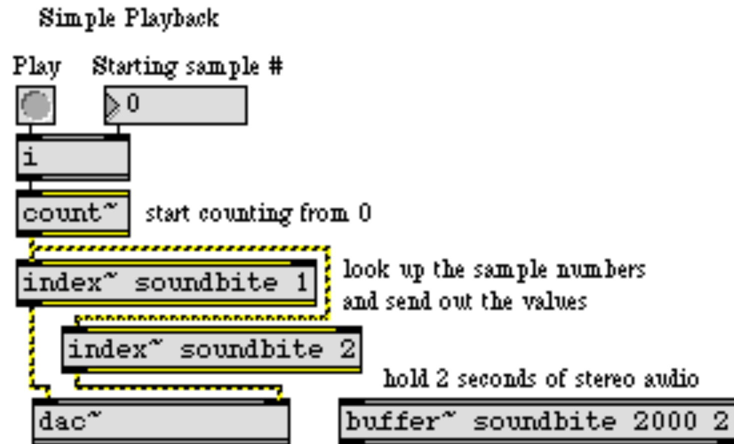
- Make sure that you have sound coming into the computer, then click on the **toggle** to record two seconds of the incoming sound. If you want to, you can double-click on the **buffer~** afterward to see the recorded signal.

Reading through a buffer~: index~

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

The **index~** object receives a signal as its input, which represents a sample number. It looks up that sample in its associated **buffer~**, and sends the value of that sample out its outlet as a signal. The **count~** object just sends out a signal value that increases by one with each sample. So, if you send the output of **count~**—a steady stream of increasing numbers—to the input of **index~**—which

will treat them as sample numbers—**index~** will read straight through the **buffer~**, playing it back at the current sampling rate.



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

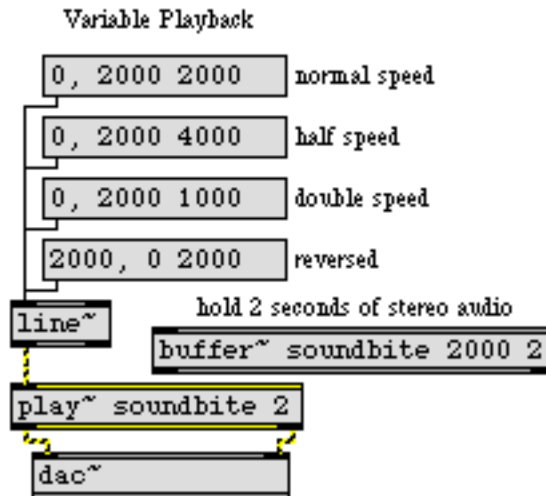
- Click on the **button** marked "Play" to play the sound in the **buffer~**. You can change the starting sample number by sending a different starting number into **count~**.

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

Variable speed playback: play~

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

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



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

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

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

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

Summary

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

See Also

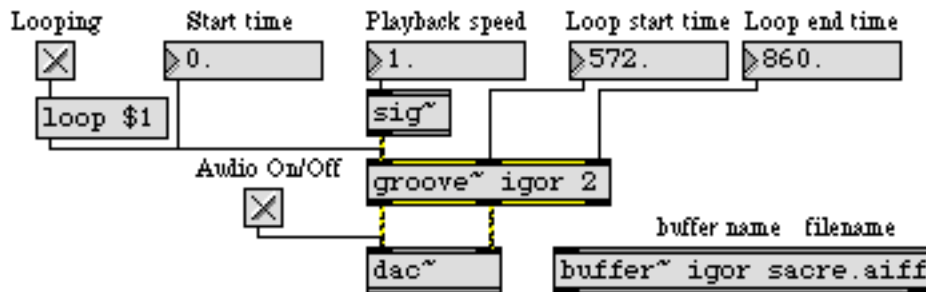
adc~	Audio input and on/off
ezadc~	Audio on/off; analog-to-digital converter
index~	Sample playback without interpolation
play~	Position-based sample playback
record~	Record sound into a buffer

Tutorial 14

Sampling: Playback with loops

Playing samples with groove~

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



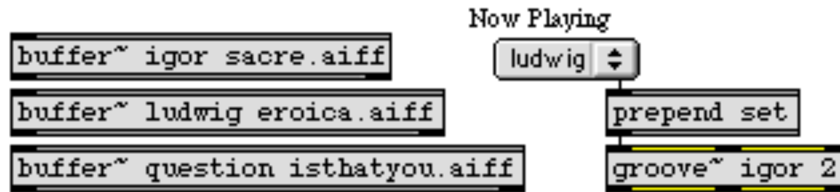
A standard configuration for the use of **groove~**

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

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

In the tutorial patch, three different **buffer~** objects are loaded with AIFF files so that a single **groove~** object can switch between various samples instantly. The message set, followed by the name of a **buffer~**, refers **groove~** to that new **buffer~** immediately. (If **groove~** always referred to

the same **buffer~**, and we used read messages to change the contents of the **buffer~**, some time would be needed to open and load each new file.)

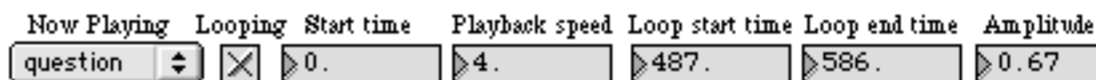


Refer **groove~** *to a different* **buffer~** *with a* set *message*

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

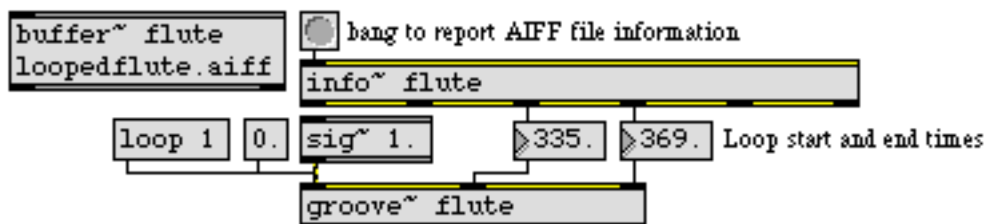
The first preset just functions as an "Off" button. The next three presets play the three **buffer~** objects at normal speed without looping. The rest of the presets demonstrate a variety of sound possibilities using different playback speeds on different excerpts of the buffered files, with or without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

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



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

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



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

Summary

The **groove~** object is the most versatile way to play sound from a **buffer~**. You can specify the **buffer~** to read, the starting point, the playback speed (either forward or backward), and starting

and ending points for a repeating loop within the sample. If the **buffer~** contains an AIFF file that has its own pre-established loop points, you can use the **info~** object to get those loop times and send them to **groove~**. The playback speed of **groove~** is determined by the value of the signal coming in its left inlet. You can set the current buffer position of **groove~** by sending a float time value in the left inlet.

See Also

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

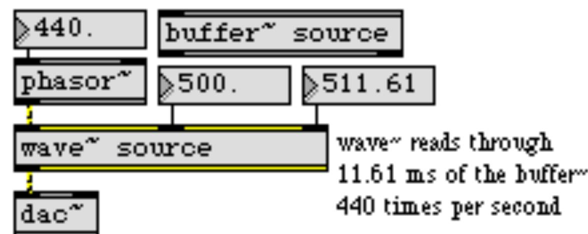
Tutorial 15

Sampling: Variable-length wavetable

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

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

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



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

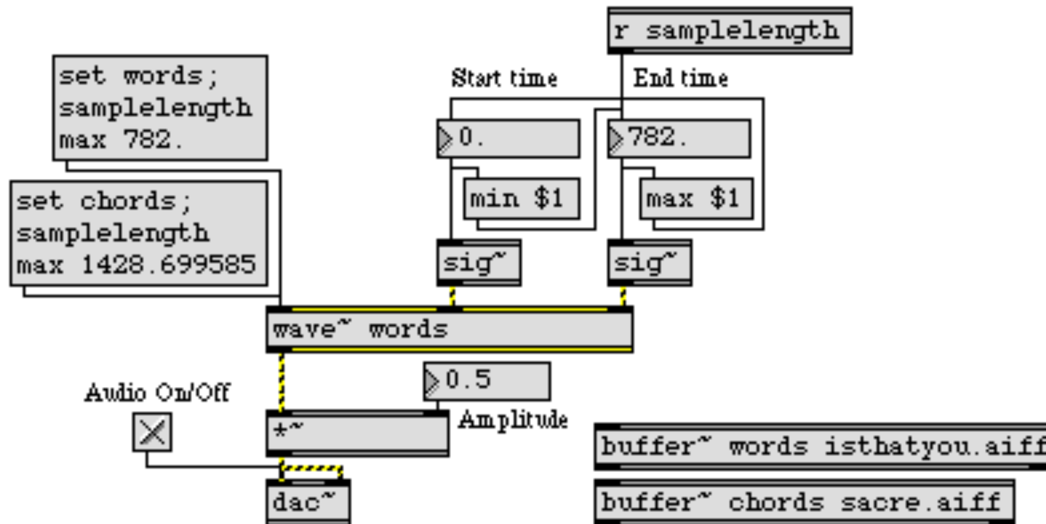
In a standard implementation of wavetable synthesis, the wavetable (512 samples in the case of **cycle~**, or a section of any length in the case of **wave~**) would be one single cycle of a waveform, and the frequency of the **cycle~** object (or the **phasor~** driving the **wave~**) would determine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

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

Synthesis with a segment of sampled sound

The tutorial patch is designed to let you try three different ways of driving **wave~**: with a repeating ramp signal (**phasor~**), a sinusoid (**cycle~**), or a single ramp (**line~**). The bottom part of the Patcher window is devoted to the basic implementation of **wave~**, and the upper part of the win-

dow contains the three methods of reading through the wavetable. First, let's look at the bottom half of the window.

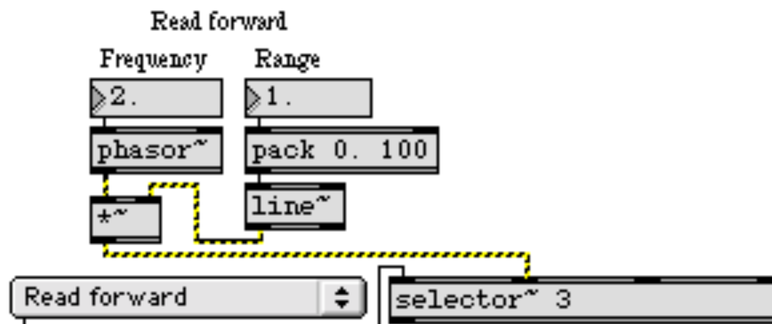


wave~ *can use an excerpt of any length from either* **buffer~** *as its wavetable*

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

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

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



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

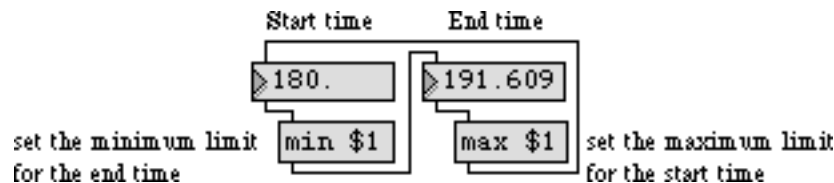
- Set the **number box** marked "Range" to 1. This sets the amplitude of the **phasor~**, so it effectively determines what fraction of the wavetable will be used. Set the **number box** marked "Frequency" to 2. The **phasor~** now goes from 0 to 1 two times per second, so you should hear **wave~** reading through the **buffer~** every half second.

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

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

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

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



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

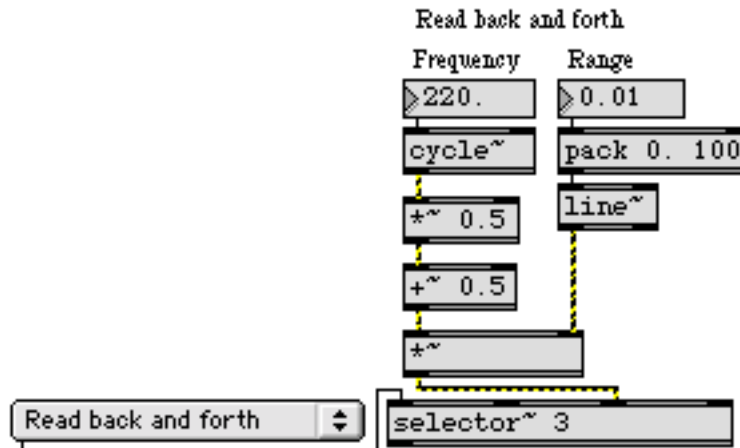
- With this new **buffer~**, experiment further by reading different length segments of the **buffer~** at various rates.

Using wave~ as a transfer function

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

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

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



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

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

Play the segment as a note

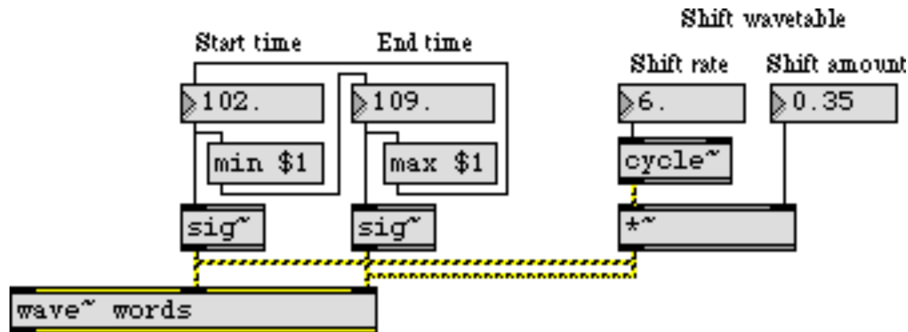
Because **wave~** accepts any signal input in the rage 0 to 1, you can read through the wavetable just once by sending **wave~** a ramp signal from 0 to 1 (or backward, from 1 to 0). Other objects such as **play~** and **groove~** are better suited for this purpose, but it is nevertheless possible with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

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

Changing the wavetable dynamically

The **cycle~** object in the right part of the Patcher window is used to add a sinusoidal position change to the wavetable. As the cosine wave rises and falls, the start and end times of the wavetable increase and decrease. As a result, the wavetable is constantly shifting its position in the **buffer~**, in a sinusoidally varying manner. Sonically this produces a unique sort of vibrato, not of fundamen-

tal frequency but of timbre. The wavetable length and the rate at which it is being read stay the same, but the wavetable's contents are continually changing.



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

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

Summary

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

See Also

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

Tutorial 16

Sampling: Record and play audio files

Playing from memory vs. playing from disk

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

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

Record audio files: sfrecord~

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

Recording audio into an audio file on disk

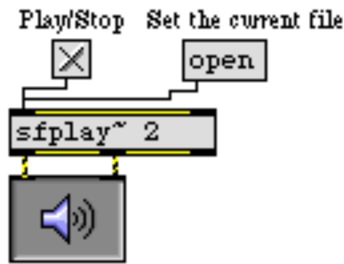
sfrecord~ records to disk whatever signal data it receives in its inlets. The signal data can come directly from an **adc~** or **ezadc~** object, or from any other MSP object.

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

Play audio files: sfplay~

In the left part of the Patcher window there is a patch for playing audio files. The basic usage of **sfplay~** requires only a few objects, as shown in the following example. To play a file, you just have

to open it and start **sfplay~**. The audio output of **sfplay~** can be sent directly to **dac~** or **ezdac~**, and/or anywhere else in MSP.



Simple implementation of audio file playback

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

Play excerpts on cue

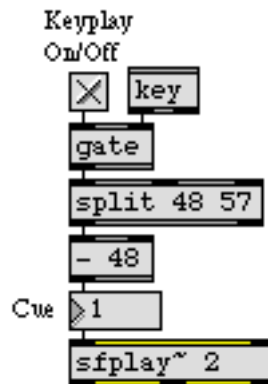
Because **sfplay~** does not need to load an entire audio file into memory, you can actually have many files open in the same **sfplay~** object, and play any of them (or any portion of them) on cue. The most recently opened file is considered by **sfplay~** to be the "current" file, and that is the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

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

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

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

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



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

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

Try different file excerpts

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

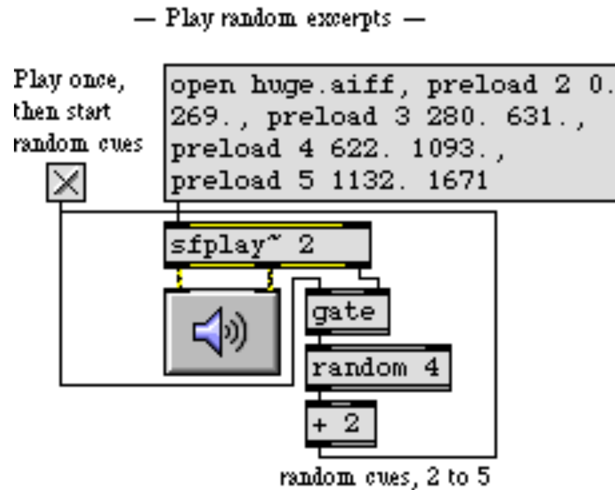
- Open your own audio file again (or any other audio file) to make it the current file. In the right portion of this patch, enter an end time for the seek message. The excerpt you have specified will begin playing. Try different start and end times.

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

Trigger an event at the end of a file

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

bang out the right outlet. In this example patch, the bang is used to trigger the next (randomly chosen) cue, so **sfplay~** effectively restarts itself when each cue is done.



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

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

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

Summary

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

See Also

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

Tutorial 17

Sampling: Review

A sampling exercise

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

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

Hints

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

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

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

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

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

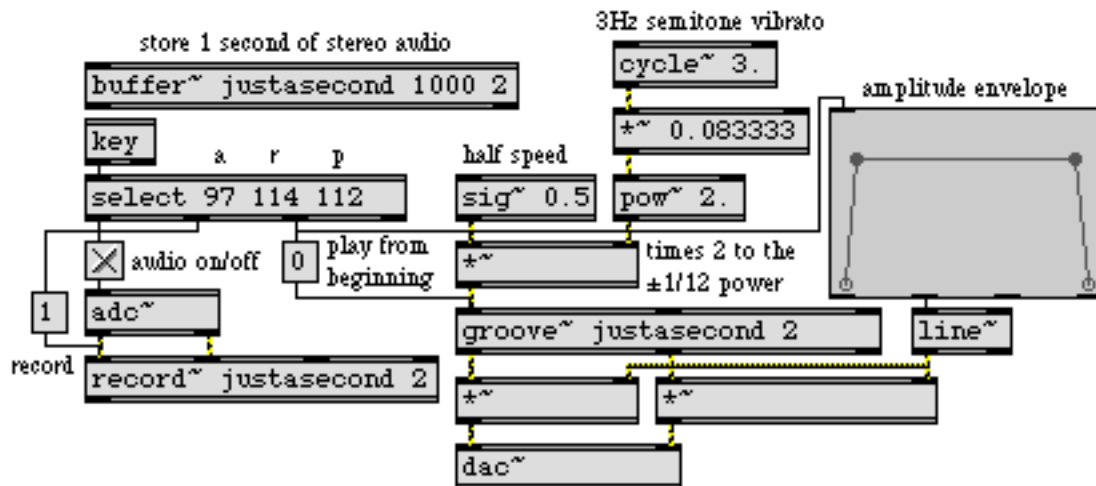
Use a sinusoidal wave from a **cycle~** object to apply vibrato to the sample. The frequency of the **cycle~** will determine the *rate* of the vibrato, and the amplitude of the sinusoid will determine the *depth* of vibrato. Therefore, you will need to scale the **cycle~** object's amplitude with a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

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

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

Solution

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



Solution to the exercise for recording and playing an audio sample

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

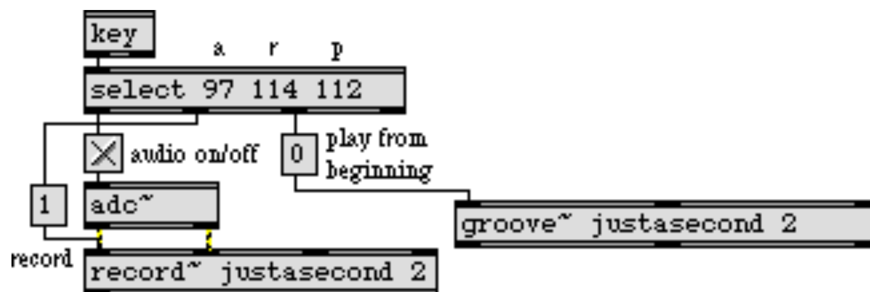
store 1 second of stereo audio

buffer~ justasecond 1000 2

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

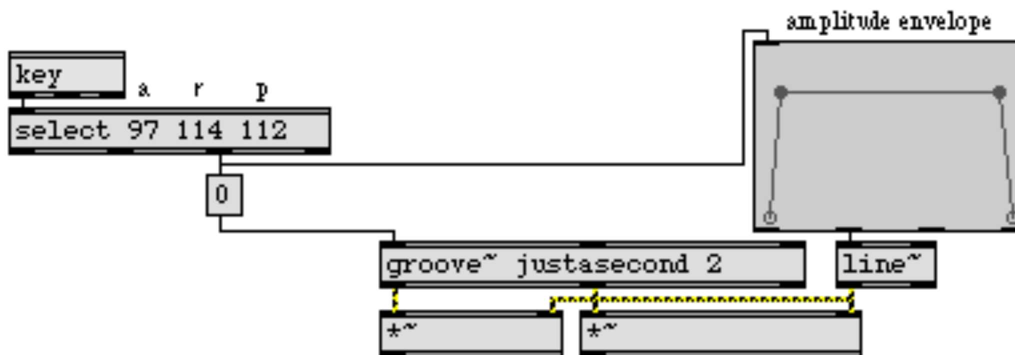
Since the memory allocated in the **buffer~** is limited to one second, there is no need to tell the **record~** object to stop when you record into the **buffer~**. It stops when it reaches the end of the **buffer~**.

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



Keystrokes are detected and used to send messages to MSP objects

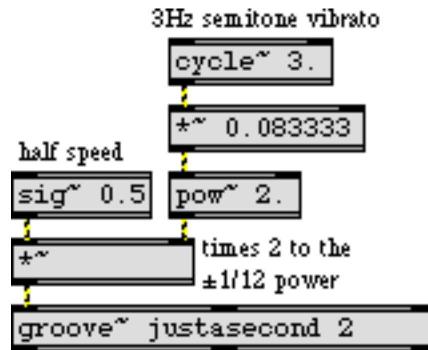
The keystroke *p* is also used to trigger the amplitude envelope at the same time as the sample is played. This envelope is used to scale the output of **groove~**.



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

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

power function in **pow~** (2 to the power of the input), and the result is used as the factor by which to multiply the playback speed.



Play at half speed, ± one semitone

Tutorial 18

MIDI control: Mapping MIDI to MSP

MIDI range vs. MSP range

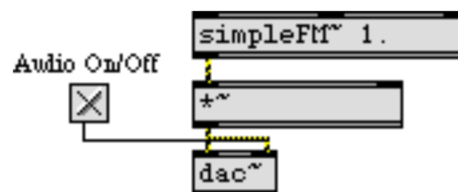
One of the greatest assets of MSP is the ease with which one can combine MIDI and digital signal processing. The great variety of available MIDI controllers means that you have many choices for the instrument you want to use to control sounds in MSP. Because Max is already a well developed environment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

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

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

Controlling synthesis parameters with MIDI

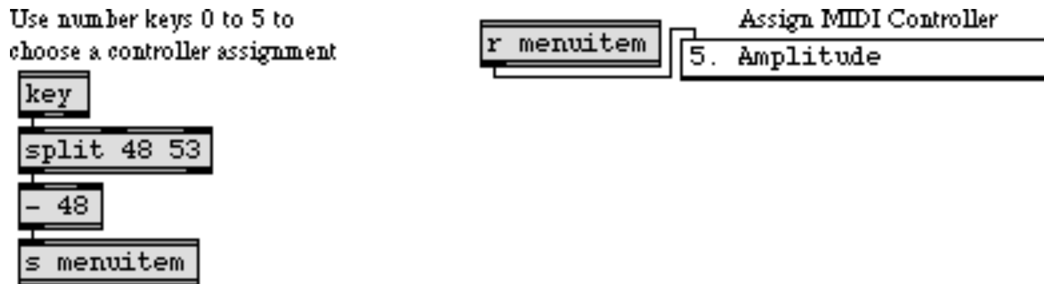
In this tutorial patch, we use MIDI continuous controller messages to control several different parameters in an FM synthesis patch. The synthesis is performed in MSP by the subpatch **simpleFM~** which was introduced in *Tutorial 11*, and we map MIDI controller 1 (the mod wheel) to affect, in turn,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



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

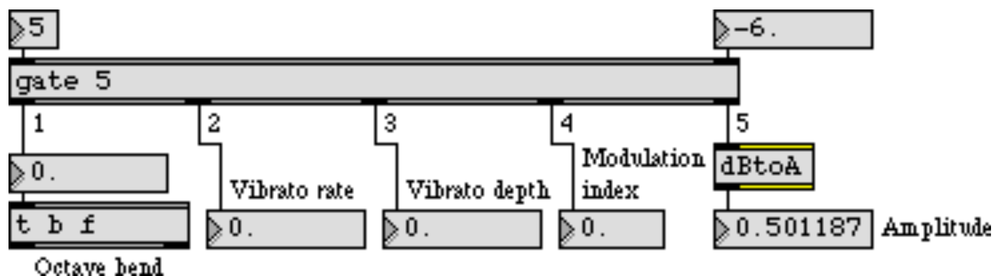
If we were designing a real performance instrument, we would probably control each of these parameters with a separate type of MIDI message—controller 7 for amplitude, controller 1 for vibrato depth, pitchbend for pitch bend, and so on. In this patch, however, we use the mod wheel controller for everything,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

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



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

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



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

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

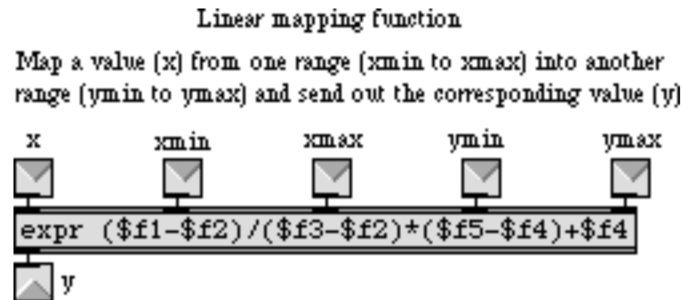
Linear mapping

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

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

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*

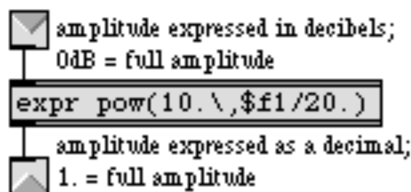
Once we have scaled the range of control values with **map**, some additional mapping may be necessary to suit various signal processing purposes, as you will see.

Mapping MIDI to amplitude

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

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

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



The contents of the **dBtoA** *subpatch*

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

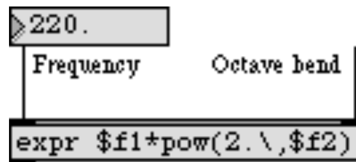
With this mapping, the amplitude changes by approximately a factor of 2 every time the controller value changes by 10. This permits the same amount of control at low amplitudes as at high amplitudes (which would not be the case with a straight linear mapping).

Mapping MIDI to frequency

Our perception of relative pitch is likewise multiplicative rather than additive with respect to frequency. In order for us to hear equal spacings of pitch, the frequency must change in equal powers of 2. (See the discussions of pitch-to-frequency conversion in *Tutorial 17* and *Tutorial 19*.)

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

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



Octave bend factor ranges from 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

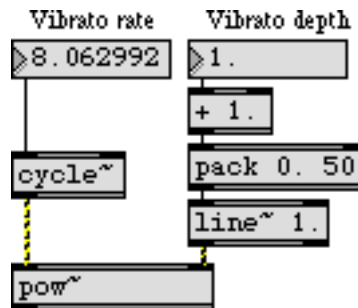
Mapping the MIDI controller to the modulation index of the FM instrument is much simpler, because a linear control is what's called for. Once the controller values are converted by the **map** subpatch, no further modification is needed. The mod wheel varies the modulation index from 0 (no modulation)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

Tutorial 19

MIDI control: Synthesizer

Implementing standard MIDI messages

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

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

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

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

Polyphony

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

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

The **poly** object assigns a voice number—1 or 2 in this case—to each incoming note message, and if more than two keys are held down at a time **poly** provides note-off messages for the earlier notes so that the later notes can be played. The voice number, key number, and velocity are packed together in a three-item list, and the **route** object uses the voice number to send the key number

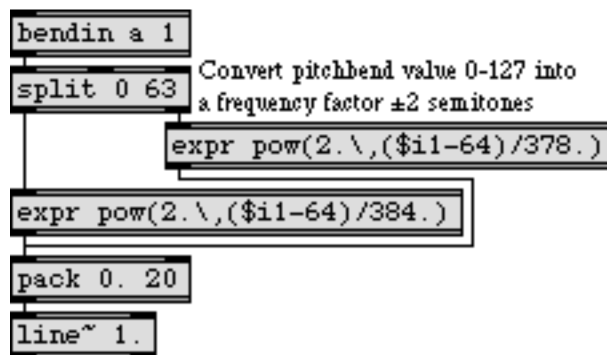
and velocity to one synthesizer "voice" or the other. If your computer is fast enough, of course, you can design synthesizers with many more voices. You can test the capability of your computer by adding more and more voices and observing the CPU Utilization in the DSP Status window.

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

Pitch bend

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

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



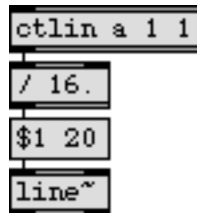
Downward bend is calculated slightly differently from upward bend

The downward bend values (0 to 63) are offset by -64 and divided by 384 so that the maximum downward bend (pitch bend value 0) produces an exponent of $^{-64}/_{384}$, which is equal to $^{-2}/_{12}$. The upward bend values (64 to 127) are offset by -64 and divided by 378 so that an upward bend produces an exponent ranging from 0 to $^{63}/_{378}$, which is equal to $^{2}/_{12}$. The **pack** and **line~** objects are used to make the frequency factor change gradually over 20 milliseconds, to avoid creating the effect of discrete stepwise changes in frequency.

Mod wheel

The mod wheel is used here to change the modulation index of our FM synthesis patch. The mapping is linear; we simply divide the MIDI controller values by 16 to map them into a range from 0

to (nearly) 8. The precise way this range is used will be seen when we look at the synthesis instrument itself.



Controller values mapped into the range 0 to 7.9375

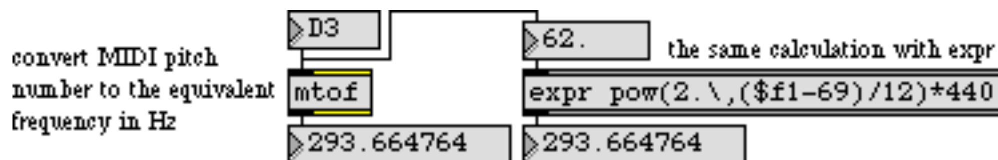
The FM synthesizer

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

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

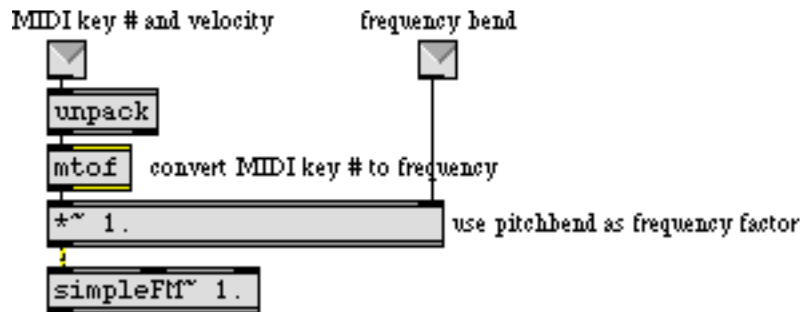
MIDI-to-frequency conversion

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



Calculate the frequency of a given pitch

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



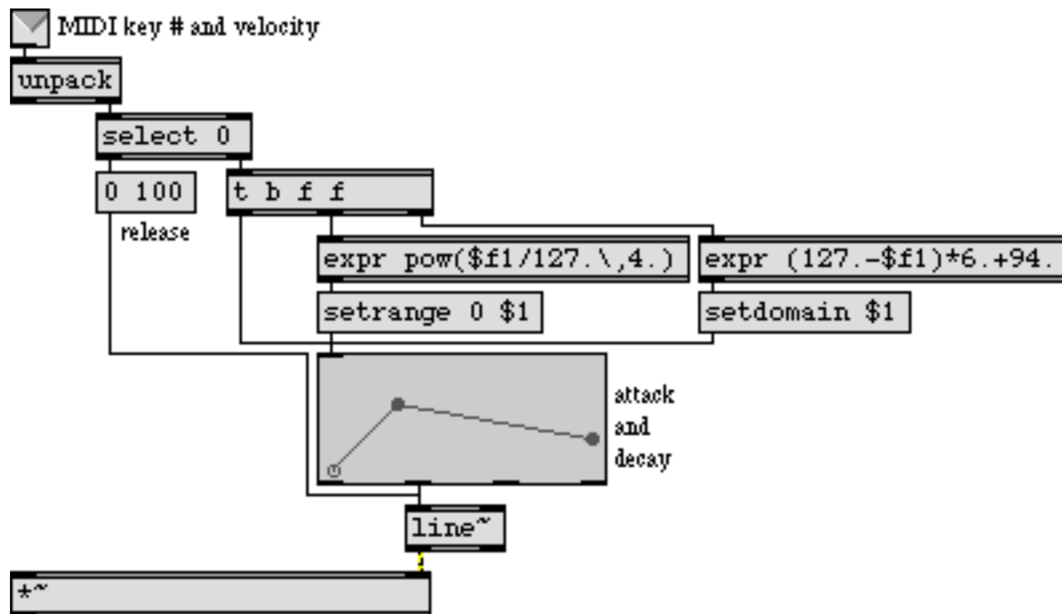
The frequency of the note calculated from key number and pitch bend data

Velocity control of amplitude envelope

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

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

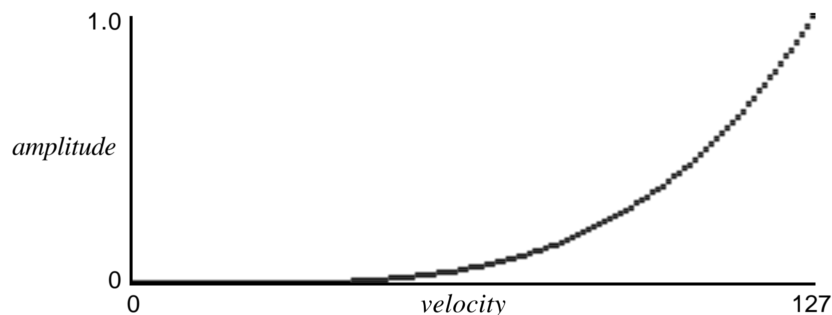
The first task is achieved easily with a **select** 0 object, so that note-on velocity triggers a **function** object to send the attack and decay shape, and note-off velocity returns the amplitude to 0, as shown in the following example.



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

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

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



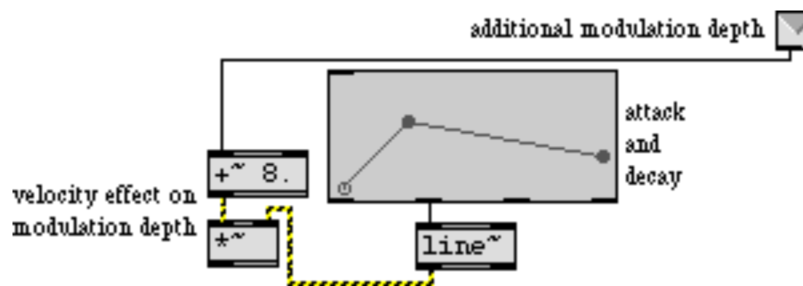
Velocity mapped to amplitude with an exponent of 4

If we used a straight linear mapping, MIDI velocities from 127 to 64 (the range in which most notes are played) would cover only about a 6 dB amplitude range. The exponential mapping

increases this to about 24 dB, so that change in the upper range of velocities produces a greater change in amplitude.

MIDI control of timbre

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



Envelope and mod wheel control modulation index

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

Summary

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

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

linearly. Velocity can also be used to alter the rate of the envelope, and/or other parameters such as modulation index.

An MSP object can only make one sound at a time, so if you want to play more than one simultaneous note via MIDI you will need to assign each note a voice number with **poly**, and route each voice to a different MSP object. In the next tutorial, we'll use the **poly** object to make polyphonic voice assignments for the simple case required for this tutorial. Tutorial 21 will introduce another way to manage polyphonic voice allocation in MSP—the **poly~** object.

See Also

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

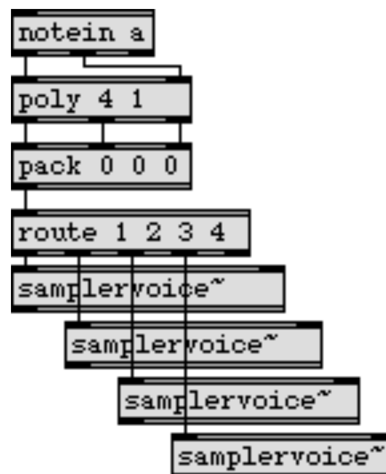
Tutorial 20

MIDI control: Sampler

Basic sampler features

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

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



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

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

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

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

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

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

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

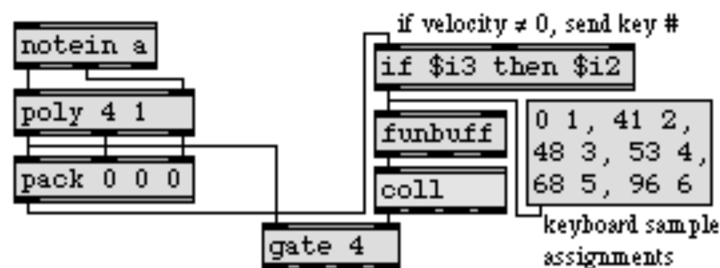
In this patch, the samples are all read into memory when the patch is first loaded.

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

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

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

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



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

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

1, 24 sample1 0 0 0;
2, 33 sample2 0 0 0;
3, 50 sample3 0.136054 373.106537 1;
4, 67 sample4 60.204079 70.476189 1;
5, 84 sample5 0 0 0;
6, 108 sample6 0 0 0;

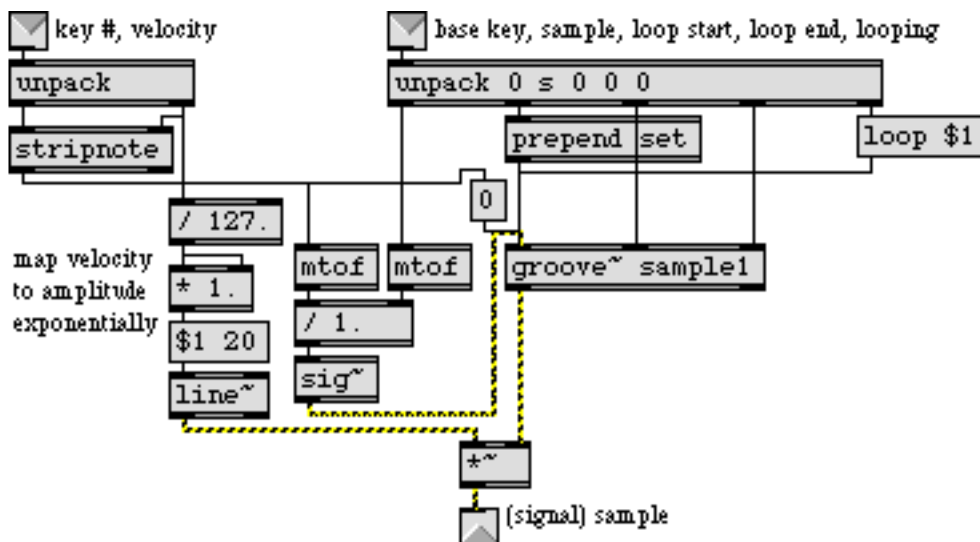
coll *contains sample information for each key region*

The key region number is used to index the information in **coll**. For example, whenever a key from 48 to 52 is pressed, **funbuff** sends out the number 3, and the information for key region 3 is recalled and sent to the appropriate **samplervoice~** subpatch. The data for each key region is: base key, **buffer~** name, loop start time, loop end time, and loop on/off flag.

The voice number from **poly** opens the correct outlet of **gate** so that the information from **coll** goes to the right subpatch.

Playing a sample: the samplervoice~ subpatch

- Close the **coll** window, and double-click on one of the **samplervoice~** subpatch objects to open its Patcher window.



The **samplervoice~** *subpatch*

You can see that the information from **coll** is unpacked in the subpatch and is sent to the proper places to prepare the **groove~** object for the note that is about to be played. This tells **groove~** what **buffer~** to read, what loop times to use, and whether looping should be on or off. Then, when the note information comes in the left inlet, the velocity is used to send an amplitude value to the ***~** object, and the note-on key number is used (along with the base key number received from the right inlet) to calculate the proper playback speed for **groove~** and to trigger **groove~** to begin playback from time 0.

MSP sample rate vs. audio file sample rate

- Close the subpatch window.

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

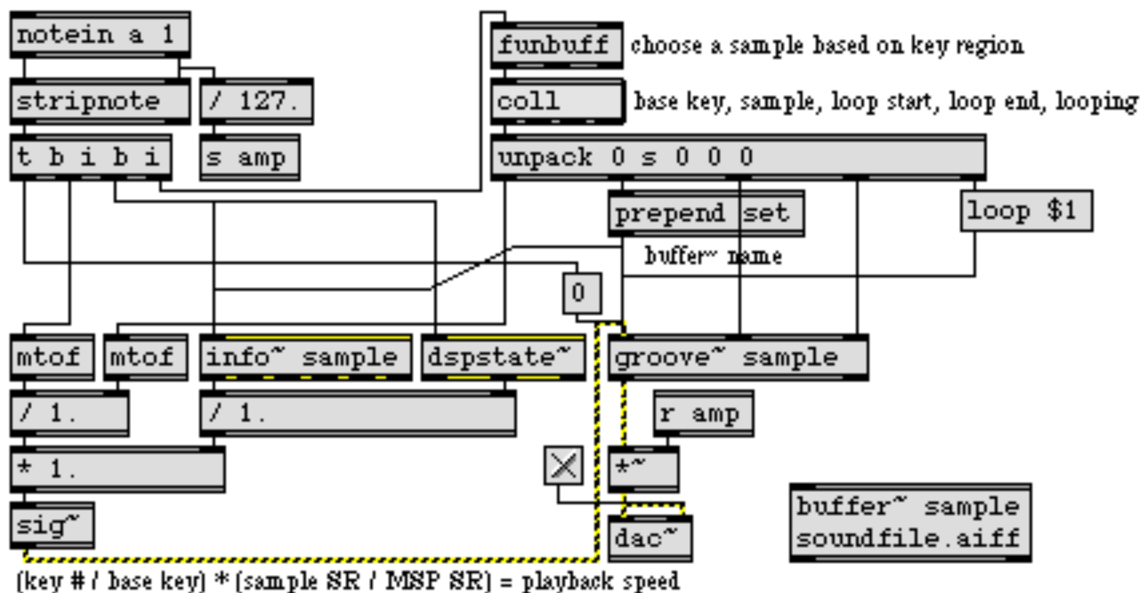
- Double-click on the **dac~** object to open the DSP Status window. Set the Sampling Rate to 22.050 kHz, then close the DSP Status window.

Note: Resetting the sampling rate may not be possible, depending on your hardware.

The difference between the sample rate of an audio file and the sample rate being used in MSP is a potential problem when playing samples. This method of resolving the difference suffices in this situation because the audio files are all at the same sample rate and because these samples are the only sounds we will be playing in MSP. In other situations, however, you're likely to want to play samples (perhaps with different sampling rates) combined with other sounds in MSP, and you'll want to use the optimum sampling rate.

For such situations, you would be best advised to use the ratio between the audio file sample rate and the MSP sample rate as an additional factor in determining the correct playback speed for **groove~**. For example, if the sample rate of the audio file is half the sample rate being used by MSP, then **groove~** should play the sample half as fast.

You can use the objects **info~** and **dspstate~** to find out the sampling rate of the sample and of MSP respectively, as demonstrated in the following example.



Calculate playback speed based on the sampling rates of the audio file and of MSP

The note-on key number is used first to recall the information for the sample to be played. The name of a **buffer~** is sent to **groove~** and **info~**. Next, a bang is sent to **dspstate~** and **info~**. Upon receiving a bang, **dspstate~** reports the sampling rate of MSP and **info~** reports the sampling rate of the AIFF file stored in the **buffer~**. In the lower left part of the example, you can see how this sampling rate information is used as a factor in determining the correct playback speed for **groove~**.

Playing samples with MIDI

- Turn audio on and set the "Output Level" **number box** to a comfortable listening level. Play a slow chromatic scale on the MIDI keyboard to hear the different samples and their arrangement on the keyboard.

To arrange a unified single instrument sound across the whole keyboard, each key region should contain a sample of a note from the same source. In this case, though, the samples are arranged on the keyboard in such a way as to make available a full "band" consisting of drums, bass, and keyboard. This sort of multi-timbral keyboard layout is useful for simple keyboard splits (such as bass in the left hand and piano in the right hand) or, as in this case, for accessing several different sounds on a single MIDI channel with a sequencer.

- For an example of how a multi-timbral sample layout can be used by a sequencer, click on the **toggle** marked "Play Sequence". Click on it again when you want to stop the sequence. Turn audio off. Double-click on the **p** sequence object to open the Patcher window of the subpatch.

The **p** sequence *subpatch*

The **seq** sampleseq.midi object contains a pre-recorded MIDI file. The **midiparse** object sends the MIDI key number and velocity to **poly** in the main patch. Each time the sequence finishes playing, a bang is sent out the right outlet of **seq**; the bang is used to restart the **seq** immediately, to play the sequence as a continuous loop. When the sequence is stopped by the user, a bang is sent to **midiflush** to turn off any notes currently being played.

- When you have finished with this patch, don't forget to open the DSP Status window and restore the Sampling Rate to its original setting.

Summary

To play samples from the MIDI keyboard, load each sample into a **buffer~** and play the samples with **groove~**. For polyphonic sample playback, you will need one **groove~** object per voice of polyphony. You can route MIDI notes to different **groove~** objects using voice assignments from the **poly** object.

To assign each sample to a region of the MIDI keyboard, you will need to keep a list of key regions, and for each key region you will need to keep information about which **buffer~** to use, what transposition to use, what loop points to use, etc. A **funbuff** object is good for storing keyboard region assignments. The various items of information about each sample can be best stored together as lists in a **coll**, indexed by the key region number. When a note is played, the key region is looked up in the **funbuff**, and that number is used to look up the sample information in **coll**.

The proper transposition for each note can be calculated by dividing the played frequency (obtained with the **mtof** object) by the base frequency of the sample. The result is used as the playback speed for **groove~**. If the sampling rate of the recorded samples differs from the sampling rate being used in MSP, that fact must be accounted for when playing the samples with **groove~**. Dividing the audio file sampling rate by the MSP sampling rate provides the correct factor by which to multiply the playback speed of **groove~**. The sampling rate of MSP can be obtained with the **dspstate~** object. The sampling rate of the AIFF file in a **buffer~** can be obtained with **info~** (Remember—resetting the sampling rate may not be possible on your hardware).

Note-on velocity can be used to control the amplitude of the samples. An exponential mapping of velocity to amplitude is usually best. Multi-timbral sample layouts on the keyboard can be useful for playing many different sounds, especially from a sequencer. The end-of-file bang from the right outlet of **seq** can be used to restart the **seq** to play it in a continuous loop. If the MIDI data goes through a **midiflush** object, any notes that are on when the **seq** is stopped can be turned off by sending a bang to **midiflush**.

See Also

buffer~	Store audio samples
dspstate~	Report current DSP setting
groove~	Variable-rate looping sample playback
poly~	Polyphony/DSP manager for patchers

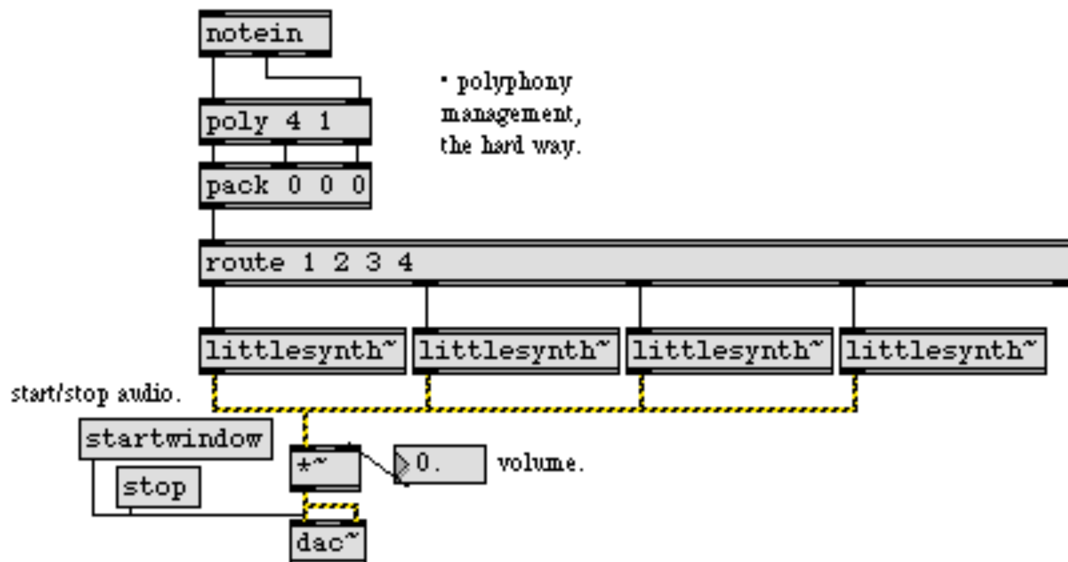
Tutorial 21

MIDI control: Using the **poly~** *object*

A different approach to polyphony

In the last chapter, we demonstrated how to use the **poly** object to make polyphonic voice assignments in a simple case. This chapter will describe a more elegant and efficient way to handle polyphonic voice allocation—the **poly~** object.

In the example in the previous chapter, we created multiple copies of our sampler subpatch and used the poly object's voice numbering to route messages to different copies of the subpatch. Our example could just as easily have used any kind of sound-producing subpatch. The following example uses the subpatch **littlesynth~** to implement a simple four-voice polyphonic synthesizer:



While this method works, it has two disadvantages. First, there's a lot of housekeeping necessary to duplicate and patch the multiple copies of **littlesynth~** together. But there is also a problem in terms of CPU usage. All four copies of the **littlesynth~** subpatcher are always on, processing their audio even when there is no sound being produced.

MSP 2.0, introduces a different way to solve the problem—the **poly~** object allows you to create and manage multiple copies of the same MSP subpatch all within one object. You can also control the signal processing activity within each copy of the subpatch to conserve CPU resources.

The poly~ object

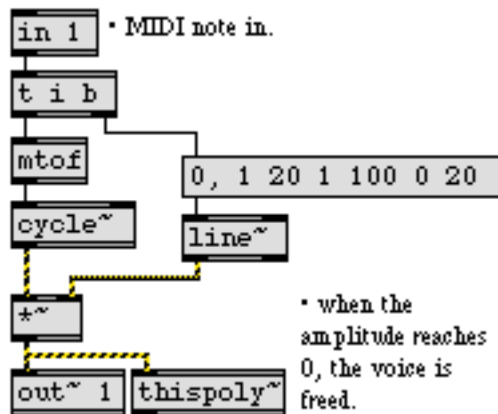
The **poly~** object takes as its argument the name of a patcher file, followed by a number that specifies the number of copies (or *instances*) of the patch to be created. You'll want to specify the same

number of copies as you would have had to duplicate manually when implementing polyphony the old-fashioned way. Here's an example of the **poly~** object.

poly~ littlebeep~ 16

• make 16 copies of littlebeep~
and manage them with poly~

Double-clicking on the **poly~** object opens up the subpatcher to show you the inside of the **littlebeep~** object:



Let's look at the **littlebeep~** patch for a minute. While you haven't seen the **in**, **out~**, or **thispoly~** objects before, the rest of the patcher is pretty straightforward; it takes an incoming MIDI note number, converts it to a frequency value using the **mtof** object, and outputs a sine wave at that frequency with a duration of 140 milliseconds and an amplitude envelope supplied by the **line~** object for 140 ms with an envelope on it.

But what about the **in** and **out~** objects? Subpatches created for use in the **poly~** object use special objects for inlets and outlets. The objects **in** and **out** create control inlets and outlets, and the **in~** and **out~** objects create signal inlets and outlets. You specify which inlet is assigned to which object by adding a number argument to the object—the **in 1** object corresponds to the leftmost inlet on the **poly~** object, and so on. The **poly~** object keeps track of the number of inlets and outlets it needs to create when you tell it which subpatch to load.

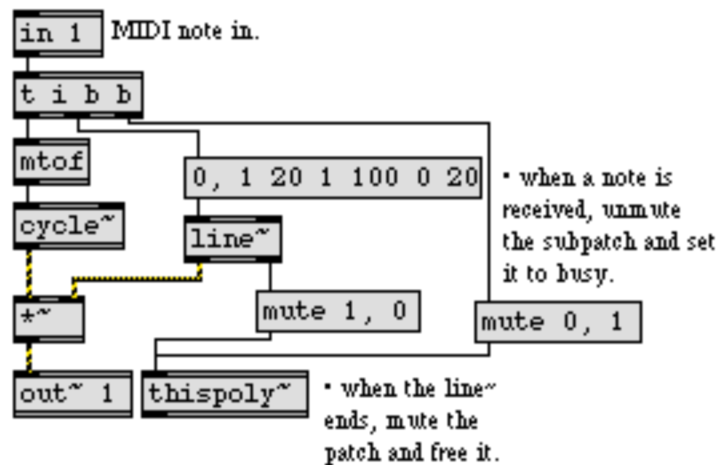
Messages sent to a **poly~** object are directed to different instances of the subpatch dynamically using the note and midinote messages, and manually using the target message.

When **poly~** receives a note message in its left inlet, it scans through the copies of the subpatch it has in memory until it finds one that is currently not busy, and then passes the message to it. A subpatch instance can tell its parent **poly~** object that it is busy using the **thispoly~** object. The **thispoly~** object accepts either a signal or number in its inlet to set its busy state. A zero signal or a value of 0 sent to its inlet tells the parent **poly~** that this instance is available for note or midinote messages. A non-zero signal or value sent to its inlet tells the parent **poly~** that the instance is busy; no note or midinote messages will be sent to the object until it is no longer busy. The busy state was intended to correspond to the duration of a note being played by the subpatcher instance, but it

could be used to mean anything. In the example above, when the audio level out of the *~ is non-zero—that iteration of the subpatch is currently busy. Once the amplitude envelope out of **line~** reaches zero and the sound stops, that subpatch's copy of **thispoly~** tells **poly~** that it is ready for more input.

The **thispoly~** object can also control the activity of signal processing within each copy of the subpatch. When the mute message is sent to **thispoly~** followed by a 1, all signal processing in that subpatch stops. When a mute 0 message is received, signal processing starts again.

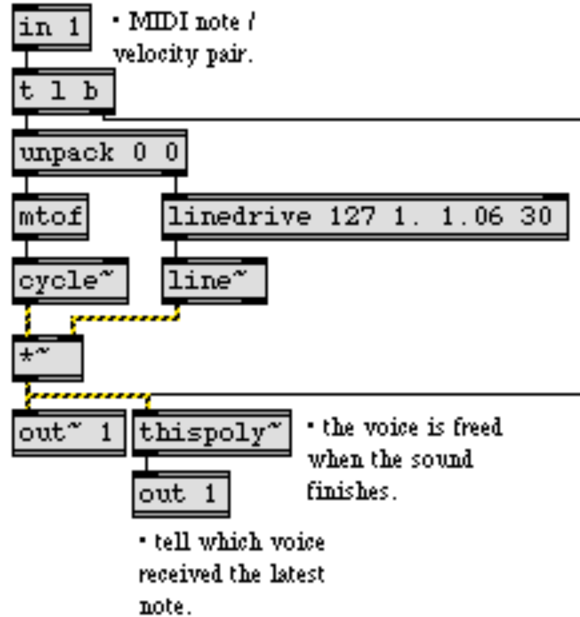
We can rewrite the **littlebeep~** subpatcher to take advantage of this by turning off signal processing when a note is finished and turning it on again when a new event is received:



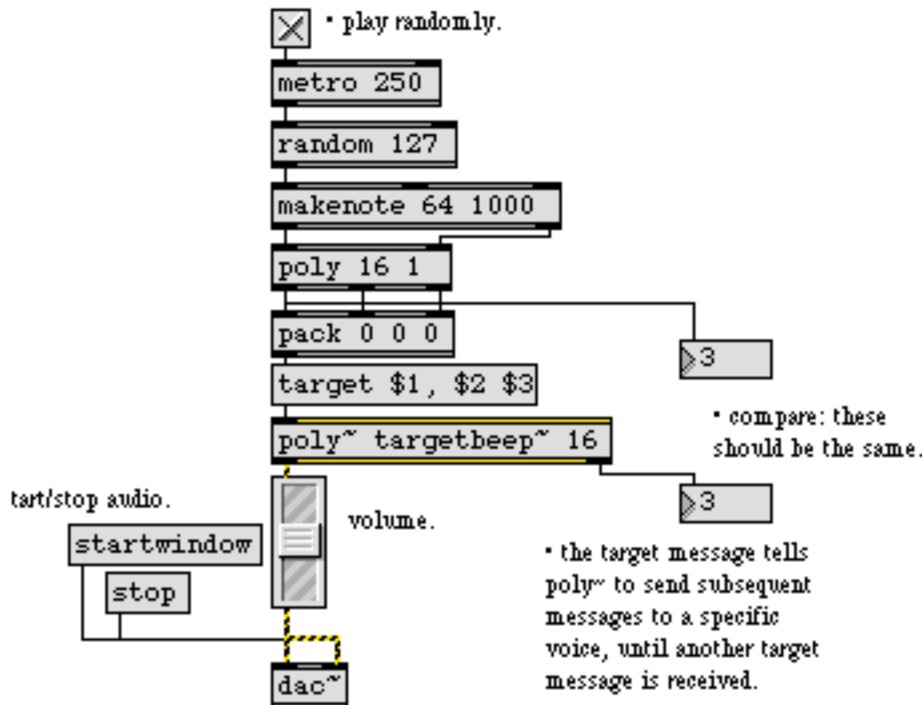
While this doesn’t change the function of the patch, it would be more efficient, since the amount of CPU allocated is always based on the number of notes currently sounding.

Another way to allocate events using **poly~** is through the target message. Sending a target message followed by an integer in the left inlet of a **poly~** subpatch tells **poly~** to send all subsequent messages to that instance of the subpatch. You can then use **poly~** in conjunction with the **poly** object from the last chapter to create a MIDI synthesizer.

A **poly~** subpatch that uses the target message looks like this:

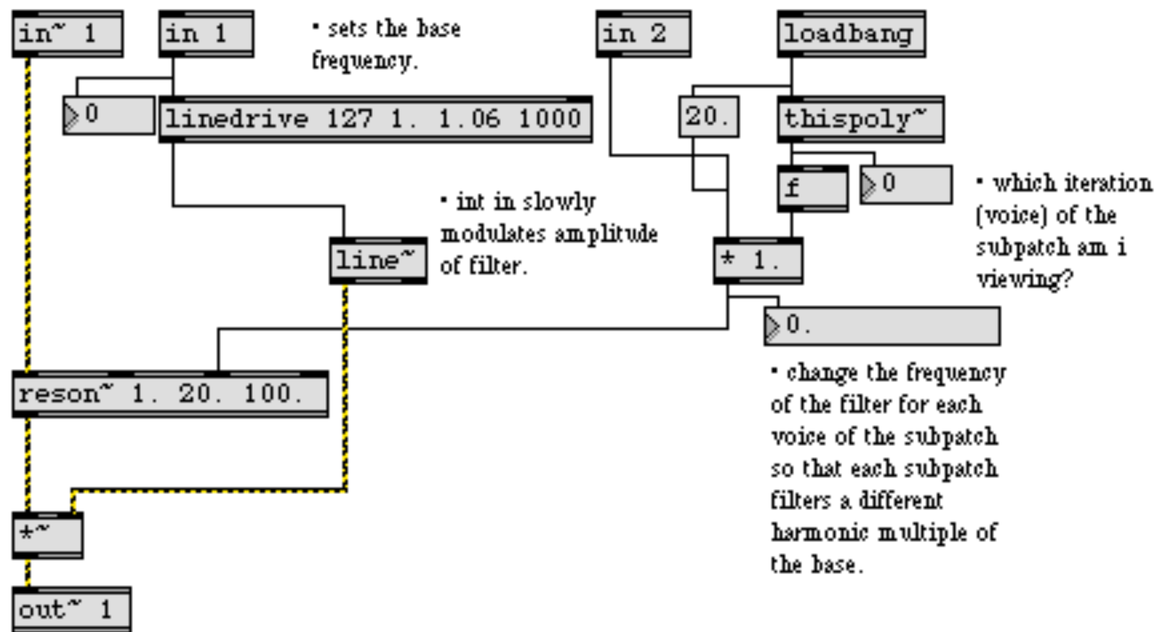


In this example subpatcher, pairs of incoming MIDI pitches and velocities are used to synthesize a sine tone. When a list is received, the subpatcher sends a bang to **thispoly~**, causing it to output the instance or voice number. In the example below, the voice number is sent out an outlet so you can watch it from the parent patch.



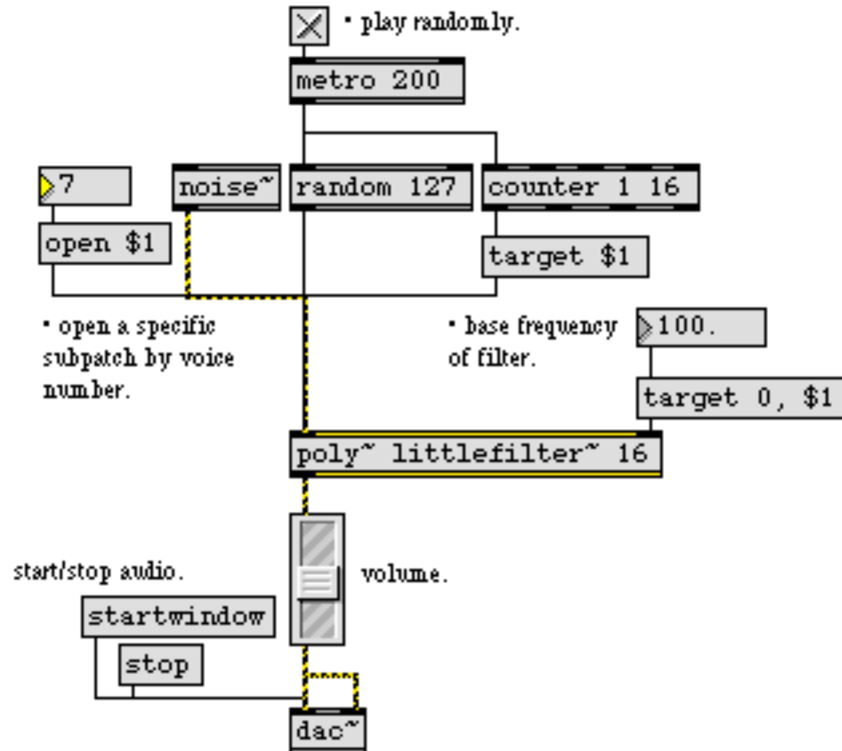
In the parent patch the **poly** object assigns voice numbers to MIDI pitch/velocity pairs output by **makenote**. The voice number from the **poly** object is sent to **poly~** with the target message prepended to it, telling **poly~** to send subsequent data to the instance of the **targetbeep~** subpatcher specified by **poly~**. When a new note is generated, the target will change. Since **poly** keeps track of note-offs, it should recycle voices properly. The second outlet of **poly~** reports the voice that last received a message—it should be the same as the voice number output by **poly**, since we're using **poly** to specify a specific target.

The **thispoly~** object can be used to specify parameters to specific instances of a **poly~** subpatcher. By connecting a **loadbang** object to **thispoly~**, we can use the voice number to control the center frequency of a filter:



The **littlefilter~** subpatcher, shown here uses the voice number from **thispoly~** and multiplies it by the base frequency received in the second inlet. The incoming signal is filtered by all sixteen instances simultaneously, with the output amplitude of each instance being controlled by an integer coming into the first inlet.

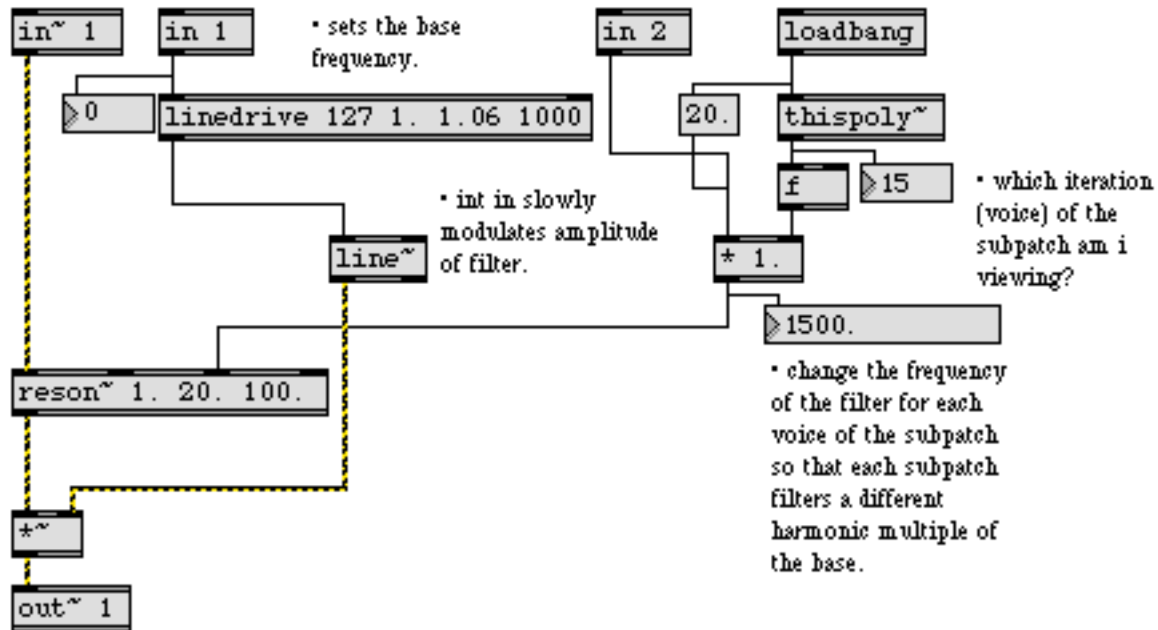
Here's an example of a patch which uses **littlefilter~**:



The **metro** object is hooked up to both a **counter** and a **random**. The **counter**, which feeds the target message, cycles through the 16 voices of **littlefilter~** loaded into the **poly~** object, supplying each with a random number which is used to control the amplitude of that voice.

A signal connected to an inlet of **poly~** will be sent to the corresponding **in~** objects of all subpatcher instances, so the **noise~** object in the example above feeds noise to all the subpatchers inside the **poly~**. The second inlet (which corresponds to the **in** 2 box in the subpatcher) controls the base frequency of the filters. Note that for the frequency to get sent to all **poly~** iterations it is preceded by a target 0 message. You can open a specific instance of a **poly~** subpatch by giving the

object the open message, followed by the voice you want to look at. The subpatch assigned to voice number 15 looks like this:



As you can see, the base frequency of this particular iteration of **littlefilter~** is 1500. Hz, which is the multiple of the voice number (15) with the most recently entered base frequency into the second inlet (100. Hz).

Summary

poly~ is a powerful way to manage multiple copies of the same subpatch for polyphonic voice allocation. The **thispoly~** object works inside a subpatch to control its busy state and turn signal processing on and off. The objects **in**, **in~**, **out**, and **out~** create special control and signal inputs and outputs that work with the inlets and outlets of the **poly~** object.

See Also

See Also

in	Message input for a patcher loaded by **poly~**
in~	Signal input for a patcher loaded by **poly~**
out	Message output for a patcher loaded by **poly~**
out~	Signal output for a patcher loaded by **poly~**
poly~	Polyphony/DSP manager for patchers
thispoly~	Control **poly~** voice allocation and muting

Tutorial 22

MIDI control: Panning

Panning for localization and distance effects

Loudness is one of the cues we use to tell us how far away a sound source is located. The relative loudness of a sound in each of our ears is a cue we use to tell us in what direction the sound is located. (Other cues for distance and location include inter-aural delay, ratio of direct to reflected sound, etc. For now we"ll only be considering loudness.)

When a sound is coming from a single speaker, we localize the source in the direction of that speaker. When the sound is equally balanced between two speakers, we localize the sound in a direction precisely between the speakers. As the balance between the two speakers varies from one to the other, we localize the sound in various directions between the two speakers.

The term panning refers to adjusting the relative loudness of a single sound coming from two (or more) speakers. On analog mixing consoles, the panning of an input channel to the two channels of the output is usually controlled by a single knob. In MIDI, panning is generally controlled by a single value from 0 to 127. In both cases, a single continuum is used to describe the balance between the two stereo channels, even though the precise amplitude of each channel at various intermediate points can be calculated in many different ways.

All other factors being equal, we assume that a softer sound is more distant than a louder sound, so the overall loudness effect created by the combined channels will give us an important distance cue. Thus, panning must be concerned not only with the proper balance to suggest direction of the sound source; it must also control the perceived loudness of the combined speakers to suggest distance.

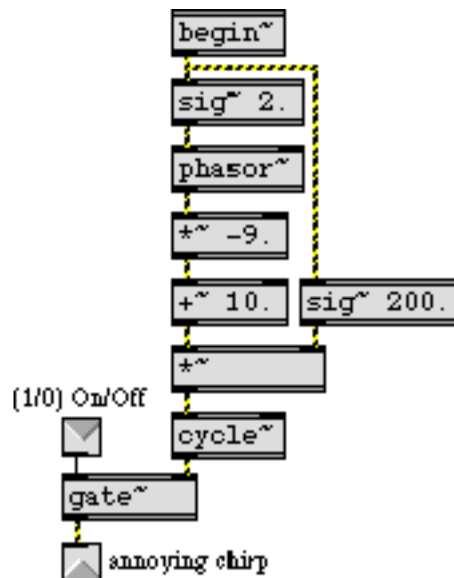
This tutorial demonstrates three ways of calculating panning, controllable by MIDI values 0 to 127. You can try out the three methods and decide which is most appropriate for a given situation in which you might want to control panning.

Patch for testing panning methods

In this tutorial patch, we use a repeated "chirp" (a fast downward glissando spanning more than three octaves) as a distinctive and predictable sound to pan from side to side.

- To see how the sound is generated, double-click on the **p** "sound source" subpatch to open its Patcher window.

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

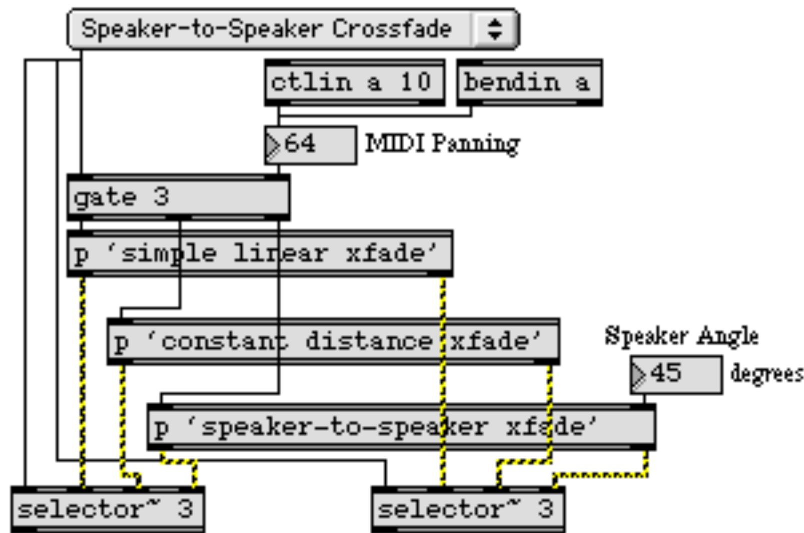


The **p** "sound source" *subpatch*

- Close the subpatch window.

The output of this subpatch is sent to two *~ objects—one for each output channel—where its amplitude at each output channel will be scaled by one of the panning algorithms. You can choose the panning algorithm you want to try from the pop-up **umenu** at the top of the patch. This opens the inlet of the two **selector~** objects to receive the control signals from the correct panning subpatch. It also opens an outlet of the **gate** object to allow control values into the desired subpatch. The panning is controlled by MIDI input from continuous controller No. 10 (designated for panning in MIDI). In case your MIDI keyboard doesn"t send controller 10 easily, you can also use the

pitch bend wheel to test the panning. (For that matter, you don"t need MIDI at all. You can just drag on the **number box** marked “MIDI panning”.)

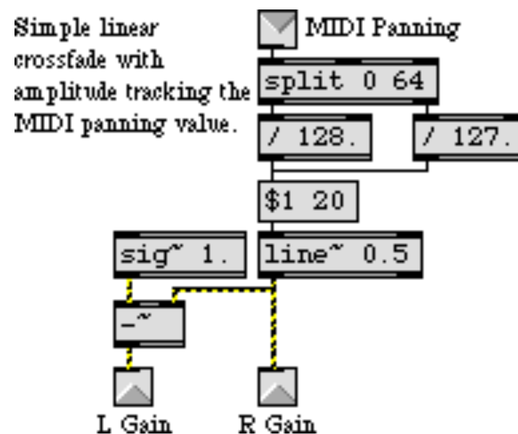


Selection from the **umenu** *opens input and output for one of the three panning subpatches*

Linear crossfade

The most direct way to implement panning is to fade one channel linearly from 0 to 1 as the other channel fades linearly from 1 to 0. This is the easiest type of panning to calculate. We map the range of MIDI values 0 to 127 onto the amplitude range 0 to 1, and use that value as the amplitude for the right channel; the left channel is always set to 1 minus the amplitude of the left channel. The only hitch is that a MIDI pan value of 64 is supposed to mean equal balance between channels, but it is not precisely in the center of the range ($^{64}/_{127}$ - 0.5). So we have to treat MIDI values 0 to 64 differently from values 65 to 127.

- Double-click on the **p** “simple linear xfade” object to open its Patcher window.



Linear crossfade using MIDI values 0 to 127 for control

This method seems perfectly logical since the sum of the two amplitudes is always 1. The problem is that the *intensity* of the sound is proportional to the sum of the *squares* of the amplitudes from each speaker. That is, two speakers playing an amplitude of 0.5 do not provide the same intensity (thus not the same perceived loudness) as one speaker playing an amplitude of 1. With the linear crossfade, then, the sound actually seems softer when panned to the middle than it does when panned to one side or the other.

- Close the subpatch window. Choose “Simple Linear Crossfade” from the **umenu**. Click on the **ezdac~** to turn audio on, click on the **toggle** to start the “chirping” sound, and use the “Amplitude” **number box** to set the desired listening level. Move the pitch bend wheel of your MIDI keyboard to pan the sound slowly from one channel to the other. Listen to determine if the loudness of the sound seems to stay constant as you pan.

While this linear crossfade might be adequate in some situations, we may also want to try to find a way to maintain a constant intensity as we pan.

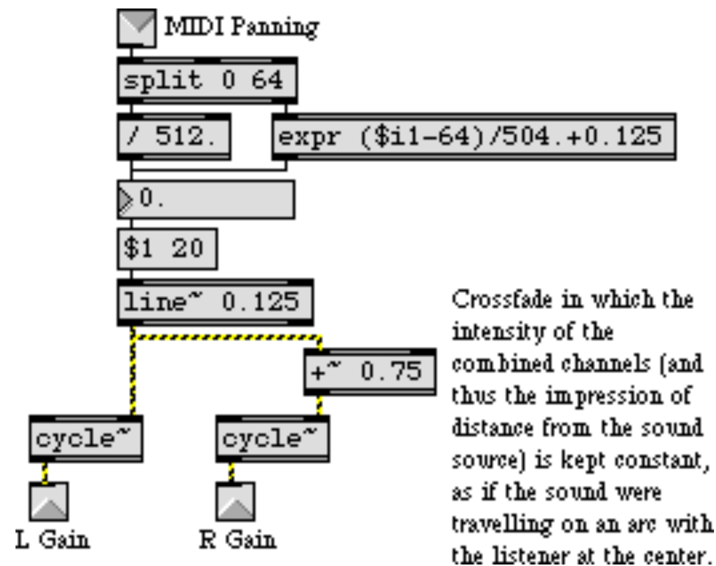
Equal distance crossfade

If we can maintain a constant intensity as we pan from one side to the other, this will give the impression that the sound source is maintaining a constant distance from the listener. Geometrically, this could only be true if the sound source were moving in an arc, with the listener at the center, so that the distance between the sound source and the listener was always equal to the radius of the arc.

It happens that we can simulate this condition by mapping one channel onto a quarter cycle of a cosine wave and the other channel onto a quarter cycle of a sine wave. Therefore, we"ll map the range of MIDI values 0 to 127 onto the range 0 to 0.25, and use the result as an angle for looking up the cosine and sine values.

> **Technical detail:** As the sound source travels on a hypothetical arc from 0° to 90° ($^1/_4$ cycle around a circle with the listener at the center), the cosine of its angle goes from 1 to 0 and the sine of its angle goes from 0 to 1. At all points along the way, the square of the cosine plus the square of the sine equals 1. This trigonometric identity is analogous to what we are trying to achieve—the sum of the squares of the amplitudes always equaling the same intensity—so these values are a good way to obtain the relative amplitudes necessary to simulate a constant distance between sound source and listener.

- Double-click on the **p** "constant distance xfade" object to open its Patcher window.



MIDI values 0 to 127 are mapped onto $^1/_4$ cycle of cosine and sine functions

Once again we need to treat MIDI values greater than 64 differently from those less than or equal to 64, in order to retain 64 as the "center" of the range. Once the MIDI value is mapped into the range 0 to 0.25, the result is used as a phase angle two **cycle~** objects, one a cosine and the other (because of the additional phase offset of 0.75) a sine.

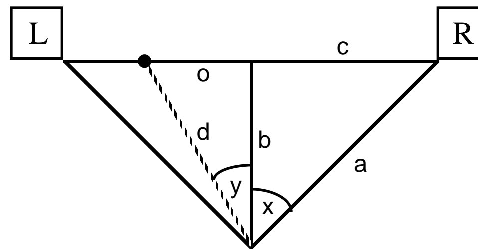
- Close the subpatch window. Choose "Equal Distance Crossfade" from the **umenu**. Listen to the sound while panning it slowly from one channel to the other.

Is the difference from the linear crossfade appreciable? Perhaps you don"t care whether the listener has the impression of movement in an arc when listening to the sound being panned. But the important point is that the equal distance method is preferable if only because it does not cause a noticeable dip in intensity when panning from one side to the other.

Speaker-to-speaker crossfade

Given a standard stereo speaker placement—with the two speakers in front of the listener at equal distances and angles—if an actual sound source (say, a person playing a trumpet) moved in a straight line from one speaker to the other, the sound source would actually be *closer* to the listener when it"s in the middle than it would be when it"s at either speaker. So, to emulate a sound source

moving in a straight line from speaker to speaker, we will need to calculate the amplitudes such that the intensity is proportional to the distance from the listener.



Distance b is shorter than distance a

Technical detail: If we know the angle of the speakers (*x* and -*x*), we can use the cosine function to calculate distance *a* relative to distance *b*. Similarly we can use the tangent function to calculate distance c relative to b. The distance between the speakers is thus *2c*, and as the MIDI pan value varies away from its center value of 64 it can be mapped as an offset (*o*) from the center ranging from -*c* to +*c*. Knowing *b* and *o*, we can use the Pythagorean theorem to obtain the distance (*d*) of the source from the listener, and we can use the arctangent function to find its angle (*y*). Armed with all of this information, we can finally calculate the gain for the two channels as $a \cdot \cos(y \pm x)/d$.

- Choose "Speaker-to-Speaker Crossfade" from the **umenu**. Listen to the sound while panning it slowly from one channel to the other. You can try different speaker angles by changing the value in the "Speaker Angle" **number box**. Choose a speaker angle best suited to your actual speaker positions.

This effect becomes more pronounced as the speaker angle increases. It is most effective with "normal" speaker angles ranging from about 30° up to 45°, or even up to 60°. Below 30° the effect is too slight to be very useful, and above about 60° it"s too extreme to be realistic.

- Double-click on the **p** "speaker-to-speaker xfade" object to open its Patcher window.

The trigonometric calculations described above are implemented in this subpatch. The straight ahead distance (*b*) is set at 1, and the other distances are calculated relative to it. The speaker angle—specified in degrees by the user in the main patch—is converted to a fraction of a cycle, and is eventually converted to radians (multiplied by 2π, or 6.2832) for the trigonometric operations. When the actual gain value is finally calculated, it is multiplied by a normalizing factor of $2/(d+b)$ to avoid clipping. When the source reaches an angle greater than 90° from one speaker or the other, that speaker"s gain is set to 0.

- To help get a better understanding of these calculations, move the pitch bend wheel and watch the values change in the subpatch. Then close the subpatch and watch the gain values change in the main Patcher window.

The signal gain values are displayed by an MSP user interface object called **number~**, which is explained in the next chapter.

Summary

MIDI controller No. 10 (or any other MIDI data) can be used to pan a signal between output channels. The relative amplitude of the two channels gives a localization cue for direction of the sound source. The overall intensity of the sound (which is proportional to the sum of the squares of the amplitudes) is a cue for perceived distance of the sound source.

Mapping the MIDI data to perform a linear crossfade of the amplitudes of the two channels is one method of panning, but it causes a drop in intensity when the sound is panned to the middle. Using the panning value to determine the *angle* of the sound source on an arc around the listener (mapped in a range from 0° to 90°), and setting the channel amplitudes proportional to the cosine and sine of that angle, keeps the intensity constant as the sound is panned.

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

See Also

expr	Evaluate a mathematical expression
gate~	Route a signal to one of several outlets

Tutorial 23

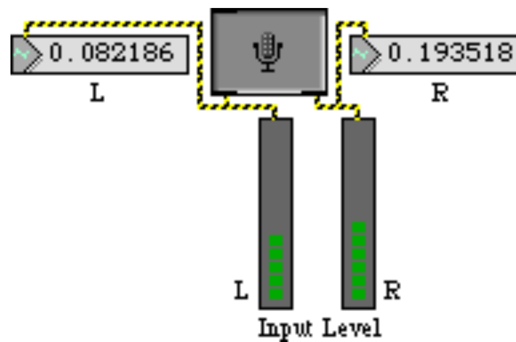
Analysis: Viewing signal data

Display the value of a signal: number~

This chapter demonstrates several MSP objects for observing the numerical value of signals, and/ or for translating those values into Max messages.

- Turn audio on and send some sound into the input jacks of the computer.

Every 250 milliseconds the **number~** objects at the top of the Patcher display the current value of the signal coming in each channel, and the **meter~** objects show a graphic representation of the peak amplitude value in the past 250 milliseconds, like an analog LED display.



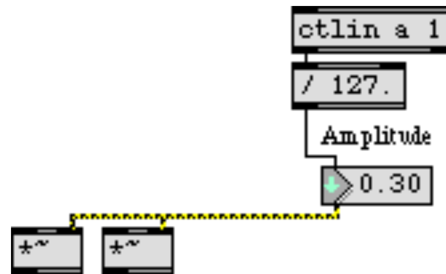
Current signal value is shown by **number~** *; peak amplitude is shown by* **meter~**

The signal coming into **number~** is sent out its right outlet as a float once every time it's displayed. This means it is possible to sample the signal value and send it as a message to other Max objects.

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

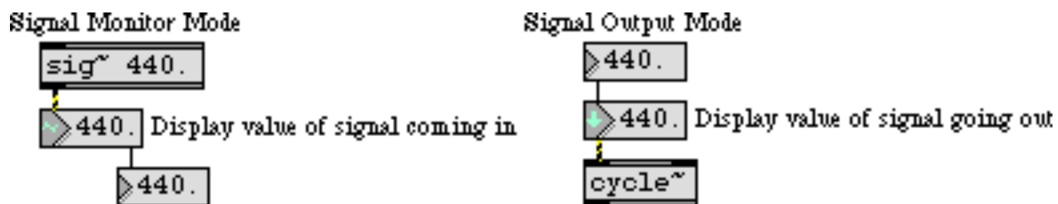
- Move the mod wheel of your MIDI keyboard or drag on the right side of the **number~** marked "Amplitude". This sets the value of the signal being sent out the **number~** object's left outlet.

The signal is connected to the right inlet of two *~ objects, to control the amplitude of the signal sent to the **ezdac~**.



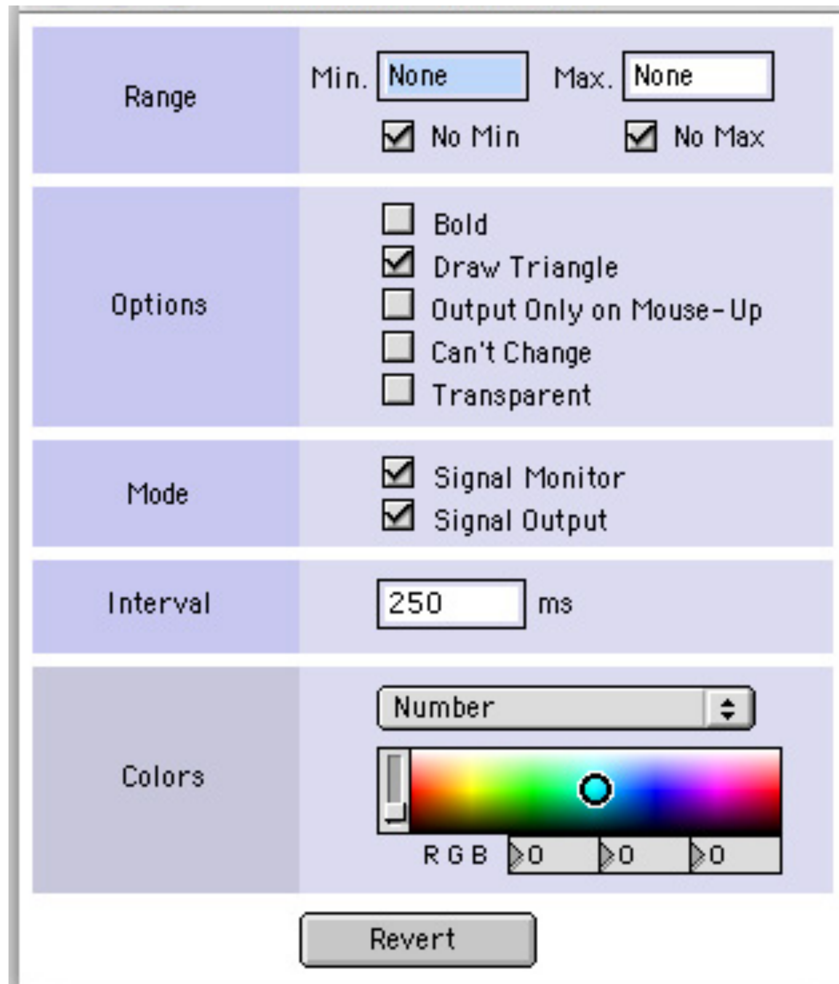
float *input to* **number~** *sets the value of the signal sent out the left outlet*

A **number~** object simultaneously converts any signal it receives into floats sent out the right outlet, and converts any float it receives into a signal sent out the left outlet. Although it can perform both tasks at the same time, it can only display one value at a time. The value displayed by **number~** depends on which *display mode* it is in. When a small waveform appears in the left part of the **number~**, it is in *Signal Monitor Mode*, and shows the value of the signal coming in the left inlet. When a small arrow appears in the left part of **number~**, it is in *Signal Output Mode*, and shows the value of the signal going out the left outlet.



The two display modes of **number~**

You can restrict **number~** to one display mode or the other by selecting the object in an unlocked Patcher and choosing **Get Info…** from the Object menu.



Allowed display modes can be chosen in the **number~** *Inspector*

At least one display mode must be checked. By default, both display modes are allowed, as shown in the above example. If both display modes are allowed, you can switch from one display mode to the other in a locked Patcher by clicking on the left side of the **number~**. The output of **number~** continues regardless of what display mode it's in.

In the tutorial patch you can see the two display modes of **number~**. The **number~** objects at the top of the Patcher window are in *Signal Monitor Mode* because we are using them to show the value of the incoming signal. The "Amplitude" **number~** is in *Signal Output Mode* because we are using it to send a signal and we want to see the value of that signal. (New values can be entered into a **number~** by typing or by dragging with the mouse only when it is in *Signal Output* display mode.) Since each of these **number~** objects is serving only one function, each has been restricted to only one display mode in the Inspector window.

- Click on the left side of the **number~** objects. They don't change display mode because they have been restricted to one mode or the other in the Inspector window.

Interpolation with number~

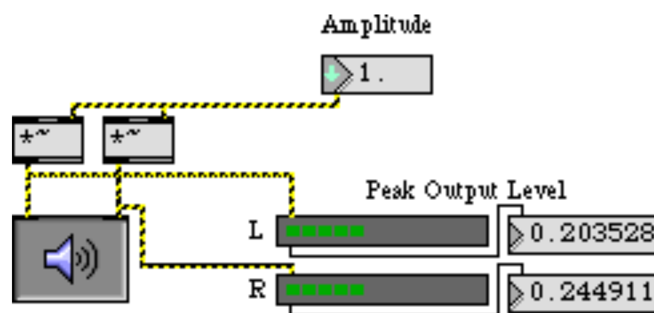
The **number~** object has an additional useful feature. It can be made to interpolate between input values to generate a ramp signal much like the **line~** object. If **number~** receives a non-zero number in its right inlet, it uses that number as an amount of time, in milliseconds, to interpolate linearly to the new value whenever it receives a number in the left inlet. This is equivalent to sending a list to **line~**.

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

Unlike **line~**, however, **number~** does not need to receive the interpolation time value more than once; it remembers the interpolation time and uses it for each new number received in the left inlet. This feature is used for the "Amplitude" **number~** so that it won't cause discontinuous changes of amplitude in the output signal.

Peak amplitude: meter~

The **meter~** object periodically displays the peak amplitude it has received since the last display. At the same time it also sends the peak signal value out its outlet as a float. The output value is always a positive number, even if the peak value was negative.



meter~ *displays the peak signal amplitude and sends it out as a* float

meter~ is useful for observing the peak amplitude of a signal (unlike **number~**, which displays and sends out the *instantaneous* amplitude of the signal). Since **meter~** is intended for audio signals, it expects to receive a signal in the range -1 to 1. If that range is exceeded, **meter~** displays a red "clipping" LED as its maximum.

- If you want to see the clipping display, increase the amplitude of the output signal until it exceeds 1. (Then return it to a desirable level.)

The default interval of time between the display updates of **meter~** is 250 milliseconds, but the display interval can be altered with the interval message. A shorter display interval makes the LED display more accurate, while a longer interval gives you more time to read its visual and numerical output.

- You can try out different display intervals by changing the number in the **number box** marked “Display Interval” in the lower left corner of the Patcher window.

By the way, the display interval of a **number~** object can be set in the same manner (as well as via its Inspector window).

Use a signal to generate Max messages: snapshot~

The **snapshot~** object sends out the current value of a signal, as does the right inlet of **number~**. With **snapshot~**, though, you can turn the output on and off, or request output of a single value by sending it a bang. When you send a non-zero number in the right inlet, **snapshot~** uses that number as a millisecond time interval, and begins periodically reporting the value of the signal in its left inlet. Sending in a time interval of 0 stops **snapshot~**.

This right half of the tutorial patch shows a simple example of how a signal waveform might be used to generate MIDI data. We’ll sample a sub-audio cosine wave to obtain pitch values for MIDI note messages.

- Use the **number~** to set the output amplitude to 0. In the **number box** objects at the top of the patch, set the “Rate” number box to 0.14 and set the “Depth” **number box** to 0.5. Click on the message box 200 to start **snapshot~** reporting signal values every fifth of a second.

Because **snapshot~** is reporting the signal value every fifth of a second, and the period of the **cycle~** object is about 7 seconds, the melody will describe one cycle of a sinusoidal wave every 35 notes. Since the amplitude of the wave is 0.5, the melody will range from 36 to 84 (60±24).

- Experiment with different “Rate” and “Depth” values for the **cycle~**. Since **snapshot~** is sampling at a rate of 5 Hz (once every 200 ms), its Nyquist rate is 2.5 Hz, so that limits the effective frequency of the **cycle~** (any greater frequency will be “folded over”). Click on the 0 **message** box to stop **snapshot~**.

Amplitude modulation

- Set the tremolo depth to 0.5 and the tremolo rate to 4. Increase the output amplitude to a desirable listening level.

The **cycle~** object is modulating the amplitude of the incoming sound with a 4 Hz tremolo.

- Experiment with faster (audio range) rates of modulation to hear the timbral effect of amplitude modulation. To hear ring modulation, set the modulation depth to 1. To remove the modulation effect, simply set the depth to 0.

View a signal excerpt: capture~

The **capture~** object is comparable to the Max object **capture**. It stores many signal values (the most recently received 4096 samples, by default), so that you can view an entire excerpt of a signal as text.

- Set the tremolo depth to 1, and set the tremolo rate to 172. Double-click on the **capture~** object to open a text window containing the last 4096 samples.

This object is useful for seeing precisely what has occurred in a signal over time. (4096 samples is about 93 milliseconds at a sampling rate of 44.1 kHz.) You can type in an argument to specify how many samples you want to view, and **capture~** will store that many samples (assuming there is enough RAM available in Max. There are various arguments and messages for controlling exactly what will be stored by **capture~**. See its description in the MSP Reference Manual for details.

Summary

The **capture~** object stores a short excerpt of a signal to be viewed as text. The **meter~** object periodically displays the peak level of a signal and sends the peak level out its outlet as a float. The **snapshot~** object sends out a float to report the current value of a signal. **snapshot~** reports the signal value once when it receives a bang, and it can also report the signal value periodically if it receives a non-zero interval time in its right inlet.

The **number~** object is like a combination of a float **number box**, **sig~**, and **snapshot~**, all at once. A signal received in its left inlet is sent out the right outlet as a float, as with **snapshot~**. A float or int received in its left inlet sets the value of the signal going out its left outlet, as with **sig~**. Both of these activities can go on at once in the same **number~** object, although **number~** can only *display* one value at a time. When **number~** is in *Signal Monitor Mode*, it displays the value of the incoming signal. When **number~** is in *Signal Output Mode*, it displays the value of the outgoing signal.

number~ can also function as a signal ramp generator, like the **line~** object. If a non-zero number has been received in the right inlet (representing interpolation time in milliseconds), whenever **number~** receives a float, its output signal interpolates linearly between the old and new values.

These objects (along with a few others such as **sig~**, **peek~** and **avg~**) comprise the primary links between MSP and Max. They convert signals to numerical Max messages, or vice versa.

See Also

capture~	Store a signal to view as text
meter~	Visual peak level indicator
number~	Signal monitor and constant generator
snapshot~	Convert signal values to numbers

Tutorial 24

Analysis: Oscilloscope

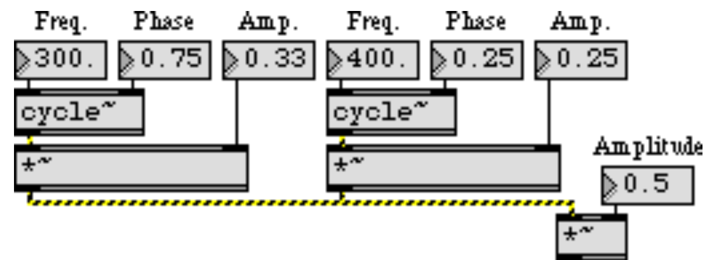
Graph of a signal over time

There are times when seeing a picture of a signal is instructive. The **scope~** object depicts the signal it receives, in the manner of an analog oscilloscope, as a graph of amplitude over time.

There are two problems **scope~** must consider when plotting a graph of a signal in real time. First of all, in order for your eye to follow a time-varying signal, an excerpt of the signal must be captured and displayed for a certain period of time (long enough for you really to see it). Therefore, the graph must be displayed periodically, and will always lag a bit behind what you hear. Second, there aren't enough pixels on the screen for you to see a plot of every single sample (at least, not without the display being updated at blinding speed), so **scope~** has to use a single pixel to summarize many samples.

A patch to view different waveforms

This tutorial shows how to get a useful display of a signal. The patch adds four cosine oscillators to create a variety of waveforms, and displays them in **scope~**. The frequency, phase, and amplitude of each sinusoid is set independently, and the over-all amplitude of the sum of the oscillators is scaled with an additional ***~** object. The settings for each waveform are stored in a **preset** object.



Additive synthesis can be used to create a variety of complex waveforms

- Click on the first preset in the **preset** object.

When audio is turned on, the **dspstate~** object sends the current sampling rate out its middle outlet. This is divided by the number of pixels per display buffer (the display buffer is where the display points are held before they're shown on the screen), and the result is the number of signal samples per display point (samples per pixel). This number is sent in the left inlet of **scope~** to tell it how many samples to assign to each display pixel. The default number of pixels per display buffer is 128, so by this method each display will consist of exactly one second of signal. In other words, once per second **scope~** displays the second that has just passed. We have stretched the **scope~** (using its grow handle) to be 256 pixels wide—twice its default width—in order to provide a better view.

On the next page we will describe the different waveforms created by the oscillators.

- One by one, click on the different presets to see different waveforms displayed in the **scope~**. The first eight waves are at the sub-audio frequency of 1 Hz to allow you to see a single cycle of the waveform, so the signal is not sent to the **dac~** until the ninth preset is recalled.

Preset 1. A 1 Hz cosine wave.

Preset 2. A 1 Hz sine wave. (A cosine wave with a phase offset of $^3/_4$ cycle.)

Preset 3. A 1 Hz cosine wave plus a 2 Hz cosine wave (i.e. octaves).

Preset 4. Four octaves: cosine waves of equal amplitude at 1, 2, 4, and 8 Hz.

Preset 5. A band-limited square wave. The four oscillators produce four sine waves with the correct frequencies and amplitudes to represent the first four partials of a square wave. (Although the amplitudes of the oscillators are only shown to two decimal places, they are actually stored in the preset with six decimal place precision.)

Preset 6. A band-limited sawtooth wave. The four oscillators produce four sine waves with the correct frequencies and amplitudes to represent the first four partials of a sawtooth wave.

Preset 7. A band-limited triangle wave. The four oscillators produce four sine waves with the correct frequencies and amplitudes to represent the first four partials of a triangle wave (which, it appears, is actually not very triangular without its upper partials).

Preset 8. This wave has the same frequencies and amplitudes as the band-limited square wave, but has arbitrarily chosen phase offsets for the four components. This shows what a profound effect the phase of components can have on the appearance of a waveform, even though its effect on the sound of a waveform is usually very slight.

Preset 9. A 32 Hz sinusoid plus a 36 Hz sinusoid (one-half cycle out of phase for the sake of the appearance in the **scope~**). The result is interference causing beating at the difference frequency of 4 Hz.

Preset 10. Combined sinusoids at 200, 201, and 204 Hz, producing beats at 1, 3, and 4 Hz.

Preset 11. Although the frequencies are all displayed as 200 Hz, they are actually 200, 200.25, 200.667, and 200.8. This produces a complicated interference pattern of six different sub-audio beat frequencies, a pattern which only repeats precisely every minute. We have set the number of samples per pixel much lower, so each display represents about 50 ms. This allows you to see about 10 wave cycles per display.

Preset 12. Octaves at 100, 200, and 400 Hz (with different phase offsets), plus one oscillator at 401 Hz creating beats at 1 Hz.

Preset 13. A cluster of equal-tempered semitones. The dissonance of these intervals is perhaps all the more pronounced when pure tones are used. Each display shows about 100 ms of sound.

Preset 14. A just-tuned dominant seventh chord; these are the 4th, 5th, 6th, and 7th harmonics of a common fundamental, so their sum has a periodicity of 100 Hz, two octaves below the chord itself.

Preset 15. Total phase cancellation. A sinusoid is added to a copy of itself 180° out of phase.

Preset 16. All oscillators off.

Summary

The **scope~** object gives an oscilloscope view of a signal, graphing amplitude over time. Because **scope~** needs to collect the samples before displaying them, and because the user needs a certain period of time to view the signal, the display always lags behind the signal by one display period. A display period (in seconds) is determined by the number of pixels per display buffer, times the number of samples per pixel, divided by the signal sampling rate. You can control those first two values by sending integer values in the inlets of **scope~**. The sampling rate of MSP can be obtained with the **dspstate~** object.

See Also

dspstate~	Report current DSP setting
scope~	Signal oscilloscope

Tutorial 25

Analysis: Using the FFT

Fourier's theorem

The French mathematician Joseph Fourier demonstrated that any periodic wave can be expressed as the sum of harmonically related sinusoids, each with its own amplitude and phase. Given a digital representation of a periodic wave, one can employ a formula known as the discrete Fourier transform (DFT) to calculate the frequency, phase, and amplitude of its sinusoidal components. Essentially, the DFT *transforms* a time-domain representation of a sound wave into a frequency-domain spectrum.

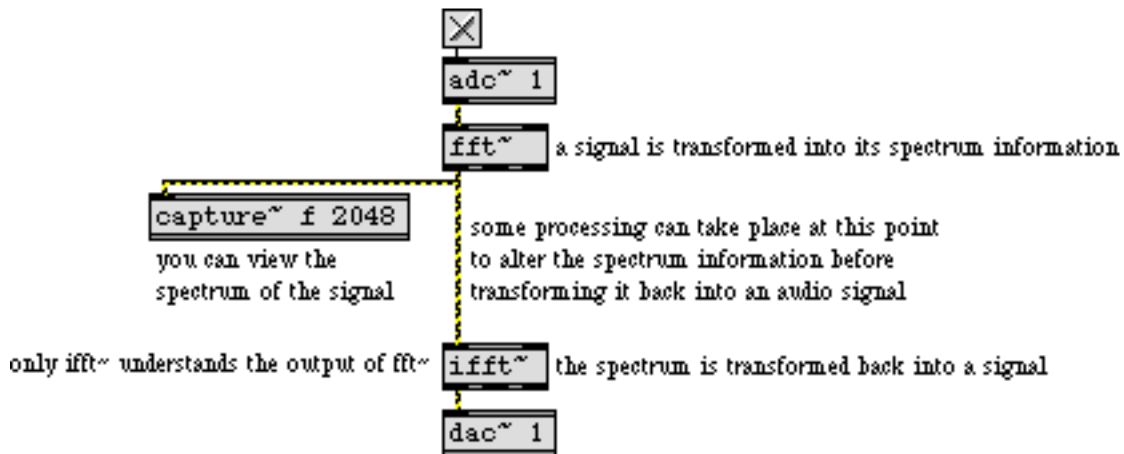
Typically the Fourier transform is used on a small "slice" of time, which ideally is equal to exactly one cycle of the wave being analyzed. To perform this operation on "real world" sounds—which are almost invariably *not* strictly periodic, and which may be of unknown frequency—one can perform the DFT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

If the number of digital samples in each time slice is a power of 2, one can use a faster version of the DFT known as the fast Fourier transform (FFT). The formula for the FFT is encapsulated in the **fft~** object. The mathematics of the Fourier transform are well beyond the scope of this manual, but this tutorial chapter will demonstrate how to use the **fft~** object for signal analysis.

Spectrum of a signal: fft~

fft~ receives a signal in its inlet. For each slice of time it receives (512 samples long by default) it sends out a signal of the same length listing the amount of energy in each frequency region. The signal that comes out of **fft~** is not anything you're likely to want to listen to. Rather, it's a list of relative amplitudes of 512 different frequency bands in the received signal. This "list" happens to be exactly the same length as the samples received in each time slice, so it comes out at the same rate as the signal comes in. The signal coming out of **fft~** is a frequency-domain analysis of the samples it received in the previous time slice.

Although the transform comes out of **fft~** in the form of a signal, it is not a time-domain signal. The only object that "understands" this special signal is the **ifft~** object, which performs an *inverse* FFT on the spectrum and transforms it back into a time-domain waveform.

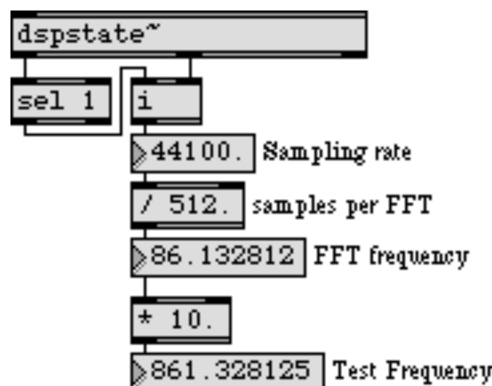


The signal coming out of **fft~** *is spectral information, not a time-domain signal*

With the **capture~** object you can grab some of the output of **fft~** and examine the frequency analysis of a signal.

- Click on one of the **ezdac~** objects to turn audio on.

When audio is turned on, **dspstate~** sends the MSP sampling rate out its middle outlet. We use this number to calculate a frequency that has a period of exactly 512 samples. This is the fundamental frequency of the FFT itself. If we send a wave of that frequency into **fft~**, each time slice would contain exactly one cycle of the waveform. We will actually use a cosine wave at ten times that frequency as the test tone for our analysis, as shown below.



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

The upper left corner of the Patcher window shows a very simple use of **fft~**. The analysis is stored in a **capture~** object, and an **ifft~** object transforms the analysis back into an audio signal. (Ordinarily you would not transform and inverse-transform an audio signal for no reason like this. The **ifft~** is used in this patch simply to demonstrate that the analysis-resynthesis process works.)

- Click on the **toggle** in the upper left part of the patch to hear the resynthesized sound. Click on the **toggle** again to close the **gate~**. Now double-click on the **capture~** object in that part of the patch to see the analysis performed by **fft~**.

In the **capture~** text window, the first 512 numbers are all 0.0000. That is the output of **fft~** during the first time slice of its analysis. Remember, the analysis it sends out is always of the previous time slice. When audio was first turned on, there was no previous audio, so the **fft~** object's analysis shows no signal.

- Scroll past the first 512 numbers. (The numbers in the **capture~** object's text window are grouped in blocks, so if your signal vector size is 256 you will have two groups of numbers that are all 0.0000.) Look at the second time slice of 512 numbers.

Each of the 512 numbers represents a harmonic of the FFT frequency itself, starting at the 0th harmonic (0 Hz). The analysis shows energy in the eleventh number, which represents the 10th harmonic of the FFT, $^{10}/_{512}$ the sampling rate—precisely our test frequency. (The analysis also shows energy at the 10th number from the end, which represents $^{502}/_{512}$ the sampling rate. This frequency exceeds the Nyquist rate and is actually equivalent to $-^{10}/_{512}$ of the sampling rate.)

> **Technical detail:** An FFT divides the entire available frequency range into as many bands (regions) as there are samples in each time slice. Therefore, each set of 512 numbers coming out of **fft~** represents 512 divisions of the frequency range from 0 to the sampling rate. The first number represents the energy at 0 Hz, the second number represents the energy at $^{1}/_{512}$ the sampling rate, the third number represents the energy at $^{2}/_{512}$ the sampling rate, and so on.
>
> Note that once we reach the Nyquist rate on the 257th number ($^{256}/_{512}$ of the sampling rate), all numbers after that are *folded back* down from the Nyquist rate. Another way to think of this is that these numbers represent negative frequencies that are now ascending from the (negative) Nyquist rate. Thus, the 258th number is the energy at the Nyquist rate *minus* $^{1}/_{512}$ of the sampling rate (which could also be thought of as $-^{255}/_{512}$ the sampling rate). In our example, we see energy in the 11th frequency region ($^{10}/_{512}$ the sampling rate) and the 503rd frequency region ($-^{256}/_{512} - -^{246}/_{512} = -^{10}/_{512}$ the sampling rate).

It appears that **fft~** has correctly analyzed the signal. There's just one problem…

Practical problems of the FFT

The FFT assumes that the samples being analyzed comprise one cycle of a periodic wave. In our example, the cosine wave was the 10th harmonic of the FFT's fundamental frequency, so it worked fine. In most cases, though, the 512 samples of the FFT will not be precisely one cycle of the wave. When that happens, the FFT still analyzes the 512 samples as if they were one cycle of a waveform, and reports the spectrum of that wave. Such an analysis will contain many spurious frequencies not actually present in the signal.

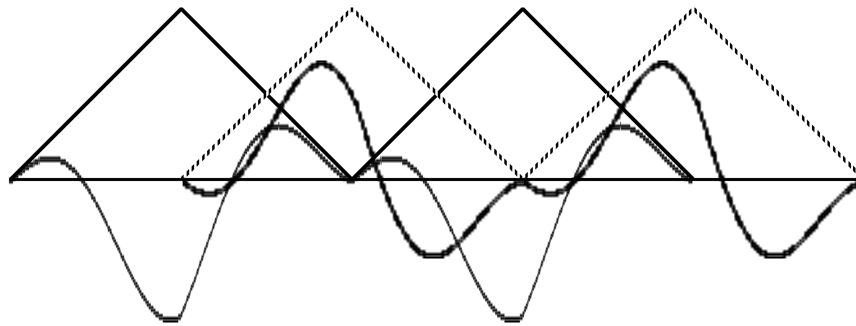
- Close the text window of **capture~**. With the audio still on, set the "Test Frequency" **number box** to 1000. This also triggers the clear message in the upper left corner of the patch to empty the **capture~** object of its prior contents. Double-click once again on **capture~**, and scroll ahead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

The analysis of the 1000 Hz tone does indeed show greater energy at 1000 Hz—in the 12th and 13th frequency regions if your MSP sampling rate is 44,100 Hz—but it also shows energy in virtually every other region. That's because the waveform it analyzed is no longer a sinusoid. (An exact number of cycles does not fit precisely into the 512 samples.) All the other energy shown in this FFT is an artifact of the "incorrect" interpretation of those 512 samples as one period of the correct waveform.

To resolve this problem, we can try to "taper" the ends of each time slice by applying an amplitude envelope to it, and use overlapping time slices to compensate for the use of the envelope.

Overlapping FFTs

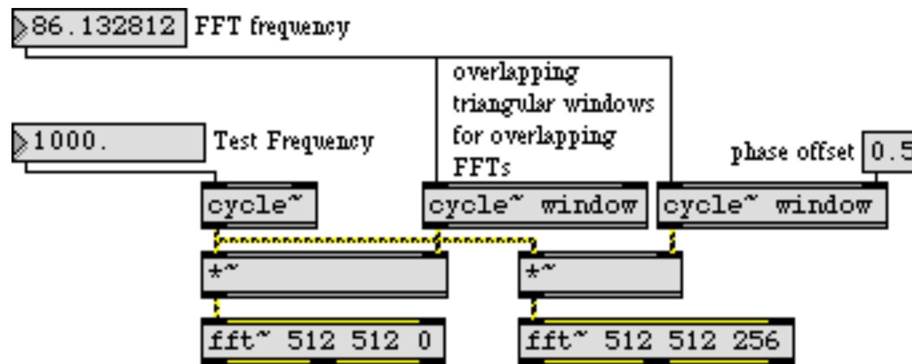
The lower right portion of the tutorial patch takes this approach of using overlapping time slices, and applies a triangular amplitude envelope to each slice before analyzing it. (Other shapes of amplitude envelope are often used for this process. The triangular window is simple and quite effective.) In this way, the **fft~** object is viewing each time slice through a triangular window which tapers its ends down, thus filtering out many of the false frequencies that would be introduced by discontinuities.



Overlapping triangular windows (envelopes) applied to a 100 Hz cosine wave

To accomplish this windowing and overlapping of time slices, we must perform two FFTs, one of which is offset 256 samples later than the other. (Note that this part of the patch will only work if your current MSP Signal Vector size is 256 or less, since **fft~** can only be offset by a multiple of the

vector size.) The offset of an FFT can be given as a (third) typed-in argument to **fft~**, as is done for the **fft~** object on the right. This results in overlapping time slices.



One FFT is taken 256 samples later than the other

The windowing is achieved by multiplying the signal by a triangular waveform (stored in the **buffer~** object) which recurs at the same frequency as the FFT—once every 512 samples. The window is offset by $^1/_2$ cycle (256 samples) for the second **fft~**.

- Double-click on the **buffer~** object to view its contents. Then close the **buffer~** window and double-click on the **capture~** object that contains the FFT of the windowed signal. Scroll past the first block or two of numbers until you see the FFT analysis of the windowed 1000 Hz tone.

As with the unwindowed FFT, the energy is greatest around 1000 Hz, but here the (spurious) energy in all the other frequency regions is greatly reduced by comparison with the unwindowed version.

Signal processing using the FFT

In this patch we have used the **fft~** object to view and analyze a signal, and to demonstrate the effectiveness of windowing the signal and using overlapping FFTs. However, one could also write a patch that alters the values in the signal coming out of **fft~**, then sends the altered analysis to **ifft~** for resynthesis. An implementation of this frequency-domain filtering scheme will be seen in a future tutorial.

Summary

The fast Fourier transform (FFT) is an algorithm for transforming a time-domain digital signal into a frequency-domain representation of the relative amplitude of different frequency regions in the signal. An FFT is computed using a relatively small excerpt of a signal, usually a slice of time 512 or 1024 samples long. To analyze a longer signal, one performs multiple FFTs using consecutive (or overlapping) time slices.

The **fft~** object performs an FFT on the signal it receives, and sends out (also in the form of a signal) a frequency-domain analysis of the received signal. The only object that understands the output of **fft~** is **ifft~** which performs an inverse FFT to synthesize a time-domain signal based on the

frequency-domain information. One could alter the signal as it goes from **fft~** to **ifft~**, in order to change the spectrum.

The FFT only works perfectly when analyzing exactly one cycle (or exactly an integer number of cycles) of a tone. To reduce the artifacts produced when this is not the case, one can window the signal being analyzed by applying an amplitude envelope to taper the ends of each time slice. The amplitude envelope can be applied by multiplying the signal by using a **cycle~** object to read a windowing function from a **buffer~** repeatedly at the same rate as the FFT itself (i.e., once per time slice).

See Also

buffer~	Store audio samples
capture~	Store a signal to view as text
fft~	Fast Fourier transform
ifft~	Inverse Fast Fourier transform

Tutorial 26

Frequency Domain Signal Processing with **pfft~**

Working in the Frequency Domain

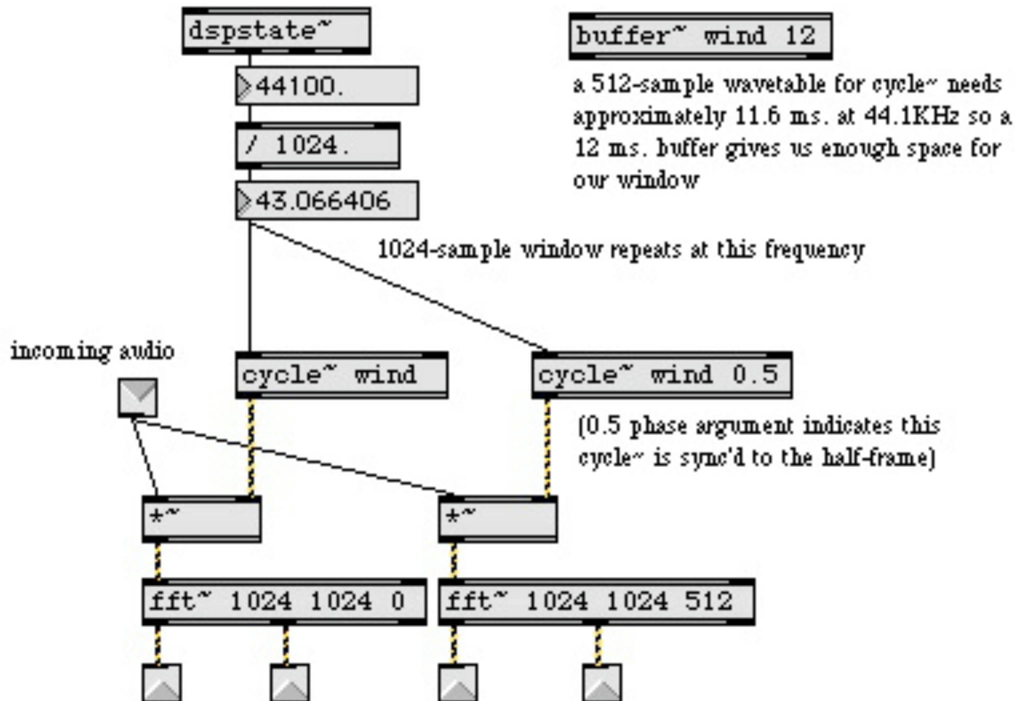
Most digital signal processing of audio occurs in what is known as the time domain. As the other MSP tutorials show you, many of the most common processes for manipulating audio consist of varying samples (or groups of samples) in amplitude (ring modulation, waveshaping, distortion) or time (filters and delays). The Fast Fourier Transform (FFT) allows you to translate audio data from the time domain into the frequency domain, where you can directly manipulate the spectrum of a sound (the component frequencies of a slice of audio).

As we have seen in Tutorial 25, the MSP objects **fft~** and **ifft~** allow you to transform signals into and out of the frequency domain. The **fft~** object takes a a group of samples (commonly called a frame) and transforms them into pairs of real and imaginary numbers representing the amplitude and phase of as many frequencies as there are samples in the frame. These are usually referred to as *bins* or *frequency bins.* (We will see later that the real and imaginary numbers are not themselves the amplitude and phase, but that the amplitude and phase can be derived from them.) The **ifft~** object performs the inverse operation, taking frames of frequency-domain samples and converting them back into a time domain audio signal that you can listen to or process further. The number of samples in the frame is called the *FFT size* (or sometimes *FFT point size*). It must be a power of 2 such as 512, 1024 or 2048 (to give a few commonly used values).

One of the shortcomings of the **fft~** and **ifft~** objects is that they work on successive frames of samples without doing any overlapping or cross-fading between them. For most practical musical uses of these objects, we usually need to construct such an overlap and crossfade system around them. There are several reasons for needing to create such a system when using the Fourier transform to process sound. In FFT analysis there is always a trade-off between frequency resolution and timing resolution. For example, if your FFT size is 2048 samples long, the FFT analysis gives you 2048 equally-spaced frequency bins from 0 Hz. up to the sampling frequency (only 1024 of these bins are of any use; see Tutorial 25 for details). However, any timing resolution that occurs within those 2048 samples will be lost in the analysis, since all temporal changes are lumped together in a single FFT frame. In addition, if you modify the spectral data after the FFT analysis and before the IFFT resynthesis you can no longer guarantee that the time domain signal output by the IFFT will match up in successive frames. If the output time domain vectors don't fit together you will get clicks in your output signal. By designing a *windowing function* in MSP (see below), you can compensate for these artifacts by having successive frames cross-fade into each other as they overlap. While this will not compensate for the loss of time resolution, the overlapping of analysis data will help to eliminate the clicks and pops that occurs at the edges of an IFFT frame after resynthesis.

This analysis/resynthesis scheme (using overlapping, windowed slices of time with the FFT and IFFT) is usually referred to as a *Short Term* (or *Short Time*) *Fourier Transform* (STFT). An STFT can be designed in MSP by creating a patch that uses one or more pairs of **fft~/ifft~** objects with the input signal "windowed" into and out of the frequency domain. While this approach works fairly well, it is somewhat cumbersome to program since every operation performed in the fre-

quency domain needs to be duplicated correctly for each **fft~/ifft~** pair. The following subpatch illustrates how one would window incoming FFT data in this manner:



How to properly window audio for use with the **fft~** *object*

In addition to the fact that this approach can often be a challenge to program, there is also the difficulty of generalizing the patch for multiple combinations of FFT size and overlap. Since the arguments to **fft~/ifft~** for FFT frame size and overlap can't be changed, multiple hand-tweaked versions of each subpatch must be created for different situations. For example, a percussive sound

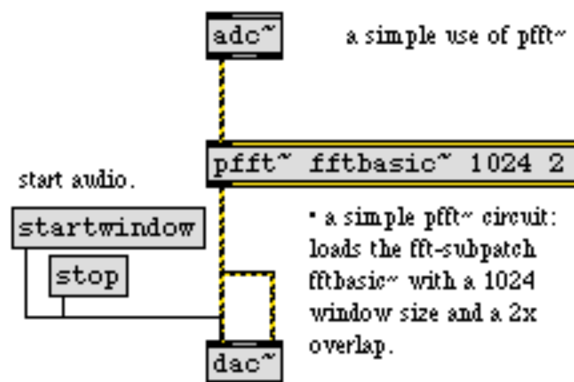
would necessitate an analysis with at least four overlaps, while a reasonably static, harmonically rich sound would call for a very large FFT size.

Technical detail: Time vs. Frequency Resolution

The FFT size we use provides us with a tradeoff; because the Fourier transform mathematically converts a small slice of time into a frozen "snapshot" representing its spectrum, you might first think that it would be beneficial to use small FFT sizes in order to avoid grouping temporal changes together in one analysis spectrum. While this is true, an FFT size with a smaller number of points also means that our spectrum will have a smaller number of frequency bins, which means that the frequency resolution will be lower. Smaller FFT sizes result in better temporal resolution, but at the cost of lower frequency resolution when the sound is modified in the frequency domain and resynthesized. Conversely, larger FFT sizes give us finer frequency detail, but tend to "smear" temporal changes in the sound. In practice, we therefore need to choose an appropriate FFT size based on the kind of sound we want to process.

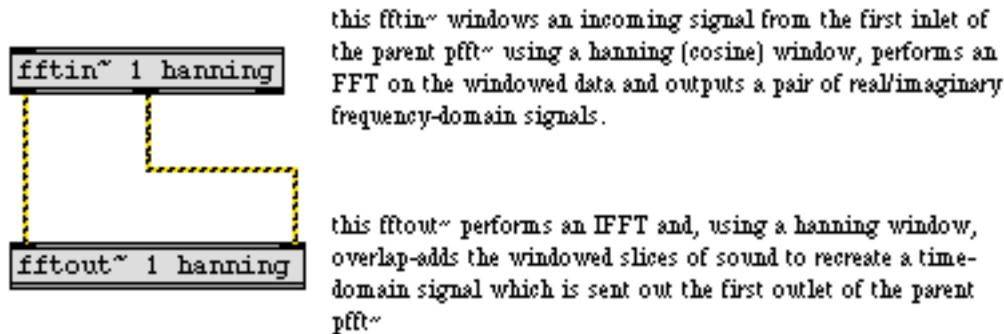
The **pfft~** object addresses many of the shortcomings of the "old" **fft~** and **ifft~** objects, allowing you to create and load special "spectral subpatches" that manipulate frequency-domain signal data independently of windowing, overlap and FFT size. A single sub-patch can therefore be suitable for multiple applications. Furthermore, the **pfft~** object manages the overlapping of FFT frames, handles the windowing functions for you, and eliminates the redundant mirrored data in the spectrum, making it both more convenient to use and more efficient than the traditional **fft~** and **ifft~** objects.

The **pfft~** object takes as its argument the name of a specially designed subpatch containing the **fftin~** and **fftout~** objects (which will be discussed below), a number for the FFT size in samples, and a number for the overlap factor (these must both be integers which are a power of 2):



A simple use of **pfft~**.

The **pfft~** subpatch fftbasic~ referenced above might look something like this:



The **fftbasic~** *subpatch used in the previous example*

The **fftbasic~** subpatch shown above takes a signal input, performs an FFT on that signal with a Hanning window (see below), and performs an IFFT on the FFT'd signal, also with a Hanning window. The **pfft~** object communicates with its sub-patch using special objects for inlets and outlets. The **fftin~** object receives a time-domain signal from its parent patch and transforms it via an FFT into the frequency domain. This time-domain signal has already been converted, by the **pfft~** object into a sequence of frames which overlap in time, and the signal that **fftin~** outputs into the spectral subpatch represents the spectrum of each of these incoming frames.

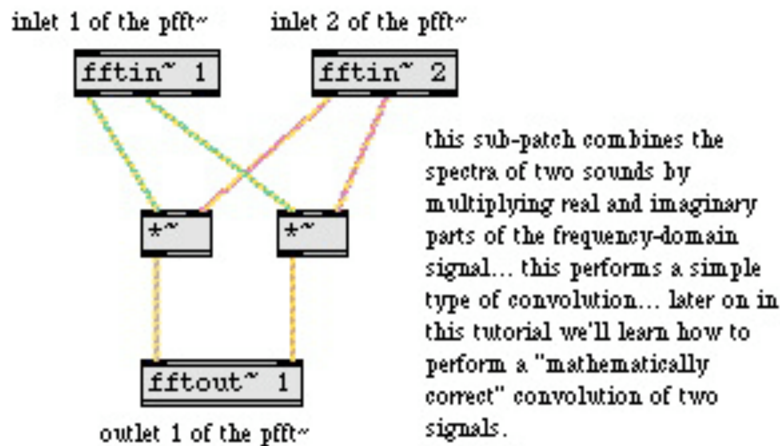
> **Technical detail:** The signal vector size inside the spectral subpatch is equal to half the FFT size specified as an argument to the **pfft~**. Here's the reason why: for efficiency's sake, **fftin~** and **fftout~** perform what is known as a *real FFT*, which is faster than the traditional *complex FFT* used by **fft~** and **ifft~**. This is possible because the time-domain signals we transform have no imaginary part (or at least they have an imaginary part which is equal to zero). A real FFT is a clever mathematical trick which re-arranges the real-only time-domain input to the FFT as real and imaginary parts of a complex FFT that is half the size of our real FFT. The result of this FFT is then re-arranged into a complex spectrum representing half (from 0Hz to half the sampling rate) of our original real-only signal. The smaller FFT size means it is more efficient for our computer's processor, and, because a complex FFT produces a mirrored spectrum of which only half is really useful to us, the real FFT contains all the data we need to define and subsequently manipulate the signal's spectrum.

The **fftout~** object does the reverse, accepting frequency domain signals, converting them back into a time domain signal, and passing it via an outlet to the parent patch. Both objects take a numbered argument (to specify the inlet or outlet number), and a symbol specifying the window function to use. The available window functions are Hanning (the default if none is specified), Hamming, Blackman, Triangle, and Square. The nofft argument to **fftin~** and **fftout~** creates a generic signal inlet or outlet for control data where no FFT/IFFT or windowing is performed. In addition, the symbol can be the name of a **buffer~** object which holds a custom windowing function. Different window functions have different bandwidths and stopband depths for each channel (or bin, as it is sometimes called) of the FFT. A good reference on FFT analysis will help you

select a window based on the sound you are trying to analyze and what you want to do with it. We recommend *The Computer Music Tutorial* by Curtis Roads or *Computer Music* by Charles Dodge and Thomas Jerse.

For testing and debugging purposes, there is a handy nofft argument to **fftin~** and **fftout~** which allows the overlapping time-domain frames to and from the **pfft~** to be passed directly to and from the subpatch without applying a window function nor performing a Fourier transform. In this case (because the signal vector size of the spectral subpatch is half the FFT size), the time-domain signal is split between the real and imaginary outlets of the **fftin~** and **fftout~** objects, which may be rather inconvenient when using an overlap of 4 or more. Although the nofft option can be used to send signal data from the parent patch into the spectral subpatch and may be useful for debugging subpatches, it is not recommended for most practical uses of **pfft~**.

A more complicated **pfft~** subpatch might look something like this:



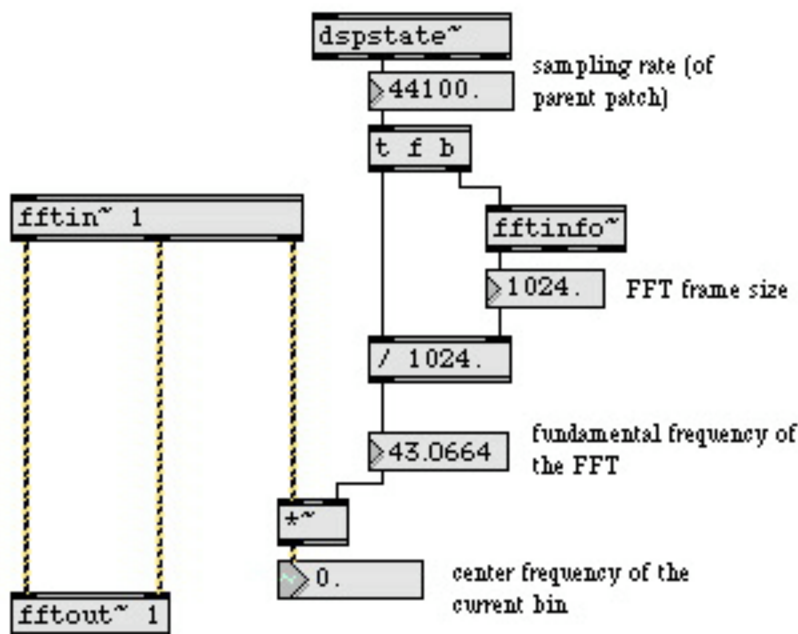
A simple type of spectral convolution

This subpatch takes two signal inputs (which would appear as inlets in the parent **pfft~** object), converts them into the frequency domain, multiplies the real signals with one another and multiplies the imaginary signals with one another and outputs the result to an **fftout~** object that converts the frequency domain data into a time domain signal. Multiplication in the frequency domain is called *convolution*, and is the basic signal processing procedure used in cross synthesis (morphing one sound into another). The result of this algorithm is that frequencies from the two analyses with larger values will reinforce one another, whereas weaker frequency values from one analysis will diminish or cancel the value from the other, whether strong or weak. Frequency content that the two incoming signals share will be retained, therefore, and disparate frequency content (i.e. a pitch that exists in one signal and not the other) will be attenuated or eliminated. This example is not a “true” convolution, however, as the multiplication of *complex numbers* (see below) is not as straightforward as the multiplication performed in this example. We'll learn a couple ways of making a “correct” convolution patch later in this tutorial.

You have probably already noticed that there are always two signals to connect when connecting **fftin~** and **fftout~**, as well as when processing the spectra in-between them. This is because the FFT algorithm produces *complex numbers* — numbers that contain a real and an imaginary part.

The real part is sent out the leftmost outlet of **fftin~**, and the imaginary part is sent out its second outlet. The two inlets of **fftout~** also correspond to real and imaginary, respectively. The easiest way to understand complex numbers is to think of them as representing a point on a 2-dimensional plane, where the real part represents the X-axis (horizontal distance from zero), and the imaginary part represents the Y-axis (vertical distance from zero). We'll learn more about what we can do with the real and imaginary parts of the complex numbers later on in this tutorial.

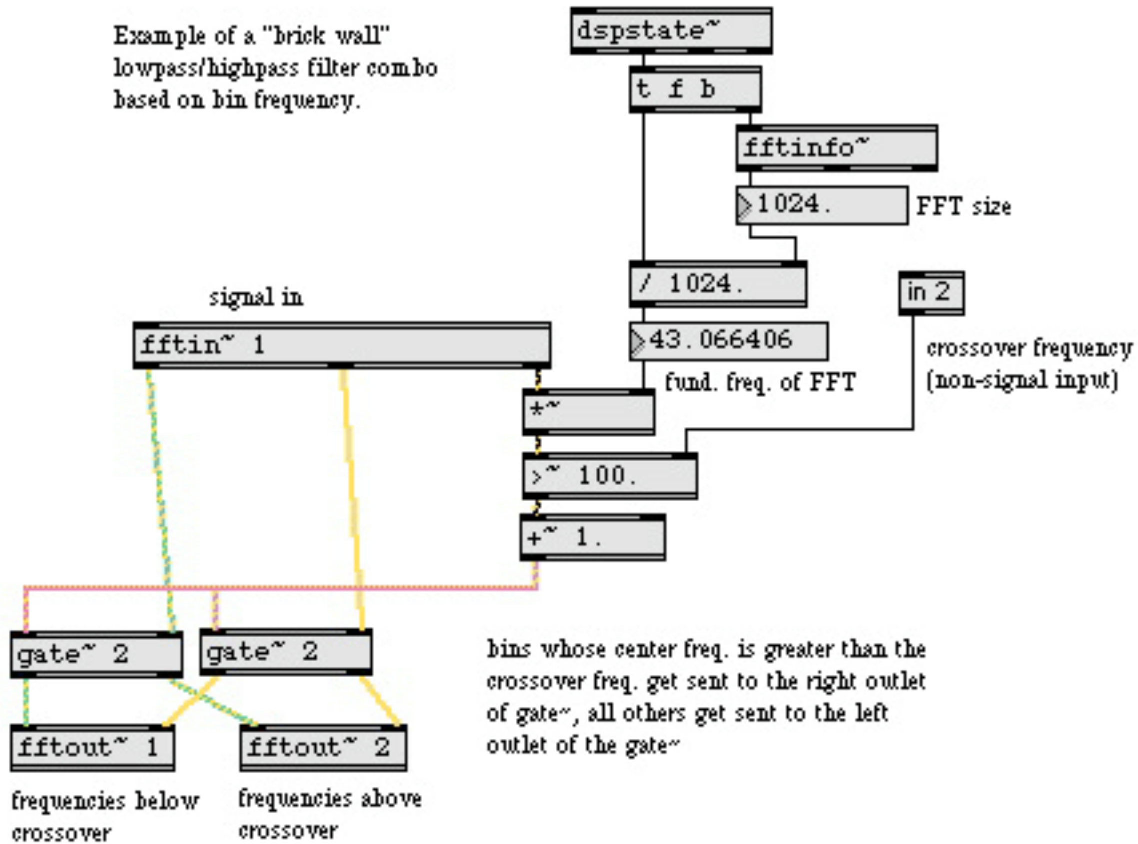
The **fftin~** object has a third outlet that puts out a stream of samples corresponding to the current frequency bin index whose data is being sent out the first two outlets (this is analogous to the third outlet of the **fft~** and **ifft~** objects discussed in Tutorial 25). For **fftin~**, this outlet outputs a number from 0 to half the FFT size minus 1. You can convert these values into frequency values (representing the "center" frequency of each bin) by multiplying the signal (called the sync signal) by the base frequency, or fundamental, of the FFT. The fundamental of the FFT is the lowest frequency that the FFT can analyze, and is inversely proportional to the size of the FFT (i.e. larger FFT sizes yield lower base frequencies). The exact fundamental of the FFT can be obtained by dividing the FFT frame size into the sampling rate. The **fftinfo~** object, when placed into a **pfft~** subpatch, will give you the FFT frame size, the FFT half-frame size (i.e. the number of bins actually used inside the **pfft~** subpatch), and the FFT hop size (the number of samples of overlap between the windowed frames). You can use this in conjunction with the **dspstate~** object or the **adstatus** object with the sr (sampling rate) argument to obtain the base frequency of the FFT:



Finding the center frequency of the current analysis bin.

Note that in the above example the **number~** object is used for the purposes of demonstration only in this tutorial. When DSP is turned on, the number displayed in the signal number box will not appear to change because the signal number box by default displays the first sample in the signal vector, which in this case will always be 0. To see the center frequency values, you will need to use the **capture~** object or record this signal into a **buffer~**.

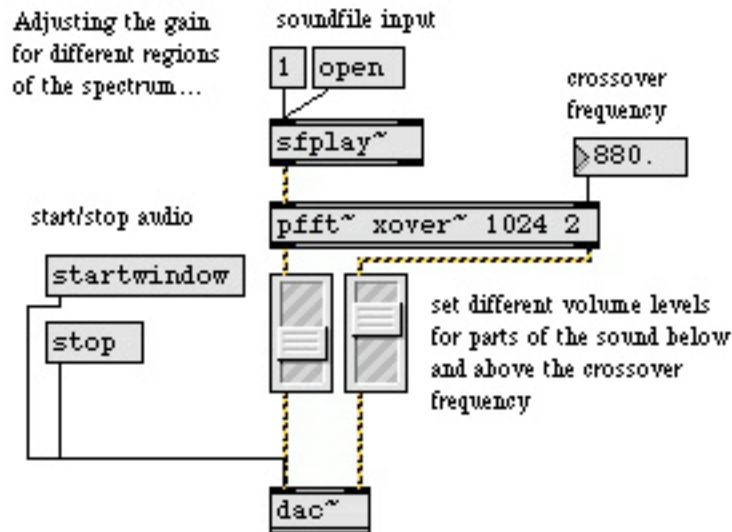
Once you know the frequency of the bins being streamed out of **fftin~**, you can perform operations on the FFT data based on frequency. For example:



A simple spectral crossover.

The above **pfft~** subpatch, called **xover~**, takes an input signal and sends the analysis data to one of two **fftout~** objects based on a crossover frequency. The crossover frequency is sent to the **pfft~** subpatch by using the in object, which passes max messages through from the parent patch via the **pfft~** object's right inlet. The center frequency of the current bin — determined by the sync outlet in conjunction with **fftinfo~** and **dspstate~** as we mentioned above — is compared with the crossover frequency. The result of this comparison flips a gate that sends the FFT data to one of the two **fftout~** objects: the part of the spectrum that is lower in pitch than the crossover frequency is sent

out the left outlet of the **pfft~** and the part that is higher than the crossover frequency is sent out the right. Here is how this subpatcher might be used with **pfft~** in a patch:

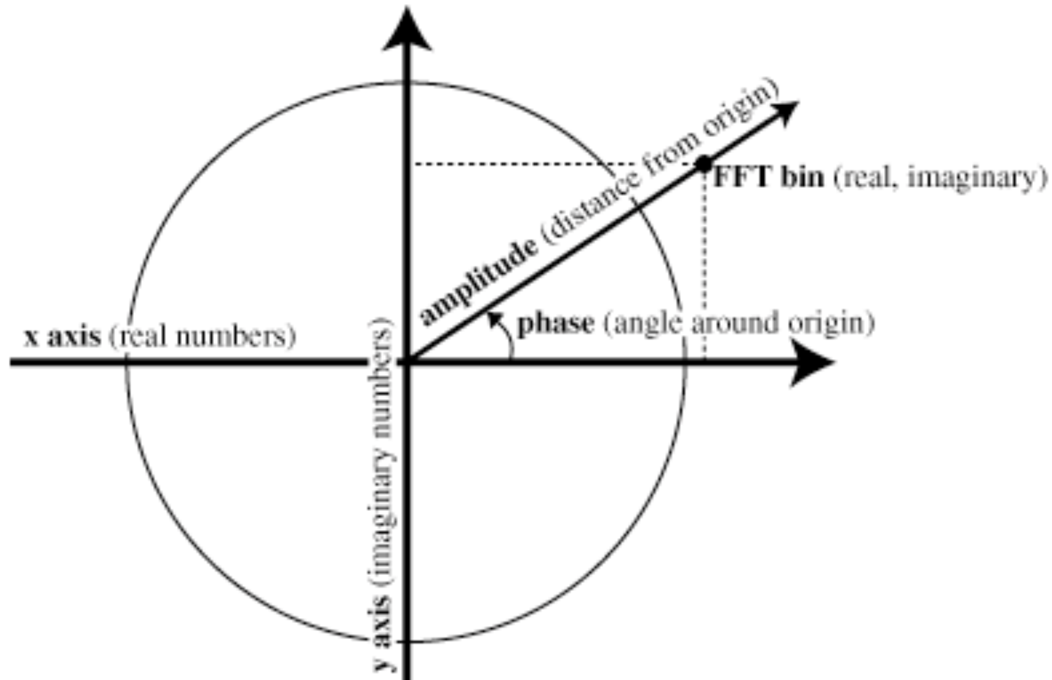


One way of using the xover~ *subpatch*

Note that we can send integers, floats, and any other Max message to and from a subpatch loaded by **pfft~** by using the **in** and **out** objects. (See *Tutorial 21, Using the* **poly~** *object* for details. Keep in mind, however, that the signal objects **in~** and **out~** currently do not function inside a **pfft~**.)

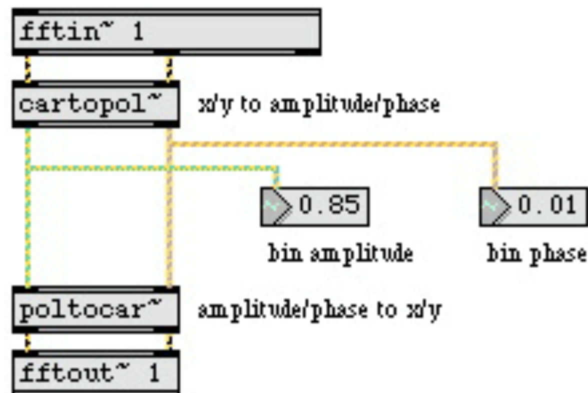
As we have already learned, the first two outlets of **fftin~** put out a stream of real and imaginary numbers for the bin response for each sample of the FFT analysis (similarly, **fftout~** expects these numbers). These are not the amplitude and phase of each bin, but should be thought of instead as pairs of Cartesian coordinates, where *x* is the real part and *y* is the imaginary, representing points on a 2-dimensional plane. The amplitude and phase of each frequency bin are the polar coordi-

nates of these points, where the distance from the origin is the bin amplitude and the angle around the origin is the bin phase:



Unit-circle diagram showing the relationship of FFT real and imaginary values to amplitude and phase

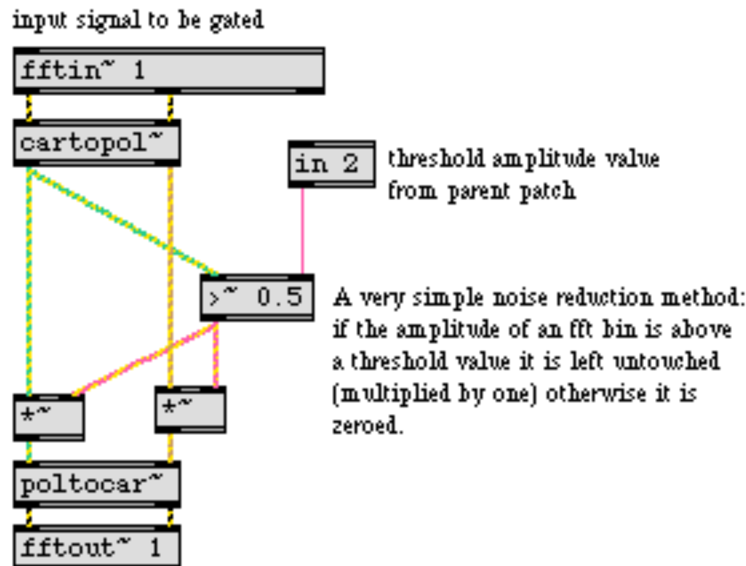
You can easily convert between real/imaginary pairs and amplitude/phase pairs using the objects **cartopol~** and **poltocar~**:



Cartesian to polar conversion

> **Technical detail:** The amplitude values output by the left outlet of **cartopol~** depend on the amplitude of the signal you send to the **pfft~** object. Due to the way **fftin~** and **fftout~** automatically scale their window functions (in order to maintain the same output amplitude after overlap-adding), the maximum amplitude value for a constant signal of 1.0 will be
>
> *(FFT size / (sqrt(sum of points in the window/hop size))*
>
> So, when using a 512-point FFT with a square window with an overlap of 2, the maximum possible amplitude value will be roughly 362, with 4-overlap it will be 256. When using a hanning or hamming window and 2 overlap, it will be approximately 325 or 341, and with 4-overlap, it will be 230 or 241, respectively. Generally, however, the peak amplitudes in a spectral frame will most likely be only one-fourth to half this high for non-periodic or semi-periodic "real-world" sounds normalized between -1.0 and 1.0.
>
> The phase values output by the right outlet of **cartopol~** will always be between -π and π.

You can use this information to create signal processing routines based on amplitude/phase data. A spectral noise gate would look something like this:

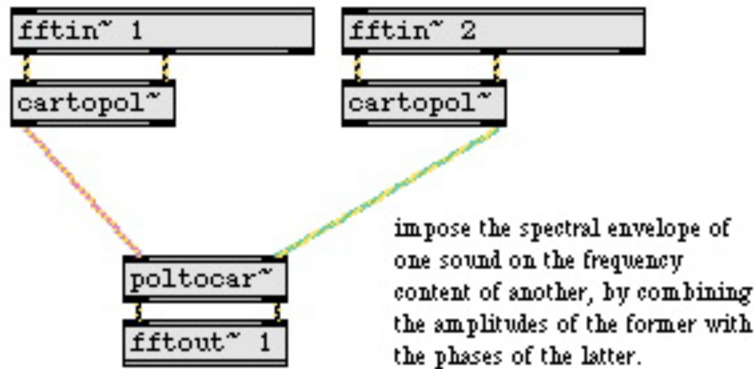


A spectral noise gate

By comparing the amplitude output of **cartopol~** with the threshold signal sent into inlet 2 of the **pfft~**, each bin is either passed or zeroed by the *~ objects. This way only frequency bins that exceed a certain amplitude are retained in the resynthesis (For information on amplitude values inside a spectral subpatch, see the Technical note above.).

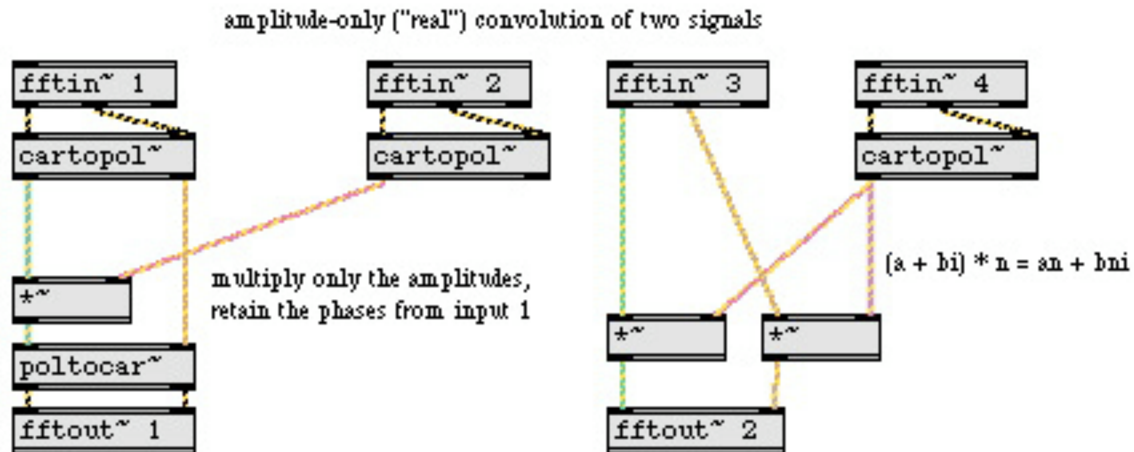
Convolution and cross-synthesis effects commonly use amplitude and phase data for their processing. One of the most basic cross-synthesis effects we could make would use the amplitude

spectrum of one sound with the phase spectrum of another. Since the phase spectrum is related to information about the sound's frequency content, this kind of cross synthesis can give us the harmonic content of one sound being "played" by the spectral envelope of another sound. Naturally, the success of this type of effect depends heavily on the choice of the two sounds used. Here is an example of a spectral subpatch which makes use of **cartopol~** and **poltocar~** to perform this type of cross-synthesis:



Simple cross-synthesis

The following subpatch example shows two ways of convolving the amplitude of one input with the amplitude of another:

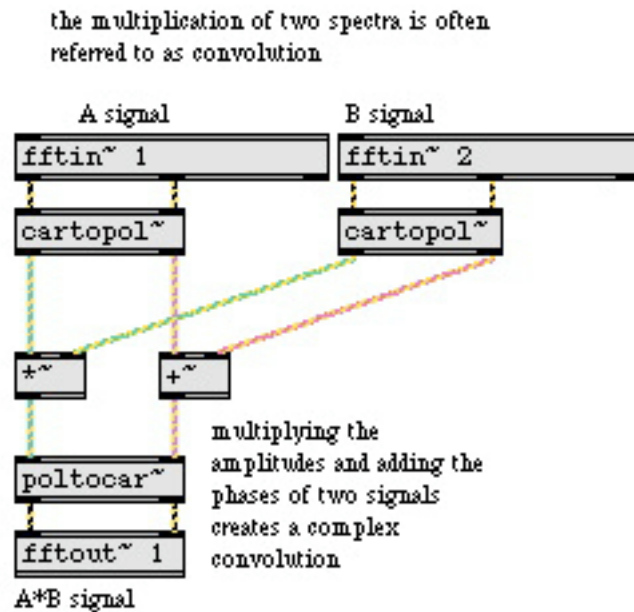


Amplitude-only convolution

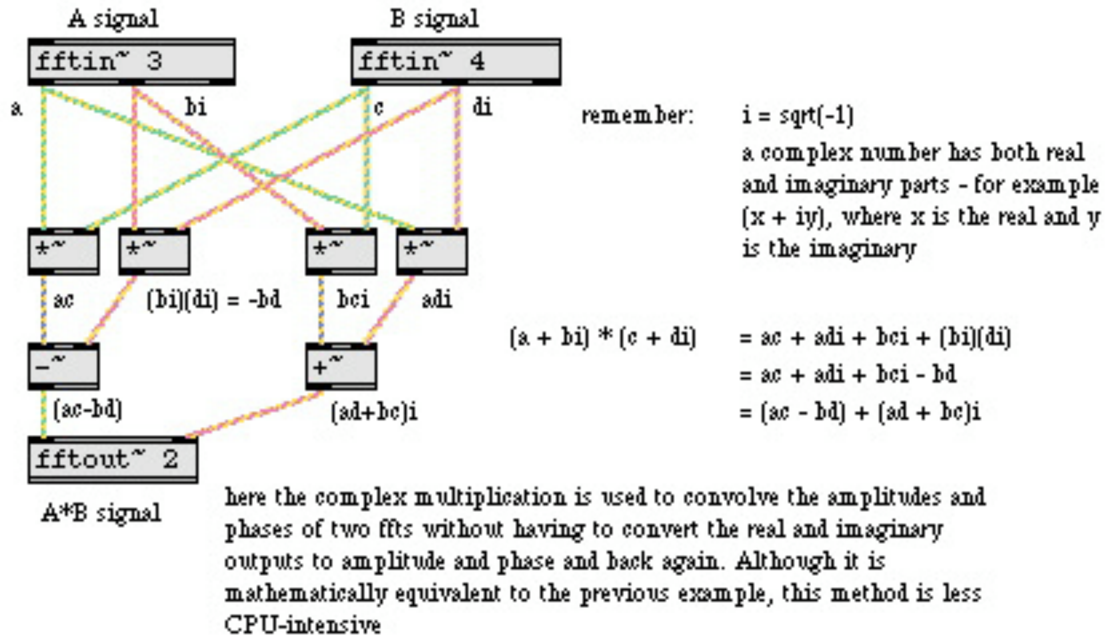
You can readily see on the left-hand side of this subpatch that the amplitude values of the input signals are multiplied together. This reinforces amplitudes which are prominent in both sounds while

attenuating those which are not. The phase response of the first signal is unaffected by complex-real multiplication; the phase response of the second signal input is ignored. You will also notice that the right-hand side of the subpatch is mathematically equivalent to the left, even though it uses only one **cartopol~** object.

Toward the beginning of this tutorial, we saw an example of the multiplication of two real/imaginary signals to perform a convolution. That example was kept simple for the purposes of explanation but was, in fact, incorrect. If you wondered what a "correct" multiplication of two complex numbers would entail, here are two ways to do it:



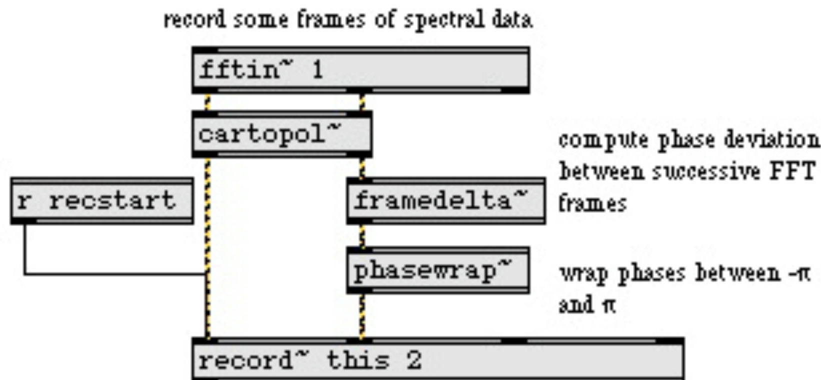
The correct method for doing complex convolution



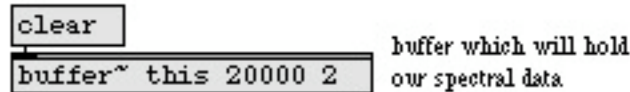
A correct and clever way of doing complex convolution

Subpatchers created for use with **pfft~** can use the full range of MSP objects, including objects that access data stored in a **buffer~** object. (Although some objects which were designed to deal with timing issues may not always behave as initially expected when used inside a **pfft~**.) The following

example records spectral analysis data into two channels of a stereo **buffer~** and then allows you to resynthesize the recording at a different speed.

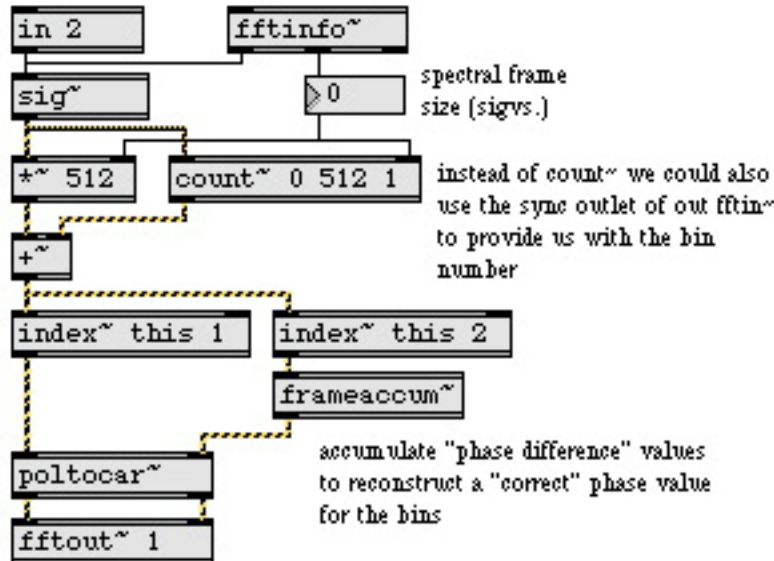


notice that we're recording amplitude and "phase difference" which can be used to reconstruct a correct phase value when we play back the spectral frames at a different speed...



20000 ms. = 882000 samples at 44.1KHz = 1722 frames of spectra at FFTsize=1024 (512 sigvs. for this subpatch)

playback frames at any rate



Recording and playback in a **pfft~** *subpatch*

The example subpatcher records spectral data into a **buffer~**, and the second reads data from that **buffer~**. In the recording portion of the subpatch you will notice that we don't just record the

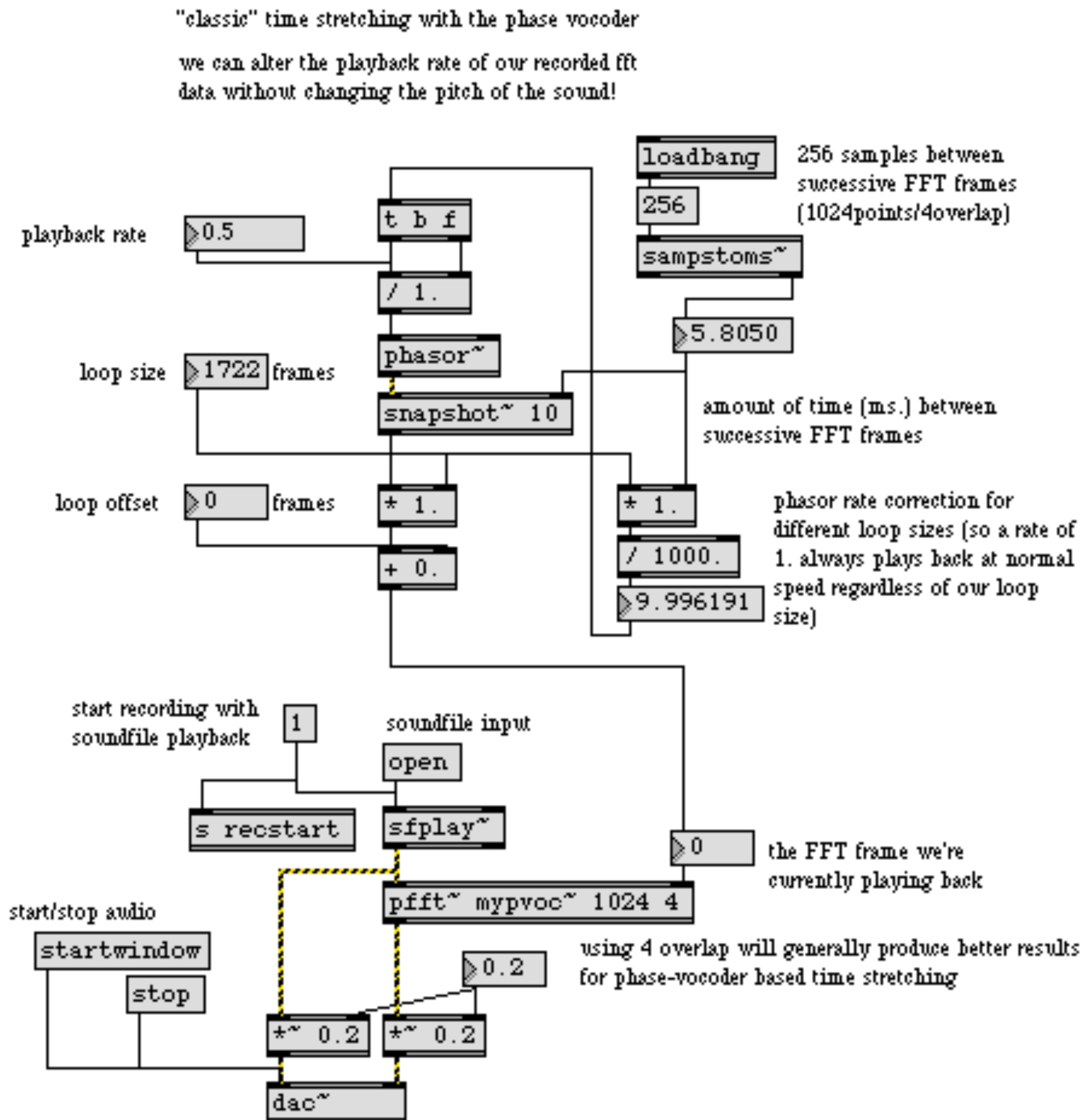
amplitude and phase as output from **cartopol~**, but instead use the **framedelta~** object to compute the phase difference (sometimes referred to as the phase deviation, or phase derivative). The phase difference is quite simply the difference in phase between equivalent bin locations in successive FFT frames. The output of **framedelta~** is then fed into a **phasewrap~** object to ensure that the data is properly constrained between $-\pi$ and π. Messages can be sent to the **record~** object from the parent patch via the **send** object in order to start and stop recording and turn on looping.

In the playback part of the subpatch we use a non-signal inlet to specify the frame number for the resynthesis. This number is multiplied by the spectral frame size and added to the output of a **count~** object which counts from 0 to the spectral frame size minus 1 in order to be able to recall each frequency bin in the given frame successively using **index~** to read both channels of our **buffer~**. (We could also have used the sync outlet of the **fftin~** object in place of **count~**, but are using the current method for the sake of visually separating the recording and playback parts of our subpatch, as well as to give an example of how to make use of **count~** in the context of a spectral subpatch.) You'll notice that we reconstruct the phase using the **frameaccum~** object, which accumulates a "running phase" value by performing the inverse of **framedelta~**. We need to do this because we might not be reading the analysis frames successively at the original rate in which they were recorded. The signals are then converted back into real and imaginary values for **fftout~** by the **poltocar~** object.

This is a simple example of what is known as a *phase vocoder*. Phase vocoders allow you to time-stretch and compress signals independently of their pitch by manipulating FFT data rather than time-domain segments. If you think of each frame of an FFT analysis as a single frame in a film, you can easily see how moving through the individual frames at different rates can change the apparent speed at which things happen. This is more or less what a phase vocoder does.

Note that because **pfft~** does window overlapping, the amount of data that can be stored in the **buffer~** is dependent on the settings of the **pfft~** object. This can make setting the buffer size correctly a rather tricky matter, especially since the spectral frame size (i.e. the signal vector size) inside a **pfft~** is half the FFT size indicated as its second argument, and because the spectral subpatch is processing samples at a different rate to its parent patch! If we create a stereo **buffer~** with 1000 milliseconds of sample memory, we will have 44100 samples available for our analysis data. If our FFT size is 1024 then each spectral frame will take up 512 samples of our buffer's memory, which amounts to 86 frames of analysis data(44100 / 512 = 86.13). Those 86 frames do not represent one second of sound, however! If we are using 4-times overlap, we are processing one spectral frame every 256 samples, so 86 frames means roughly 22050 samples, or a half second's worth of time with respect to the parent patch. As you can see this all can get rather complicated…

Let's take a look at the parent patch for the above phase vocoder subpatch (called **mypvoc~**):



Wrapper for mypvoc

Notice that we're using a **phasor~** object with a **snapshot~** object in order to generate a ramp specifying the read location inside our subpatch. We could also use a **line** object, or even a slider, if we wanted to "scrub" our analysis frames by hand. Our main patch allows us to change the playback rate for a loop of our analysis data. We can also specify the loop size and an offset into our collection of analysis frames in order to loop a given section of analysis data at a given playback rate. You'll notice that changing the playback rate does *not* affect the pitch of the sound, only the speed. You may also notice that at very slow playback rates, certain parts of your sound (usually note attacks, consonants in speech or other percussive sounds) become rather "smeared" and gain an artificial sound quality.

Summary

Using **pfft~** to perform spectral-domain signal processing is generally easier and visually clearer than using the traditional **fft~** and **ifft~** objects, and lets you design patches that can be used at varying FFT sizes and overlaps. There are myriad applications of **pfft~** for musical signal processing, including filtering, cross synthesis and time stretching.

See Also

adstatus	Access audio driver output channels
cartopol~	Signal Cartesian to Polar coordinate conversion
dspstate~	Report current DSP setting
fftin~	Input for a patcher loaded by **pfft~**
fftout~	Output for a patcher loaded by **pfft~**
framedelta~	Compute phase deviation between successive FFT frames
pfft~	Spectral processing manager for patchers
phasewrap~	Wrap a signal between $-\pi$ and π
poltocar~	Signal Polar to Cartesian coordinate conversion

Tutorial 27

Processing: Delay lines

Effects achieved with delayed signals

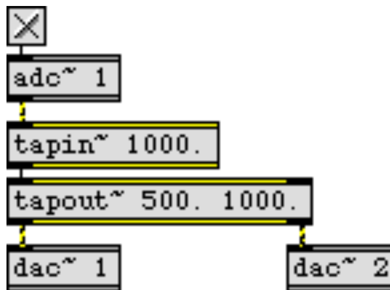
One of the most basic yet versatile techniques of audio processing is to delay a signal and mix the delayed version with the original signal. The delay time can range from a few milliseconds to several seconds, limited only by the amount of RAM you have available to store the delayed signal.

When the delay time is just a few milliseconds, the original and delayed signals interfere and create a subtle filtering effect but not a discrete echo. When the delay time is about 100 ms we hear a "slapback" echo effect in which the delayed copy follows closely behind the original. With longer delay times, we hear the two signals as discrete events, as if the delayed version were reflecting off a distant mountain.

This tutorial patch delays each channel of a stereo signal independently, and allows you to adjust the delay times and the balance between direct signal and delayed signal.

Creating a delay line: tapin~ and tapout~

The MSP object **tapin~** is a buffer that is continuously updated so that it always stores the most recently received signal. The amount of signal it stores is determined by a typed-in argument. For example, a **tapin~** object with a typed-in argument of 1000 stores the most recent one second of signal received in its inlet.



A 1-second delay buffer tapped 500 and 1000 ms in the past

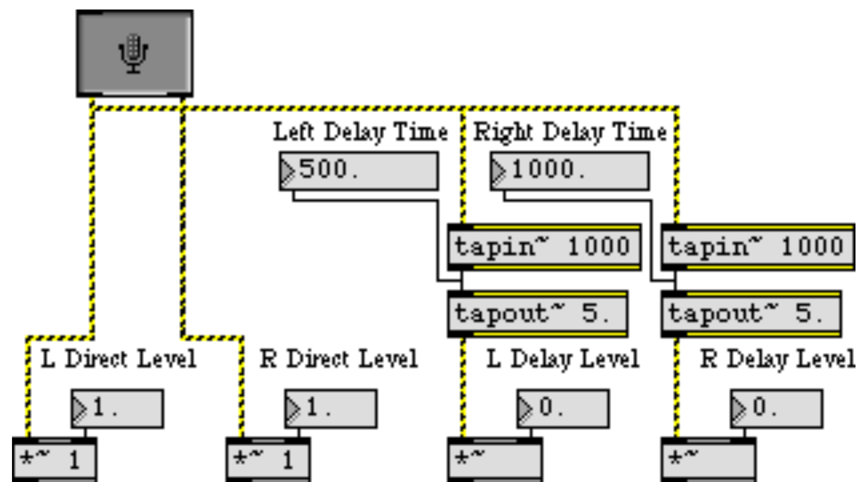
The only object to which the outlet of **tapin~** should be connected is a **tapout~** object. This connection links the **tapout~** object to the buffer stored by **tapin~**. The **tapout~** object "taps into" the delayed signal at certain points in the past. In the above example, **tapout~** gets the signal from **tapin~** that occurred 500 ms ago and sends it out the left outlet; it also gets the signal delayed by 1000 ms and sends that out its right outlet. It should be obvious that **tapout~** can't get signal delayed beyond the length of time stored in **tapin~**.

A patch for mixing original and delayed signals

The tutorial patch sends the sound coming into the computer to two places: directly to the output of the computer and to a **tapin~**-**tapout~** delay pair. You can control how much signal you hear from each place for each of the stereo channels, mixing original and delayed signal in whatever proportion you want.

- Turn audio on and send some sound in the input jacks of your computer. Set the **number box** marked "Output Level" to a comfortable listening level. Set the "Left Delay Time" **number box** to 500 and the "Right Delay Time" to 1000.

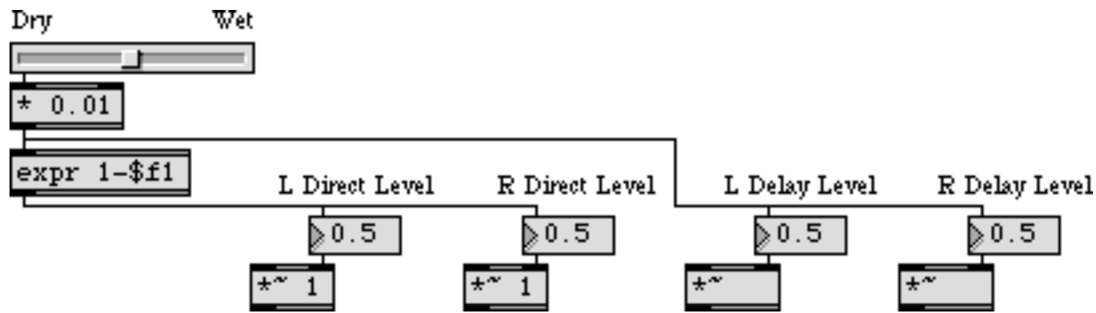
At this point you don't hear any delayed signal because the "Direct Level" for each channel is set at 1 and the "Delay Level" for each channel is set at 0. The signal is being delayed, but you simply don't hear it because its amplitude is scaled to 0.



Direct signal is on full; delayed signal is turned down to 0

The **hslider** in the left part of the Patcher window serves as a balance fader between a "Dry" (all direct) output signal and a "Wet" (fully processed) output signal.

- Drag the **hslider** to the halfway point so that both the direct and delayed signal amplitudes are at 0.5. You hear the original signal in both channels, mixed with a half-second delay in the left channel and a one-second delay in the right channel.



Equal balance between direct signal and delayed signal

- You can try a variety of different delay time combinations and wet-dry levels. Try very short delay times for subtle comb filtering effects. Try creating rhythms with the two delay times (with, for example, delay times of 375 and 500 ms).

Changing the parameters while the sound is playing can result in clicks in the sound because this patch does not protect against discontinuities. You could create a version of this patch that avoids this problem by interpolating between parameter values with **line~** or **number~** objects.

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

Summary

The **tapin~** object is a continuously updated buffer which always stores the most recently received signal. Any connected **tapout~** object can use the signal stored in **tapin~**, and access the signal from any time in the past (up to the limits of the **tapin~** object's storage). A signal delayed with **tapin~** and **tapout~** can be mixed with the undelayed signal to create discrete echoes, early reflections, or comb filtering effects.

See Also

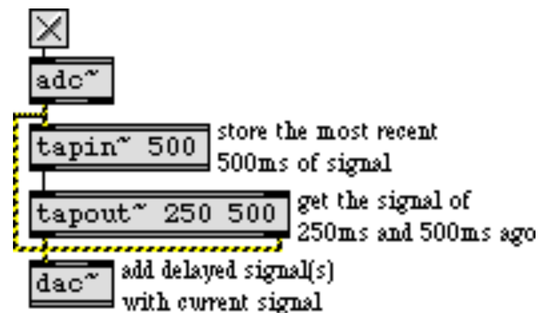
tapin~	Input to a delay line
tapout~	Output from a delay line

Tutorial 28

Processing: Delay lines with feedback

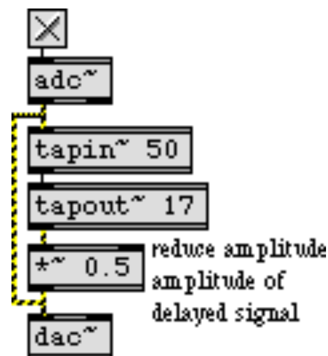
Delay emulates reflection

You can delay a signal for a specific amount of time using the **tapin~** and **tapout~** objects. The **tapin~** object is a continually updated buffer that stores the most recent signal it has received, and **tapout~** accesses that buffer at one or more specific points in the past.



Delaying a signal with **tapin~** *and* **tapout~**

Combining a sound with a delayed version of itself is a simple way of emulating a sound wave reflecting off of a wall before reaching our ears; we hear the direct sound followed closely by the reflected sound. In the real world some of the sound energy is actually absorbed by the reflecting wall, and we can emulate that fact by reducing the amplitude of the delayed sound, as shown in the following example.

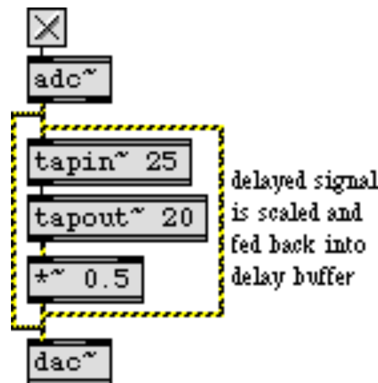


Scaling the amplitude of a delayed signal, to emulate absorption

Technical detail: Different materials absorb sound to varying degrees, and most materials absorb sound in a way that is frequency-dependent. In general, high frequencies get absorbed more than low frequencies. That fact is being ignored here.

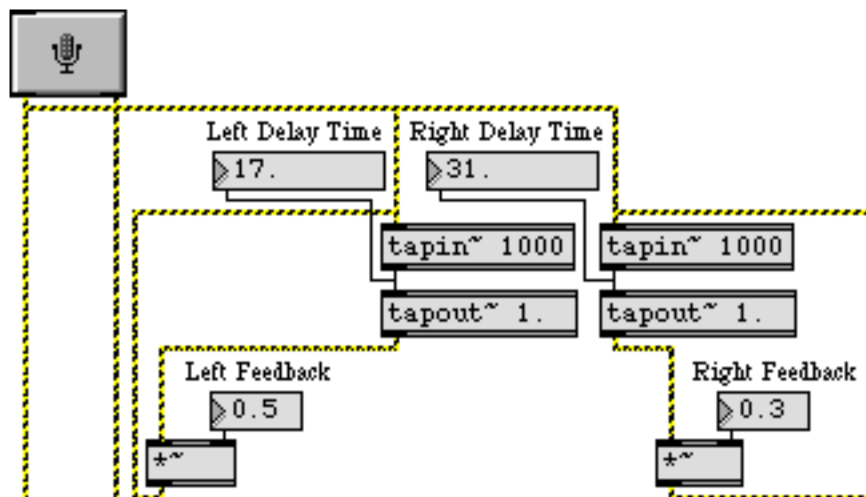
Delaying the delayed signal

Also, in the real world there's usually more than one surface that reflects sound. In a room, for example, sound reflects off of the walls, ceiling, floor, and objects in the room in myriad ways, and the reflections are in turn reflected off of other surfaces. One simple way to model this "reflection of reflections" is to feed the delayed signal back into the delay line (after first "absorbing" some of it).



Delay with feedback

A single feedback delay line like the one above is too simplistic to sound much like any real world acoustical situation, but it can generate a number of interesting effects. Stereo delay with feedback is implemented in the example patch for this tutorial. Each channel of audio input is delayed, scaled, and fed back into the delay line.



Stereo delay with individual delay times and feedback amounts

- Set the **number box** marked "Output Level" to 1., and move the **hslider** to its middle position so that the "Direct Level" and "Delay Level" **number box** objects read 0.5. Turn audio on, and send some sound into the audio input of the computer. Experiment with different delay times and feedback amounts. For example, you can use the settings shown above to achieve a blurring effect. Increase the feedback amounts for a greater resonant ringing at the rate of feedback

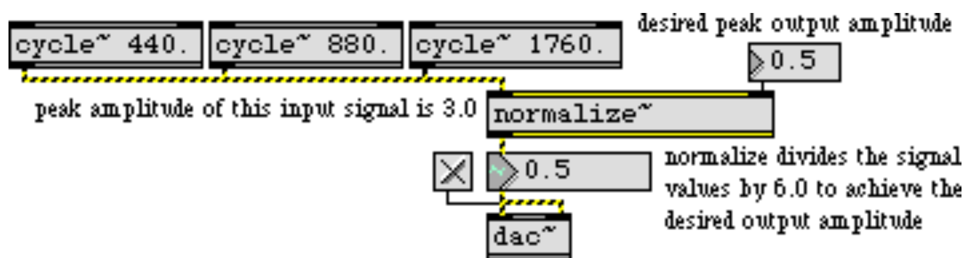
(1000 divided by the delay time). Increase the delay times to achieve discrete echoes. You can vary the Dry/Wet mix with the **hslider**.

Note that any time you feed audio signal back into a system, you have a potential for overloading the system. That's why it's important to scale the signal by some factor less than 1.0 (with the ***~** objects and the "Feedback" **number box** objects) before feeding it back into the delay line. Otherwise the delayed sound will continue indefinitely and even increase as it is added to the new incoming audio.

Controlling amplitude: normalize~

Since this patch contains user-variable level settings (notably the feedback levels) and since we don't know what sound will be coming into the patch, we can't really predict how we will need to scale the final output level. If we had used a ***~** object just before the **ezdac~** to scale the output amplitude, we could set the output level, but if we later increase the feedback levels, the output amplitude could become excessive. The **normalize~** object is good for handling such unpredictable situations.

The **normalize~** object allows you to specify a peak (maximum) amplitude that you want sent out its outlet. It looks at the peak amplitude of its input, and calculates the factor by which it must scale the signal in order to keep the peak amplitude at the specified maximum. So, with **normalize~** the peak amplitude of the output will never exceed the specified maximum.



normalize~ *sends out the current input * peak output / peak input*

One potential drawback of **normalize~** is that a single loud peak in the input signal can cause **normalize~** to scale the entire signal way down, even if the rest of the input signal is very soft. You can give **normalize~** a new peak input value to use, by sending a number or a reset message in the left inlet.

- Turn audio off and close the Patcher window before proceeding to the next chapter.

Summary

One way to make multiple delayed versions of a signal is to feed the output of **tapout~** back into the input of **tapin~**, in addition to sending it to the DAC. Because the fed back delayed signal will be added to the current incoming signal at the inlet of **tapin~**, it's a good idea to reduce the output of **tapout~** before feeding it back to **tapin~**.

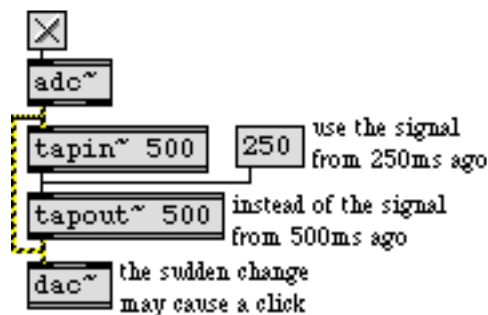
In a patch involving addition of signals with varying amplitudes, it's often difficult to predict the amplitude of the summed signal that will go to the DAC. One way to control the amplitude of a signal is with **normalize~**, which uses the peak amplitude of an incoming signal to calculate how much it should reduce the amplitude before sending the signal out.

See Also

normalize~	Scale on the basis of maximum amplitude
tapin~	Input to a delay line
tapout~	Output from a delay line

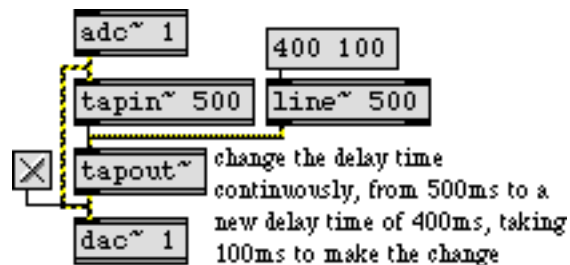
Variable delay time

So far, we have been delaying signals for a fixed amount of time using **tapin~** and **tapout~**. You can change the delay time of any tap in the **tapout~** object by sending a new number in the proper inlet; however, this will cause a discontinuity in the output signal at the instant when then new delay time is received, because **tapout~** suddenly begins tapping a new location in the **tapin~** buffer.



Changing the delay time creates a discontinuity in the output signal

On the other hand, it's possible to provide a new delay time to **tapout~** using a continuous signal instead of a discrete Max message. We can use the **line~** object to make a continuous transition between two delay times (just as we did to make continuous changes in amplitude in *Tutorial 2*).



Providing delay time in the form of a signal

Technical detail: Note that when the delay time is being changed by a continuous signal, **tapout~** has to interpolate between the old delay time and the new delay time for every sample of output. Therefore, a **tapout~** object has to do much more computation whenever a signal is connected to one of its inlets.

While this avoids the click that could be caused by a sudden discontinuity, it does mean that the pitch of the output signal will change while the delay time is being changed, emulating the *Doppler effect*.

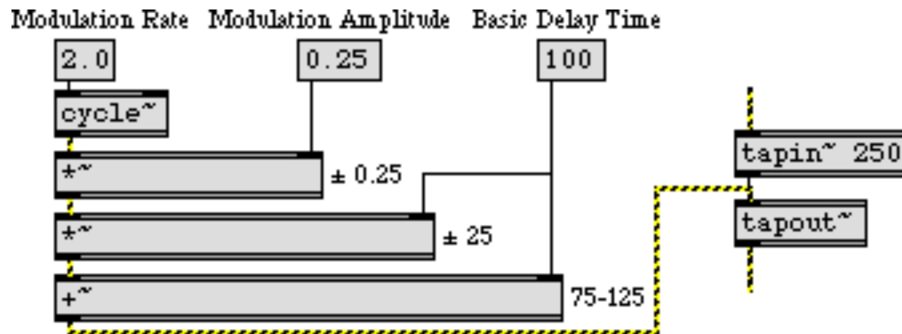
> **Technical detail:** The Doppler effect occurs when a sound source is moving toward or away from the listener. The moving sound source is, to some extent, outrunning the wavefronts of the sound it is producing. That changes the frequency at which the listener receives the wavefronts, thus changing the perceived pitch. If the sound source is moving toward the listener, wavefronts arrive at the listener with a slightly greater frequency than they are actually being produced by the source. Conversely, if the sound source is moving away from the listener, the wavefronts arrive at the listener slightly less frequently than they are actually being produced. The classic case of Doppler effect is the sound of an ambulance siren. As the ambulance passes you, it changes from moving toward you (producing an increase in received frequency) to moving away from you (producing a decrease in received frequency). You perceive this as a swift drop in the perceived pitch of the siren.
>
> A delayed signal emulates a reflection of the sound wave. As the delay time decreases, it is as if the (virtual) reflecting wall were moving toward you. The source of the delayed sound (the reflecting wall) is “moving toward you”, causing an increase in the received frequency of the sound. As the delay time increases, the reverse is true; the source of the delayed sound is effectively moving away from you. That is why, during the time when the delay time is actually changing, the perceived pitch of the output sound changes.

A delayed signal emulates a reflection of the sound wave. As the delay time decreases, it is as if the (virtual) reflecting wall were moving toward you. The source of the delayed sound (the reflecting wall) is “moving toward you”, causing an increase in the received frequency of the sound. As the delay time increases, the reverse is true; the source of the delayed sound is effectively moving away from you. That is why, during the time when the delay time is actually changing, the perceived pitch of the output sound changes.

A pitch shift due to Doppler effect is usually less disruptive than a click that’s caused by discontinuity of amplitude. More importantly, the pitch variance that results from continuously varying the delay time can be used to create some interesting effects.

Flanging: Modulating the delay time

Since the delay time can be provided by any signal, one possibility is to use a time-varying signal like a low-frequency cosine wave to modulate the delay time. In the example below, a **cycle~** object is used to vary the delay time.



Modulating the delay time with a low-frequency oscillator

The output of **cycle~** is multiplied by 0.25 to scale its amplitude. That signal is multiplied by the basic delay time of 100 ms, to create a signal with an amplitude ±25. When that signal is added to the basic delay time, the result is a signal that varies sinusoidally around the basic delay time of 100, going as low as 75 and as high as 125. This is used to express the delay time in milliseconds to the **tapout~** object.

When a signal with a time-varying delay (especially a very short delay) is added together with the original undelayed signal, the result is a continually varying comb filter effect known as *flanging*. Flanging can create both subtle and extreme effects, depending on the rate and depth of the modulation.

Stereo flange with feedback

This tutorial patch is very similar to that of the preceding chapter. The primary difference here is that the delay times of the two channels are being modulated by a cosine wave, as was described on the previous page. This patch gives you the opportunity to try a wide variety of flanging effects, just by modifying the different parameters: the wet/dry mix between delayed and undelayed signal, the left and right channel delay times, the rate and depth of the delay time modulation, and the amount of delayed signal that is fed back into the delay line of each channel.

- Send some sound into the audio input of the computer, and click on the buttons of the **preset** object to hear different effects. Using the example settings as starting points, experiment with different values for the various parameters. Notice that the modulation depth can also be controlled by the mod wheel of your synth, demonstrating how MIDI can be used for realtime control of audio processing parameters.

The different examples stored in the **preset** object are characterized below.

1. Simple thru of the audio input to the audio output. This is just to allow you to test the input and output.

2. The input signal is combined equally with delayed versions of itself, using short (mutually prime) delay times for each channel. The rate of modulation is set for 0.2 Hz (one sinusoid every 5 seconds), but the depth of modulation is initially 0. Use the mod wheel of your synth (or drag on the “Mod Wheel” **number box**) to introduce some slow flanging.

3. The same as before, but now the modulation rate is 6 Hz. The modulation depth is set very low for a subtle vibrato effect, but you can increase it to obtain a decidedly un-subtle wide vibrato.

4. A faster vibrato, with greater depth, and with the delayed signal fed back into the delay line, creates a complex warbling flange effect.

5. The right channel is delayed a short time for a flange effect and the left channel is delayed a longer time for an echo effect. Both delay times change sinusoidally over a two second period, and each delayed signal is fed back into its own delay line (causing a ringing resonance in the right channel and repeated echoes in the left channel).

6. Both delay times are set long with considerable feedback to create repeated echoes. The rate (and pitch) of the echoes is changed up and down by a very slow modulating frequency—one cycle every 10 seconds.

7. A similar effect, but modulated sinusoidally every 2 seconds.

8. Similar to example 5, but with flanging occurring at an audio rate of 55 Hz, and no original sound in the mix. The source sound is completely distorted, but the modulation rate gives the distortion its fundamental frequency.

Summary

You can provide a continuously varying delay time to **tapout~** by sending a signal in its inlet. As the delay time varies, the pitch of the delayed sound shifts oppositely. You can use a repeating low frequency wave to modulate the delay time, achieving either subtle or extreme pitch-variation effects. When a sound with a varying delay time is mixed with the original undelayed sound, the result is a variable comb filtering effect known as *flanging*. The depth (strength) of the flanging effect depends primarily on the amplitude of the signal that is modulating the delay time.

See Also

noise~	White noise generator
rand~	Band-limited random signal
tapin~	Input to a delay line
tapout~	Output from a delay line

Tutorial 30

Processing: Chorus

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*.

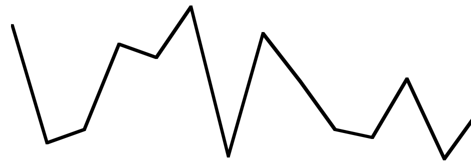
We've already seen in the preceding chapter how slight pitch shifts can be introduced by varying the delay time of a signal. When we mix this signal with its original undelayed version, we create interference between the two signals, resulting in a constantly varying filtering effect known as flanging.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

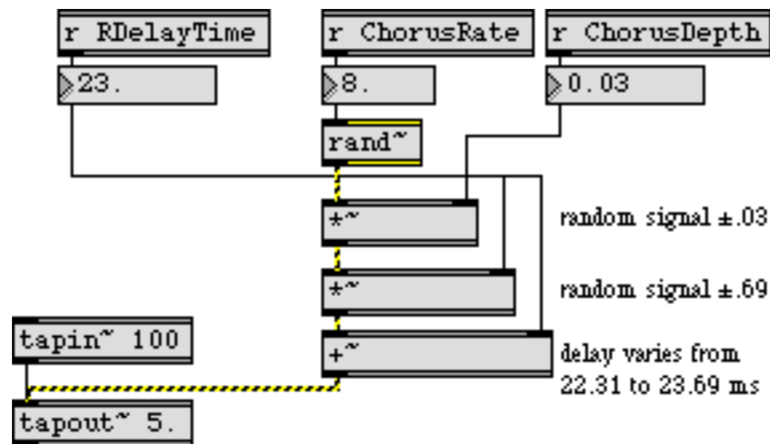
The **rand~** object chooses random numbers between -1 and 1, but does so less frequently than every sample. You can specify the frequency at which it chooses a new random value. In between those randomly chosen samples, **rand~** interpolates linearly from one value to the next to produce an unpredictable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

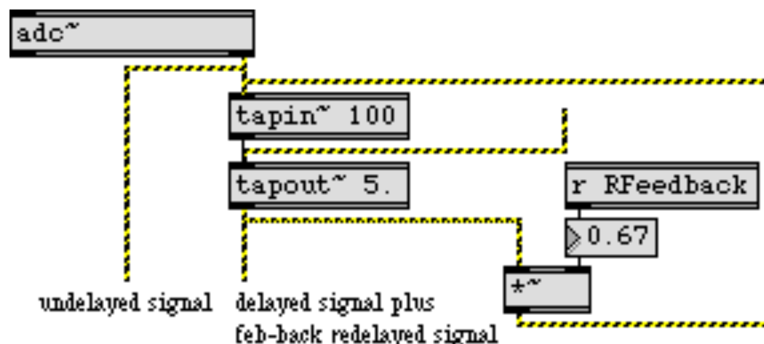


Unpredictable variations using **rand~**

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.

See Also

rand~	Band-limited random signal
tapout~	Output from a delay line

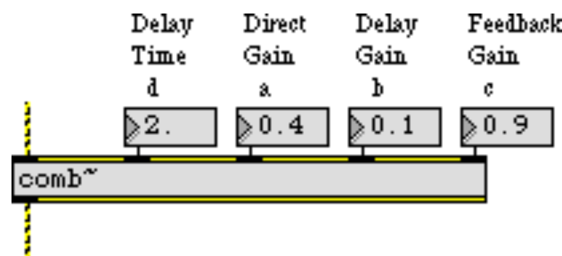
Comb filter: comb~

The minimum delay time that can be used for feedback into a delay line using **tapin~** and **tapout~** is determined by the signal vector size. However, many interesting filtering formulae require feedback using delay times of only a sample or two. Such filtering processes have to be programmed within a single MSP object.

An example of such an object is **comb~**, which implements a formula for *comb filtering*. Generally speaking, an audio filter is a frequency-dependent amplifier; it boosts the amplitude of some frequency components of a signal while reducing other frequencies. A comb filter accentuates and attenuates the input signal at regularly spaced frequency intervals—that is, at integer multiples of some fundamental frequency.

> **Technical detail:** The fundamental frequency of a comb filter is the inverse of the delay time. For example, if the delay time is 2 milliseconds ($^1/_{500}$ of a second), the accentuation occurs at intervals of 500 Hz (500, 1000, 1500, etc.), and the attenuation occurs between those frequencies. The extremity of the filtering effect depends on the factor (between 0 and 1) by which the feedback is scaled. As the scaling factor approaches 1, the accentuation and attenuation become more extreme. This causes the sonic effect of resonance (a "ringing" sound) at the harmonics of the fundamental frequency.

The **comb~** object sends out a signal that is a combination of a) the input signal, b) the input signal it received a certain time ago, and c) the output signal it sent that same amount of time ago (which would have included prior delays). In the inlets of **comb~** we can specify the desired amount of each of these three (*a*, *b*, and *c*), as well as the delay time (we'll call it *d*).



You can adjust all the parameters of the comb filter

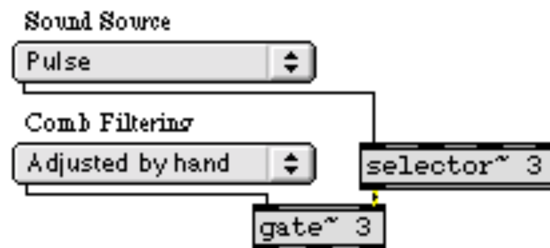
> **Technical detail:** At any given moment in time (we'll call that moment t), **comb~** uses the value of the input signal (x_t), to calculate the output y_t in the following manner.
> $$y_t = ax_t + bx_{(t-d)} + cy_{(t-d)}$$

The fundamental frequency of the comb filter depends on the delay time, and the intensity of the filtering depends on the other three parameters. Note that the scaling factor for the feedback (the

right inlet) should usually not exceed 1, since that would cause the output of the filter to increase steadily as a greater and greater signal is fed back.

Trying out the comb filter

The tutorial patch enables you to try out the comb filter by applying it to different sounds. The patch provides you with three possible sound sources for filtering—the audio input of your computer, a band-limited pulse wave, or white noise—and three filtering options—unfiltered, comb filter with parameters adjusted manually, or comb filter with parameters continuously modulated by other signals.



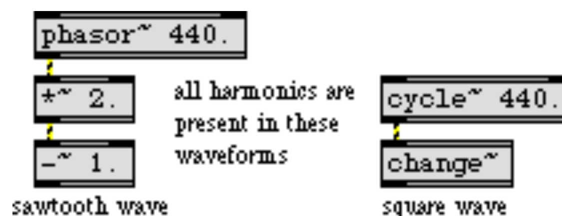
Choose a sound source and route it to the desired filtering using the pop-up menus

- Click on the buttons of the **preset** to try out some different combinations, with example parameter settings. Listen to the effect of the filter, then experiment by changing parameters yourself. You can use MIDI note messages from your synth to provide pitch and velocity (frequency and amplitude) information for the pulse wave, and you can use the mod wheel to change the delay time of the filter.

A comb filter has a characteristic harmonic resonance because of the equally spaced frequencies of its peaks and valleys of amplification. This trait is particularly effective when the comb is swept up and down in frequency, thus emphasizing different parts of the source sound. We can cause this frequency sweep simply by varying the delay time.

Band-limited pulse

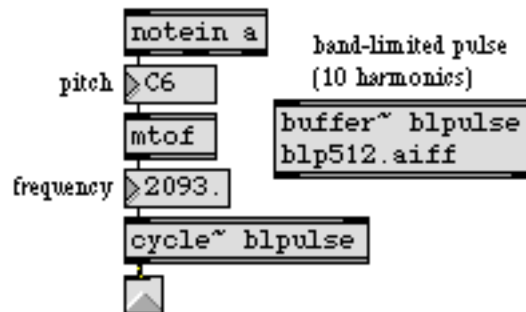
The effects of a filter are most noticeable when there are many different frequencies in the source sound, which can be altered by the filter. If we want to apply a comb filter to a pitched sound with a harmonic spectrum, it makes most sense to use a sound that has many partials such as a sawtooth wave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



These mathematically ideal waves may be too "perfect" for use as computer sound waves

The problem with such mathematically derived waveforms, though, is that they may actually be too rich in high partials. They may have partials above the Nyquist rate that are sufficiently strong to cause inharmonic aliasing. (This issue is discussed in more detail in *Tutorial 5*.)

For this tutorial we're using a waveform called a ***band-limited pulse***. A band-limited pulse has a harmonic spectrum with equal energy at all harmonics, but has a limited number of harmonics in order to prevent aliasing. The waveform used in this tutorial patch has ten harmonics of equal energy, so its highest frequency component has ten times the frequency of the fundamental. That means that we can use it to play fundamental frequencies up to 2,205 Hz if our sampling rate is 44,100 Hz. (Its highest harmonic would have a frequency of 22, 050 Hz, which is equal to the Nyquist rate.) Since the highest key of a 61-key MIDI keyboard plays a frequency of 2,093 Hz, this waveform will not cause aliasing if we use that as an upper limit.



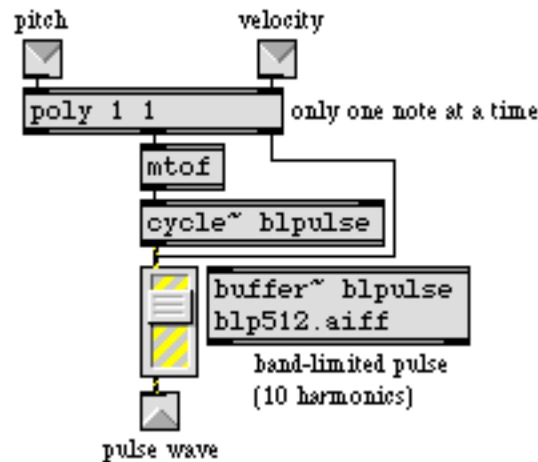
Playing a band-limited pulse wave with MIDI

Technical detail: In an idealized (optimally narrow) pulse wave, each cycle of the waveform would consist of a single sample with a value of 1, followed by all samples at 0. This would create a harmonic spectrum with all harmonics at equal amplitude, continuing upward infinitely. It's possible to make an MSP signal network that calculates—based on the fundamental frequency and the sampling rate—a band-limited pulse signal containing the maximum number of possible harmonics without foldover. In this case, though, we have chosen just to use a stored waveform containing ten partials.

Velocity-to-amplitude conversion: gain~

The subpatch **p** Pulse_Wave contains a simple but effective way to play a sound in MSP via MIDI. It uses a **poly** object to implement voice stealing, limiting the incoming MIDI notes to one note at a time. (It turns off the previous note by sending it out with a velocity of 0 before it plays the incoming note.) It then uses **mtof** to convert the MIDI note number to the correct frequency value for

MSP, and it uses the MSP object **gain~** to scale the amplitude of the signal according to the MIDI velocity.



Converting MIDI pitch and velocity data to frequency and amplitude information for MSP

The **gain~** object takes both a signal and a number in its left inlet. The number is used as an amplitude factor by which to scale the signal before sending it out. One special feature of **gain~** (aside from its utility as a user interface object for scaling a signal) is that it can convert the incoming numbers from a linear progression to a logarithmic or exponential curve. This is very appropriate in this instance, since we want to convert the linear velocity range (0 to 127) into an exponential amplitude curve (0 to 1) that corresponds roughly to the way that we hear loudness. Each change of velocity by 10 corresponds to a change of amplitude by 6 dB. The other useful feature of **gain~** is that, rather than changing amplitude abruptly when it receives a new number in its left inlet, it takes a few milliseconds to progress gradually to the new amplitude factor. The time it takes to make this progression can be specified by sending a time, in milliseconds, in the right inlet. In this patch, we simply use the default time of 20 ms.

- Choose one of the preset example settings, and choose "Pulse Wave" from the "Sound Source" pop-up menu. Play long notes with the MIDI keyboard. You can also obtain a continuous sound at any amplitude and frequency by sending numbers from the pitch and velocity **number box** objects (first velocity, then pitch) into the inlets of the **p** Pulse_Wave subpatch.

Varying parameters to the filter

As illustrated in this patch, it's usually best to change the parameters of a filter by using a gradually changing signal instead of making an abrupt change with single number. So parameter changes made to the "Adjusted By Hand" **comb~** object are sent first to a **line~** object for interpolation over a time of 25 ms.

The "Modulated" **comb~** object has its delay time varied at low frequency according to the shape of the band-limited pulse wave (just because it's a more interesting shape than a simple sinusoid). The modulation could actually be done by a varying signal of any shape. You can vary the rate of this modulation using the mod wheel of your synth (or just by dragging on the **number box**). The gain of the *x* and *y* delays (the two rightmost inlets) is modulated by a sine wave ranging between

0.01 and 0.99 (for the feedback gain) and a cosine wave ranging from 0.01 to 0.49 (for the feedforward gain). As the amplitude of one increases, the other decreases.

- Experimenting with different combinations of parameter values may give you ideas for other types of modulation you might want to design in your own patches.

Summary

The **comb~** object allows you to use very short feedback delay times to *comb filter* a signal. A comb filter creates frequency-dependent increases and decreases of amplitude in the signal that passes through it, at regularly spaced (i.e., harmonically related) frequency intervals. The frequency interval is determined by the inverse of the delay time. The comb filter is particularly effective when the delay time (and thus the frequency interval) changes over time, emphasizing different frequency regions in the filtered signal.

The user interface object **gain~** is useful for scaling the amplitude of a signal according to a specific logarithmic or exponential curve. Changes in amplitude caused by **gain~** take place gradually over a certain time (20 ms by default), so that there are no unwanted sudden discontinuities in the output signal.

See Also

comb~	Comb filter
gain~	Exponential scaling volume slider

This section, MSP Reference, contains information about each individual MSP objects. It includes:

MSP Objects

Contains precise technical information on the workings of each of the built-in and external objects supplied with MSP, organized in alphabetical order.

MSP Object Thesaurus

Consists of a reverse index of MSP objects, alphabetized by keyword rather than by object name. Use this Thesaurus when you want to know what object(s) are appropriate for the task you are trying to accomplish, then look up those objects by name in the *Objects* section.

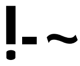
Manual Conventions

The central building block of Max is the object. Names of objects are always displayed in bold type, **like this**.

Messages (the arguments that are passed to and from objects) are displayed in plain type, like this.

Text that is displayed in blue type, **like this,** is hyperlinked to a Tutorial or MSP object reference page within this document. Clicking on the blue text will jump to the Tutorial or the reference page for the specified object.

In the "See Also" sections, anything in regular type is a reference to a section of either this manual or the Max Tutorials and Topics manual.



!-~

Signal subtraction (inlets reversed)

The **!-~** object functions just like the **-~** object, but the inlet order is reversed.

Input

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

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

float or int — In left inlet: An amount to subtract from 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: Subtracts the signal coming into the left inlet from this value. 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 right inlet. If a signal is connected to the left inlet, the argument is ignored. If no argument is present, and no signal is connected to the left inlet, the initial value is 0 by default.

Output

signal — The difference between the two inputs.

Examples

***-~** with the inlets reversed*

See Also

+~	Add signals

The **!/~** object functions just like the **/~** object, but the inlet order is reversed.

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

Input

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

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

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

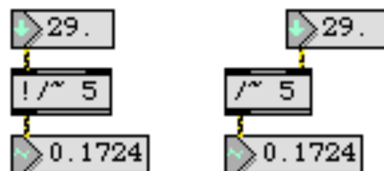
Arguments

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 right input divided by the left input.

Examples



***/~** with the inlets reversed*

See Also

*~	Multiply 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 not 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.

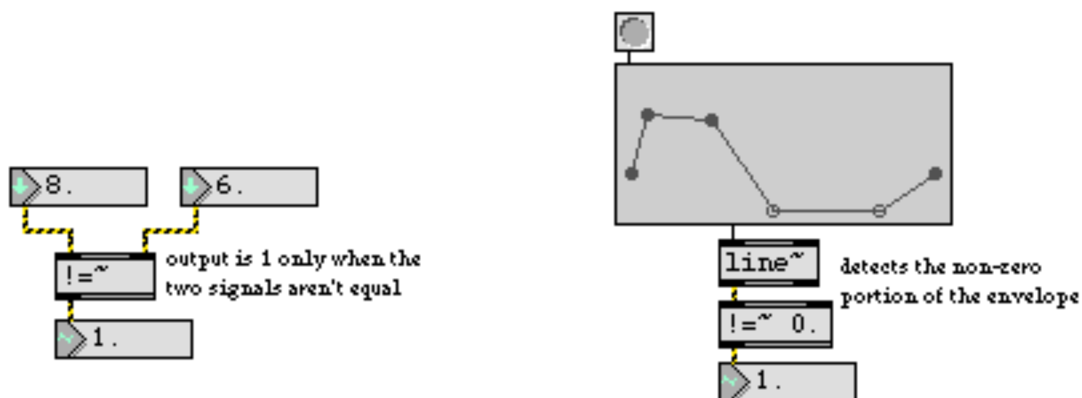
Arguments

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 not 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 not equal to the value in the right inlet, 1 is sent out; otherwise, 0 is sent out.

Examples



Use **!=~** *to detect the non-zero portion of a signal or envelope*

See Also

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

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, and the *remainder* is sent out the outlet.

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.

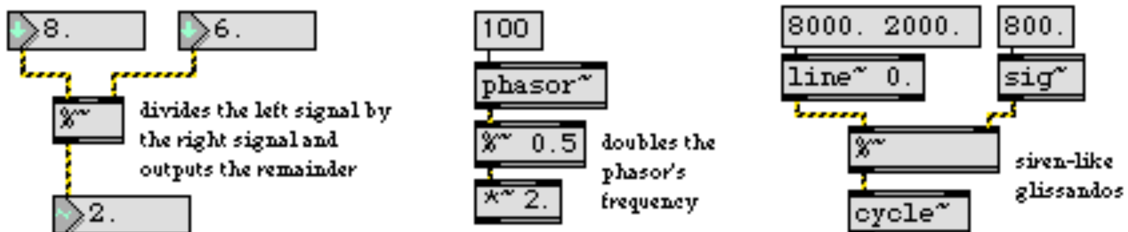
Arguments

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 — When the two signals in the inlets are divided, the remainder is sent out the outlet. **%** is called the *modulo* operator.

Examples



See Also

!/~	Signal division (inlets reversed)
/~	Divide one signal by another
Tutorial 8	Doing math in Max

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.

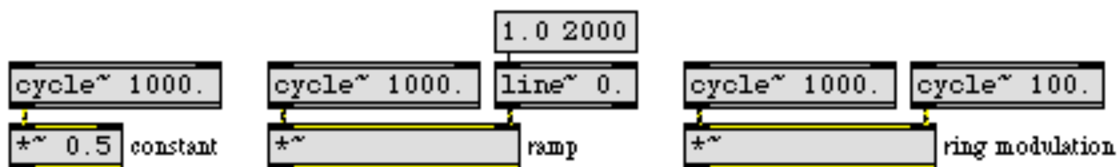
Arguments

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
!/~	Signal division (inlets reversed)
Tutorial 2	Fundamentals: Adjustable oscillator
Tutorial 8	Synthesis: Tremolo and ring modulation

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.

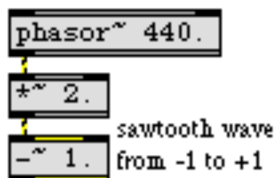
Arguments

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.

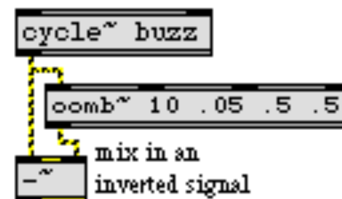
Output

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
!-~	Signal subtraction (inlets reversed)

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.

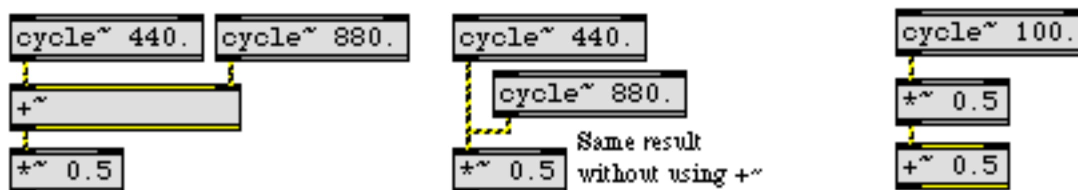
Arguments

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

+=~	Signal accumulator
-~	Signal subtraction
!-~	Signal subtraction (inlets reversed)

Input

signal Each sample of the input is added to the current sum and the sum is the corresponding sample of the output signal. For instance, assuming the sum started at 0, an input signal consisting of 1,1,1,1 would produce 1,2,3,4 as an output signal.

bang Resets the sum to 0.

set The word set, followed by a number, sets the sum to that number.

bang In left inlet: Outputs the currently stored value.

set The word set, followed by a number, sets the stored value to that number, without triggering output.

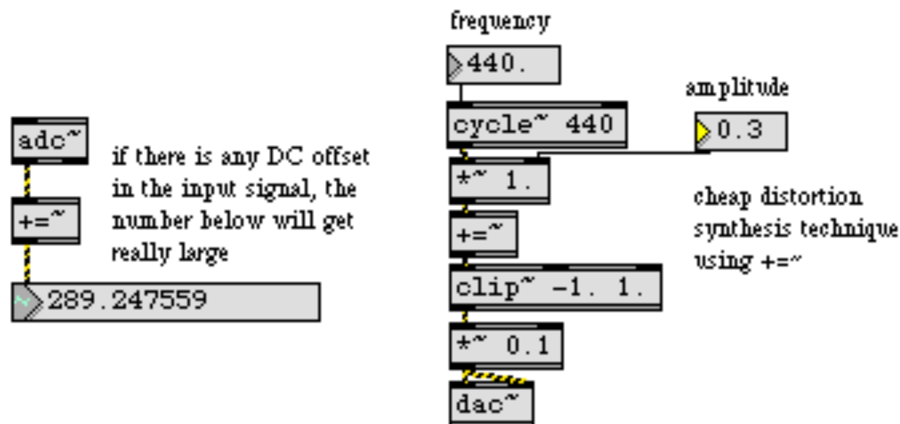
Arguments

float Optional. Sets the initial value for the sum. The default is 0.

Output

signal Each sample of the output is the sum of all previous input samples.

Examples



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.

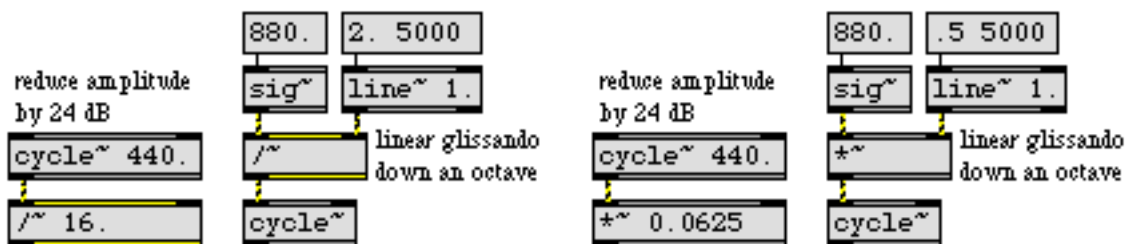
Arguments

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

!/~	Signal division (inlets reversed)
***~**	Multiply two signals
%~	Divide two signals, output the remainder

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.

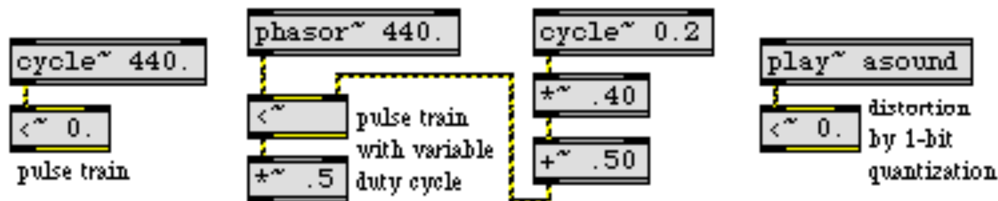
Arguments

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 less than or equal to,* comparison of two signals
>~	*Is greater than,* comparison of two signals
>=~	*Is greater than or equal to,* comparison of two signals
==~	*Is equal to,* comparison of two signals
!=~	*Not equal to,* 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 or 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.

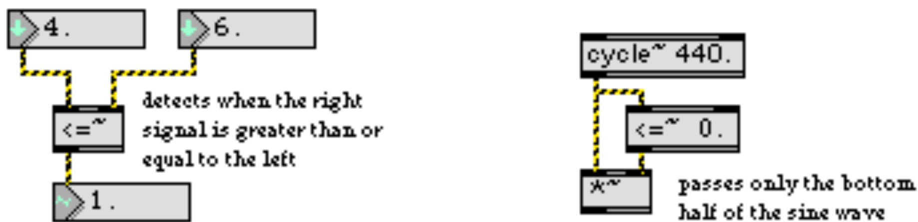
Arguments

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 less than or 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 less than or equal to the value in the right inlet, 1 is sent out; otherwise, 0 is sent out.

Examples



See Also

<~	*Is less than,* comparison of two signals
>~	*Is greater than,* comparison of two signals
>=~	*Is greater than or equal to,* comparison of two signals
==~	*Is equal to,* comparison of two signals
!=~	*Not equal to,* 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.

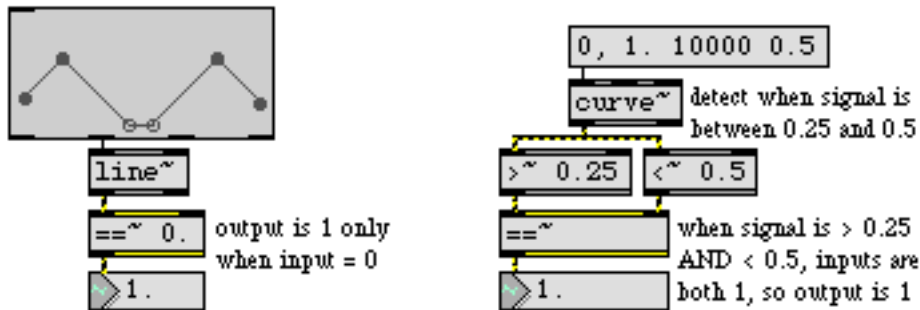
Arguments

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 less than or equal to,* comparison of two signals
>~	*Is greater than,* comparison of two signals
>=~	*Is greater than or equal to,* comparison of two signals
!=~	*Not equal to,* comparison of two signals
change~	Report signal direction
edge~	Detect logical signal transitions

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.

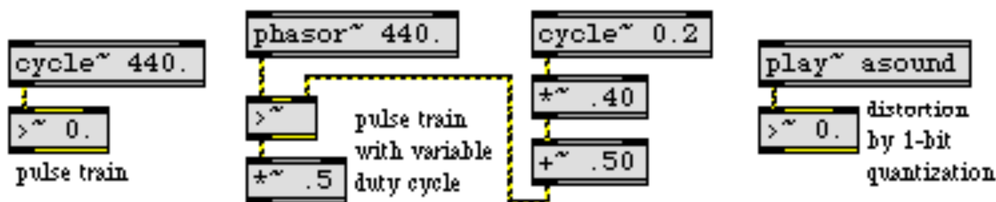
Arguments

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 less than or equal to,* comparison of two signals
>=~	*Is greater than or equal to,* comparison of two signals
==~	*Is equal to,* comparison of two signals
!=~	*Not equal to,* comparison of two signals
sah~	Sample and hold

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 or 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 greater than or 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 greater than or equal to the value in the right inlet, 1 is sent out; otherwise, 0 is sent out.

Examples

See Also

<~	*Is less than,* comparison of two signals
<=~	*Is less than or equal to,* comparison of two signals
>~	*Is greater than,* comparison of two signals
==~	*Is equal to,* comparison of two signals
!=~	*Not equal to,* comparison of two signals
sah~	Sample and hold

2d.wave~

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 **2d.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 **2d.wave~**, the perceived pitch of the signal coming out of **2d.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 2nd inlet: Input signal values progressing from 0 to 1 are used to determine which of the row(s) specified by the rows message will be used for playback. You can invert the **phasor~** to reverse the order in which row(s) are played.

In 3rd inlet: The start of the waveform as a millisecond offset from the beginning of a **buffer~** object's sample memory.

In 4th inlet: The end of the waveform as a millisecond offset from the beginning of a **buffer~** object's sample memory.

float or int — In 3rd or 4th inlets: Numbers can be used instead of signal objects to control the start and end points of the waveform, provided a signal is not connected to the inlet that receives the number.

rows — The word rows, followed by an int, sets the number of rows a given range of an audio file will be divided into. The phase input signal value received in the 2nd inlet of **2d.wave~** determines which row(s) are used for playback. The default value is 0.

set — The word set, followed by a symbol, sets the **buffer~** used by **2d.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 signal objects 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 **2d.wave~** for its stored waveform. Note that if the underlying data in a **buffer~** changes, the signal output of **2d.wave~** will change, since it does not copy the sample data in a **buffer~**. **2d.wave~** always uses the first *n* channels of a multi-channel **buffer~**, where *n* is the number of the **2d.wave~** object's output channels. The default number of channels, set by the third argument to the **2d.wave~** object, is 1.

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

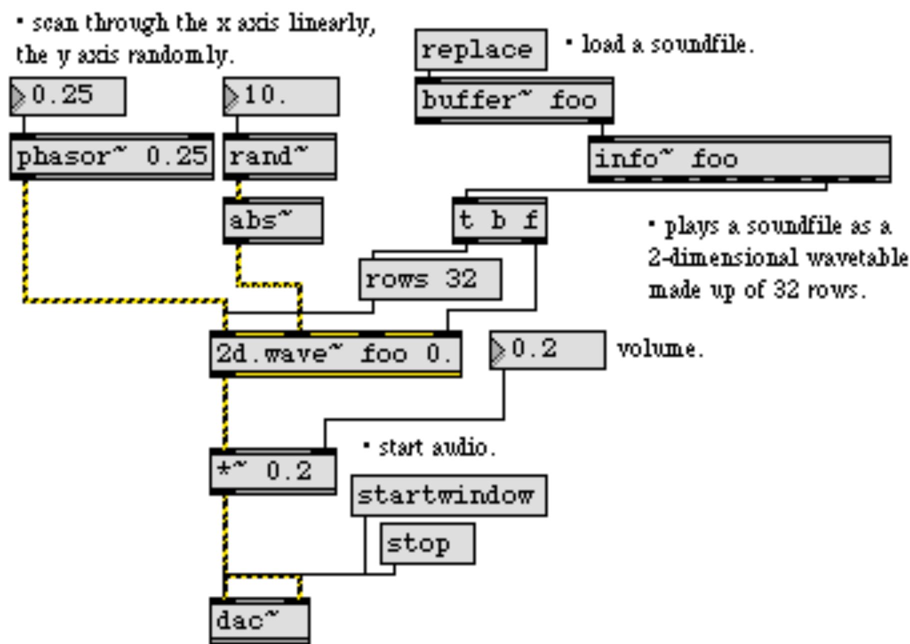
buffer~ object's sample memory. By default the start point is 0 and the end point is the end of the sample. If you want to set a non-zero start point but retain the sample end as the waveform end point, use only a single typed-in argument after the **buffer~** name. If a signal is connected to the start 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. The number of channels in the buffer~ file and the number of rows to be used may also be specified.

int Optional. Sets the number of output channels, which determines the number of outlets that the **2d.wave~** object will have. The maximum number of channels is 8. The default is 1. If the audio file being played has more output channels than the **2d.wave~** object, higher-numbered channels will not be played. If the audio file has fewer channels, the signals coming from the extra outlets of **2d.wave~** will be 0.

Output

signal The portion of the **buffer~** specified by the **2d.wave~** object's start and end points is scanned by signal values ranging from 0 to 1 in the **2d.wave~** object's inlet, and the corresponding sample value from the **buffer~** is sent out the **2d.wave~** object's outlet. If the signal received in the object'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 audio samples
groove~	Variable-rate looping sample playback
phasor~	Sawtooth wave generator
play~	Position-based sample playback
wave~	Variable-size wavetable
Tutorial 15	Sampling: Variable-length wavetable

abs~

Absolute value of a signal

Input

signal   Any signal.

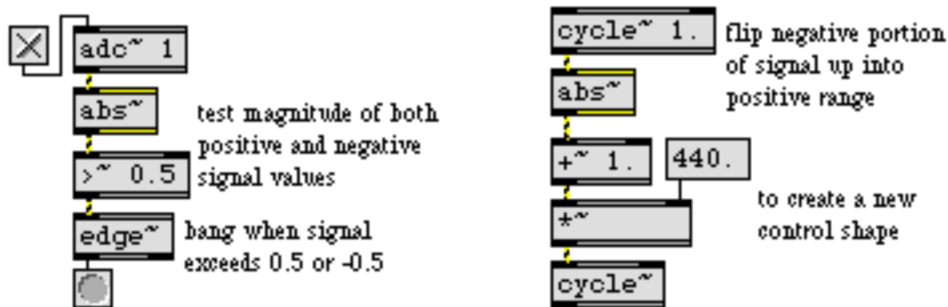
Arguments

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

acos~

Input

signal Input to a arc-cosine function.

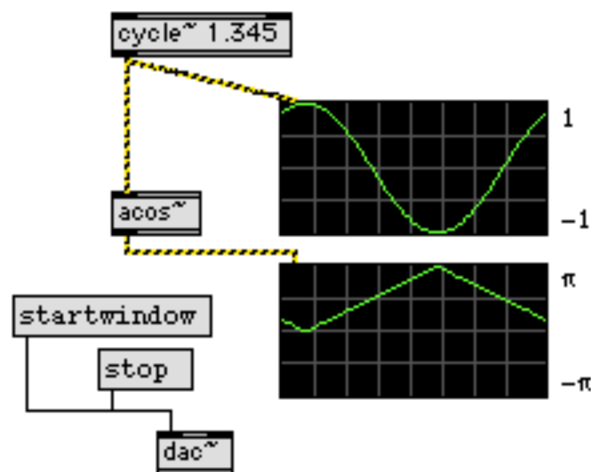
Arguments

None.

Output

signal The arc-cosine of the input in radians.

Examples



Using acos~ *to create an inverse linear ramp in radians (-π—π).*

See Also

acosh~	Signal hyperbolic arc-cosine function
asin~	Signal arc-sine function
asinh~	Signal hyperbolic arc-sine function
atan~	Signal arc-tangent function
atanh~	Signal hyperbolic arc-tangent function
atan2~	Signal arc-tangent function (two variables)
cos~	Signal cosine function (0-1 range)
cosh~	Signal hyperbolic cosine function
cosx~	Signal cosine function
sinh~	Signal hyperbolic sine function
sinx~	Signal sine function
tanh~	Signal hyperbolic tangent function
tanx~	Signal tangent function

acosh~

Input

signal Input to a hyperbolic arc-cosine function.

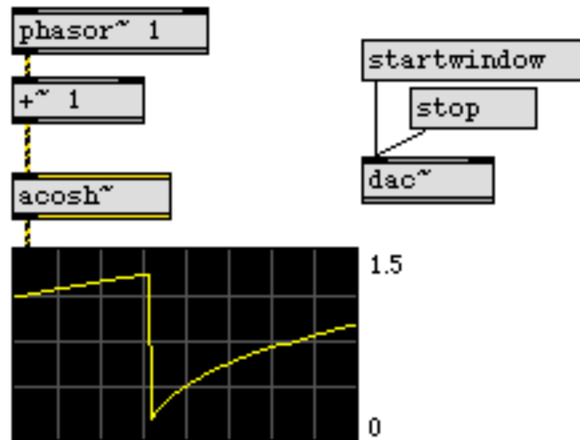
Arguments

None.

Output

signal The hyperbolic arc-cosine of the input.

Examples



See Also

acos~	Signal arc-cosine function
asin~	Signal arc-sine function
asinh~	Signal hyperbolic arc-sine function
atan~	Signal arc-tangent function
atanh~	Signal hyperbolic arc-tangent function
atan2~	Signal arc-tangent function (two variables)
cos~	Signal cosine function (0-1 range)
cosh~	Signal hyperbolic cosine function
cosx~	Signal cosine function
sinh~	Signal hyperbolic sine function
sinx~	Signal sine function
tanh~	Signal hyperbolic tangent function
tanx~	Signal tangent function

adc~

Input

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

open — Opens the DSP Status window.

set — The word set, followed by two numbers, sets the logical input channel for one of the object's signal outlets. The first number specifies the outlet number, where 1 is the leftmost outlet. The second number specifies the logical input channel (from 1 to 512). If the second number is 0, the outlet sends out the zero signal.

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.

wclose — Closes the DSP Status window if it is open

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

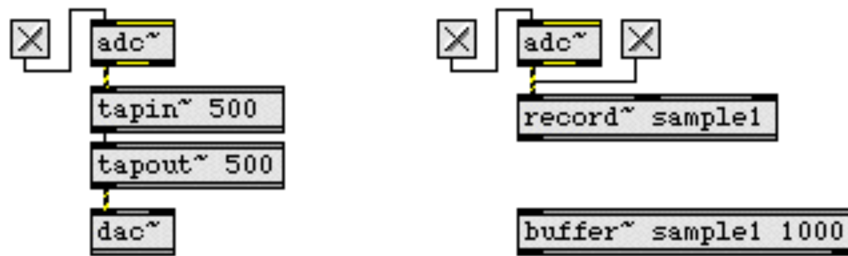
Arguments

int — Optional. You can create a **adc~** object that uses one or more audio input channel numbers between 1 and 512. These numbers refer to *logical channels* and can be dynamically reassigned to physical device channels of a particular driver using either the DSP Status window, its I/O Mappings subwindow, or an **adstatus** object with an input keyword argument. If the computer's built-in audio hardware is being used, there will be two input channels available. Other audio drivers and/or devices may have more than two channels. If no argument is typed in, **adc~** will have two outlets, initially set to logical 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

adstatus	Access audio driver output channels
ezadc~	Audio on/off; analog-to-digital converter
dac~	Audio output and on/off
Audio I/O	Audio input and output with MSP
Tutorial 13	Sampling: Recording and playback

adoutput~

Input

set — The word set, followed by two numbers, assigns an audio driver output channel to a signal outlet of the **adoutput~** object. The first number is the index of the outlet, where a value of 1 refers to the left outlet. The second number is the index of the audio driver output device channel where 1 refers to the first channel. If the second number if 0, the specified outlet is turned off and outputs a zero signal.

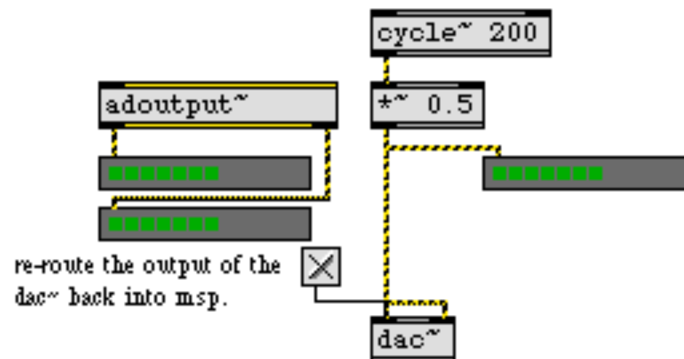
Arguments

int — Optional. The arguments specify output channels of the current audiodriver. There is no limit to the number of channels you can specify. By default, **adoutput~** creates two outlets and assigns the audio output from channels 1 and 2 of the current audiodriver to them. Note that these channel numbers are not the same as the logical channel numbers used by the **dac~** and **adc~** objects, but represent the "physical" outputs of the driver after any remapping has taken place. You configure the relationship between logical **dac~** channels and the audiodriver's real channels with the I/O Mappings subwindow of the DSP Status window.

Output

signal — Each outlet of **adoutput~** outputs a signal from the assigned audiodriver channel, delayed by the number of samples of the current signal vector size.

Examples



Capture the output of physical DAC channels to record (or re-process) the output of your patch

See Also

adstatus	Report and control audio driver settings
dac~	Audio output and on/off

adstatus

Access audio driver output channels

The **adstatus** object controls different audio settings depending on the argument you use. The possible arguments are listed in the Arguments section below.

Input

bang	In left inlet: Reports the current state of the setting. In many cases, messages are sent out the **adstatus** object's left outlet to set a pop-up menu object to display the current setting with a set message. In these cases, the numerical value of the setting is sent out the **adstatus** object's right outlet. The exact behaviors are listed in the Output section below.
override	In left inlet: The word override, followed by a 1, turns on override mode for the setting associated with the object. When override mode is enabled, any change to the setting is not saved in the MSP Preferences file. The message override 0 turns override mode off. By default, override is off for all settings. However, some settings are specific to audio drivers and may not be saved by the driver.
int	In left inlet: Changes the setting. In most cases, the number will correspond to the index of the menu item whose value was set by the bang message to **adstatus**. In right inlet: If the adstatus object is used with the input, iovs, output, sigvs, sr settings, an int in the right inlet sets the value numerically rather than by using a menu index (see the reset or loadbang message below). For all other settings, a number in the right inlet behaves identically to one in the left inlet.
set	In left inlet: The word set, followed by a number between 1 and 512, changes the logical channel associated with an **adstatus** input or **adstatus** output object. The current real audio driver input or output channel set for the new logical channel is sent out the object's outlets.
float	Same as int.
reset or loadbang	For **adstatus** objects that work with pop-up menus, the reset or loadbang messages output the necessary messages to make a pop-up menu that can control the **adstatus** object. The clear message is sent out first, followed by an append message for each menu item, followed by a set message to set the displayed value of the menu based on the current value of the setting.

Argument	*Behavior*
cpu	None.
cpulimit	Sets the percentage of CPU utilization above which audio processing will be suspended. A value of 0 turns off CPU utilization limiting.
driver	The number is interpreted as an index into the menu of available audio drivers generated by **adstatus** driver. The number loads the driver object corresponding to the menu index.

info	None.
input	The number is interpreted as an index into the menu of available audio input channels generated by **adstatus** input. The number sets the object's assigned logical channel to accept input from the driver's channel corresponding to the menu index.
iovs	The number is interpreted as an index into the menu of available I/O vector sizes generated by **adstatus** iovs. The number sets the driver's I/O vector size to the value of the item at the specified menu index.
latency	None.
numinputs	None.
numoutputs	None.
optimize	0 turns optimize mode off, 1 turns optimize mode on.
option	The number is interpreted as an index into the menu of choices for the specified option generated by **adstatus** option. The number sets the option to the value that corresponds with the menu index.
optionname	None.
output	The number is interpreted as an index into the menu of available audio output channels generated by **adstatus** output. The number sets the object's assigned logical channel to output to the driver's channel corresponding to the menu index.
overdrive	0 turns overdrive mode off, 1 turns overdrive mode on.
sigvs	The number is interpreted as an index into the menu of available signal vector sizes generated by **adstatus** sigvs. The number sets the current signal vector size to the value of the item at the specified menu index.
sr	The number is interpreted as an index into the menu of available sampling rates generated by **adstatus** sr. The number sets the current sampling rate to the value of the item at the specified menu index.
switch	0 turns the DSP off, 1 turns it on.
takeover	0 turns scheduler in audio interrupt mode off, 1 turns it on.
timecode	0 turns timecode output off, 1 turns it on.

Arguments

various — Obligatory. The first argument is a symbol that specifies the setting to be controlled by the **adstatus** object. Some settings require an additional int argument. The possible settings are:

cpu	Reports current CPU utilization.
cpulimit	Reports and sets the CPU utilization limit as a percentage from 0-100.
driver	Lists the available audio drivers and allows the current one to be changed.
info	Reports the number of function calls and signals used in the top level DSP chain.
input	Requires an additional argument specifying a logical channel number (used by the **adc~** object) between 1 and 512. Lists the available audio driver input channels and allows the current setting to be changed.
iovs	Reports the available I/O vector sizes of the current audio driver and allows the current I/O vector size setting to be changed.
latency	If supported by the audio driver, reports the input and output latencies of the driver in samples.
numinputs	Reports the number of input channels of the current audio driver.
numoutputs	Reports the number of output channels of the current audio driver.
optimize	Turns the optimization flag on or off. On the Macintosh, this is used to control the use of Altivec (G4 processor) optimizations.
option	Requires an additional argument specifying the option number (starting at 1). If the current audio driver uses the numbered option, reports the available choices for setting the value of the option.
optionname	Requires an additional argument specifying the option number (starting at 1). If the current audio driver uses the numbered option, the name of the option is reported.
output	Requires an additional argument specifying a logical channel number (used by the **dac~** object) between 1 and 512. Lists the available audio driver output channels and allows the current setting to be changed.

overdrive — Controls the setting of overdrive mode (where the scheduler runs in a high-priority interrupt).

sigvs — Reports the available signal vector sizes and allows the current signal vector size setting to be changed.

sr — Reports the available sampling rates and allows the current sampling rate setting to be changed.

switch — Turns the DSP on or off.

takeover — Controls the setting of scheduler in audio interrupt mode.

timecode — If supported by the audio driver, reports the current timecode value.

Output

various — Out left outlet: For many settings, a series of messages intended to set up a pop-up menu object are sent out the left outlet when the reset or loadbang message is received by **adstatus**. See the reset message in the Input section for more details.

The following settings have a menu-style output: driver, input, iovs, optimize, output, sigvs, sr, switch, and takeover.

set — Out left outlet: When a bang message is received or when the value of the setting that has a menu-style output is changed, the word set, followed by a number with a menu item index (starting at 0) is sent out. Here are details of outputs from the left outlet for specific settings with menu-style outputs:

driver — Lists all current audio driver choices.

input — Lists audio input channels for the audio driver currently in use.

iovs — Lists I/O vector sizes for the audio driver currently in use.

optimize — Creates an On/Off menu for use with this setting.

option — Creates a list of choices for the specified option.

optionname — Sets a menu that names the specified option. Intended for use with a pop-up menu object in label mode.

output — Lists audio output channels for the audio driver currently in use.

overdrive — Creates an On/Off menu for use with this setting.

sigvs — Lists signal vector sizes for the audio driver currently in use.

	sr	Lists sampling rates available for the audio driver currently in use.
	switch	Creates an On/Off menu for turning the DSP on and off.
	takeover	Creates an On/Off menu for switching scheduler in audio interrupt mode.

int or float — Out left outlet: For objects that don't use a menu-style output, the current value of the setting is sent out the left outlet. Here are details for specific settings:

cpu	Reports CPU utilization as a percentage (normally from 0 to 100).
cpulimit	Reports the current CPU utilization limit.
info	Reports the number of function calls used in the top-level DSP chain.
latency	If supported by the audio driver, reports the input latency of the audio driver.
numinputs	Reports the number of inputs in the current audio driver.
numoutputs	Reports the number of outputs in the current audio driver.
timecode	If supported by the audio driver, reports the current timecode as a list in the following format:

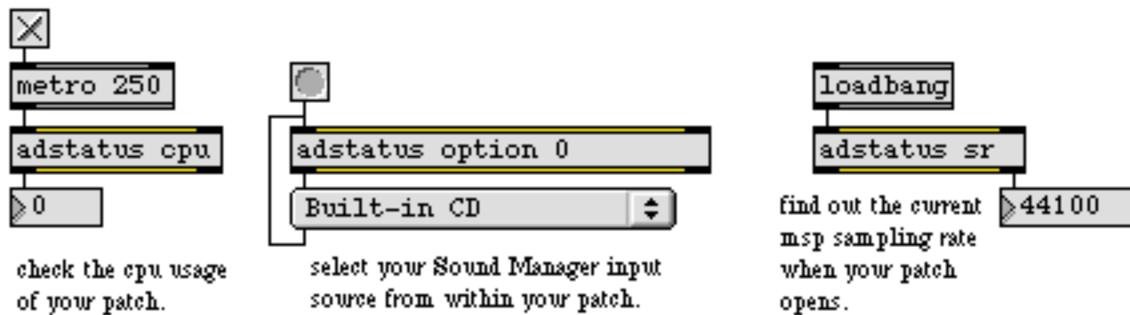
1. time code sample count most significant word
2. time code sample count least significant word
3. time code subframes
4. time code flags
5. time code frame rate

int or float — Out right outlet: Here are the objects that output something out the value outlet of the object:

info	Reports the number of signals used in the top-level DSP chain.
iovs	Reports the current I/O vector size.
sigvs	Reports the current signal vector size.
option	Reports the menu item index of the option's current value.
switch	Reports the current on/off setting of the DSP.
takeover	Reports the current on/off setting of takeover mode.
input	Reports the current input channel for the specified logical channel.

output	Reports the current output channel for the specified logical channel.
overdrive	Reports the current on/off setting of overdrive mode.
sr	Reports the current sampling rate.
numinputs	Reports the number of inputs in the current audio driver (same as left outlet).
numoutputs	Reports the number of outputs in the current audio driver (same as left outlet).
overdrive	Reports the current on/off setting of overdrive mode.

Examples



adstatus *lets you monitor and change audio parameters from within your patch.*

See Also

dspstate~	Report current DSP setting
adoutput~	Access audio driver output channels
Audio I/O	Audio input and output with MSP

allpass~

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 the **allpass~** object's memory of previous outputs, resetting them to 0.

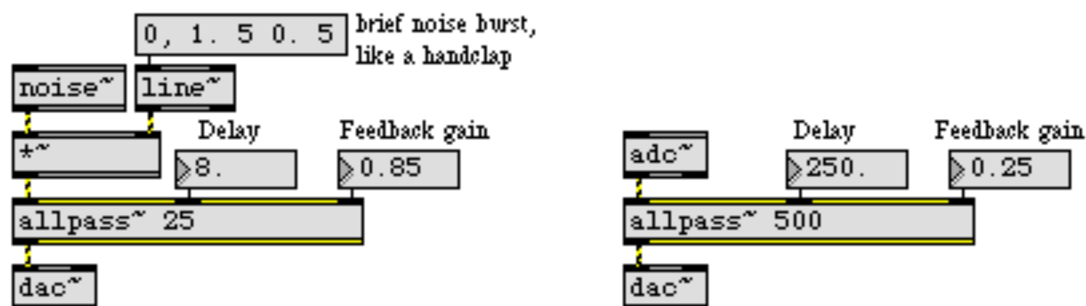
Arguments

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 echoes

See Also

biquad~	Two-pole two-zero filter
comb~	Comb filter
lores~	Resonant lowpass filter
reson~	Resonant bandpass filter
teeth~	Comb filter with feedforward and feedback delay control

asin~

Input

signal Input to a arc-sine function.

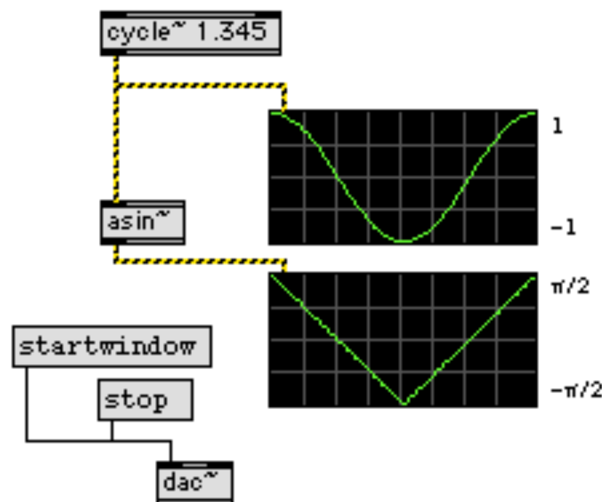
Arguments

None.

Output

signal The arc-sine of the input in radians.

Examples



asin~ *lets you create linear ramps in radians in the range -π/2—π/2*

See Also

acos~	Signal arc-cosine function
acosh~	Signal hyperbolic arc-cosine function
asinh~	Signal hyperbolic arc-sine function
atan~	Signal arc-tangent function
atanh~	Signal hyperbolic arc-tangent function
atan2~	Signal arc-tangent function (two variables)
cos~	Signal cosine function (0-1 range)
cosh~	Signal hyperbolic cosine function
cosx~	Signal cosine function
sinh~	Signal hyperbolic sine function
sinx~	Signal sine function
tanh~	Signal hyperbolic tangent function
tanx~	Signal tangent function

asinh~

Signal hyperbolic arc-sine function

Input

signal Input to a hyperbolic arc-sine function.

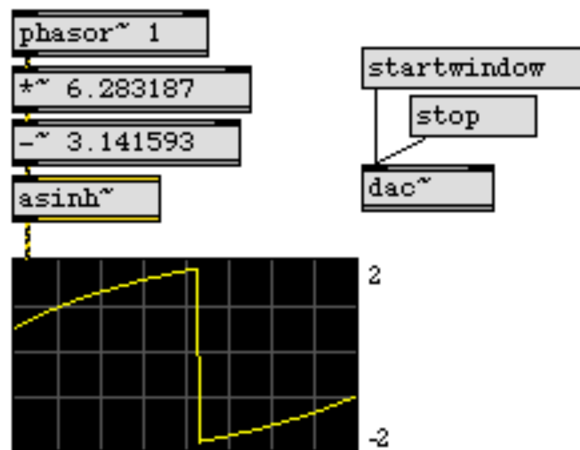
Arguments

None.

Output

signal The hyperbolic arc-sine of the input in radians.

Examples



See Also

acos~	Signal arc-cosine function
acosh~	Signal hyperbolic arc-cosine function
asin~	Signal arc-sine function
atan~	Signal arc-tangent function
atanh~	Signal hyperbolic arc-tangent function
atan2~	Signal arc-tangent function (two variables)
cos~	Signal cosine function (0-1 range)
cosh~	Signal hyperbolic cosine function
cosx~	Signal cosine function
sinh~	Signal hyperbolic sine function
sinx~	Signal sine function
tanh~	Signal hyperbolic tangent function
tanx~	Signal tangent function

Input

signal Input to a arc-tangent function.

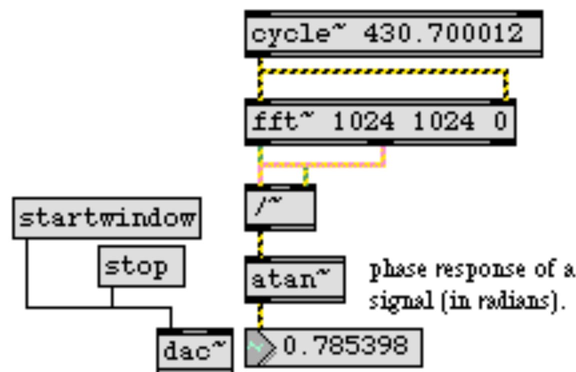
Arguments

None.

Output

signal The arc-tangent of the input.

Examples



atan~ *performs the arctangent function on a signal*

See Also

acos~	Signal arc-cosine function
acosh~	Signal hyperbolic arc-cosine function
asin~	Signal arc-sine function
asinh~	Signal hyperbolic arc-sine function
atanh~	Signal hyperbolic arc-tangent function
atan2~	Signal arc-tangent function (two variables)
cos~	Signal cosine function (0-1 range)
cosh~	Signal hyperbolic cosine function
cosx~	Signal cosine function
sinh~	Signal hyperbolic sine function
sinx~	Signal sine function
tanh~	Signal hyperbolic tangent function
tanx~	Signal tangent function

atanh~

Signal hyperbolic arc-tangent function

Input

signal Input to a hyperbolic arc-tangent function.

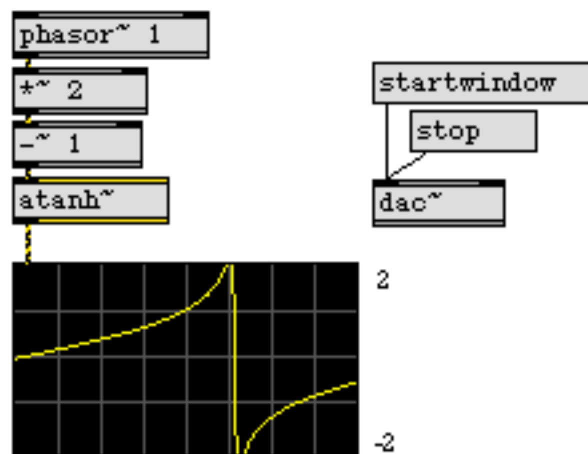
Arguments

None.

Output

signal The hyperbolic arc-tangent of the input.

Examples



See Also

acos~	Signal arc-cosine function
acosh~	Signal hyperbolic arc-cosine function
asin~	Signal arc-sine function
asinh~	Signal hyperbolic arc-sine function
atan~	Signal arc-tangent function
atan2~	Signal arc-tangent function (two variables)
cos~	Signal cosine function (0-1 range)
cosh~	Signal hyperbolic cosine function
cosx~	Signal cosine function
sinh~	Signal hyperbolic sine function
sinx~	Signal sine function
tanh~	Signal hyperbolic tangent function
tanx~	Signal tangent function

atan2~

Signal arc-tangent function (two variables)

Input

signal In left input: *x* value input to an arc-tangent function.

In right input: *y* value input to an arc-tangent function.

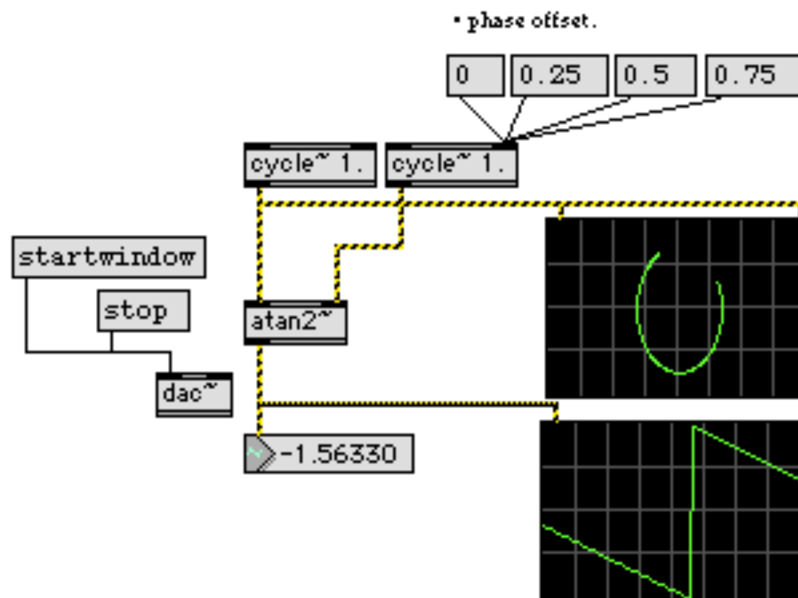
Arguments

None.

Output

signal The arc-tangent input values (i.e. *Arc-tangent(y/x)*).

Examples



atan2~ *Calculate the angle of two points around an origin (0, 0), in radians*

See Also

acos~	Signal arc-cosine function
acosh~	Signal hyperbolic arc-cosine function
asin~	Signal arc-sine function
asinh~	Signal hyperbolic arc-sine function
atan~	Signal arc-tangent function
atanh~	Signal hyperbolic arc-tangent function
cos~	Signal cosine function (0-1 range)
cosh~	Signal hyperbolic cosine function
cosx~	Signal cosine function
sinh~	Signal hyperbolic sine function
sinx~	Signal sine function
tanh~	Signal hyperbolic tangent function
tanx~	Signal tangent function

average~

Multi-mode signal average

Input

signal — The signal to be averaged.

int — Sets the interval in samples used for each of the three modes of signal averaging. The default value is 100.

bipolar — Sets bipolar averaging mode (default). In bipolar mode, the sample values are averaged.

absolute — Sets absolute averaging mode. This mode averages the absolute value of the incoming samples.

rms — Sets root mean square (RMS) averaging mode. This mode computes the square root of the average of the sample values squared.

The RMS mode of the **average~** object is more CPU-intensive than the bipolar and absolute modes.While RMS values are often used to measure signal levels, the absolute mode often works as well as the RMS mode in many level-detection tasks.

Arguments

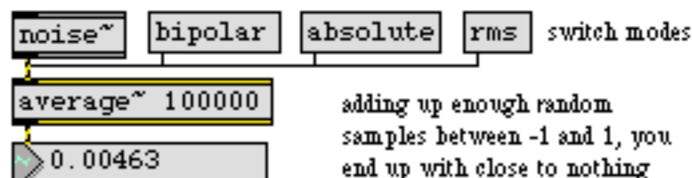
int — Optional. Sets the maximum averaging interval in samples. The default value is 100.

symbol — Optional. Sets the averaging mode, as defined above. The default is bipolar.

Output

float — The running average value of the input signal averaged over the specified number of samples.

Examples



Running average of a signal across n samples

See Also

avg~	Signal average
meter~	Visual peak level indicator

avg~

Input

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

signal — The signal to be averaged.

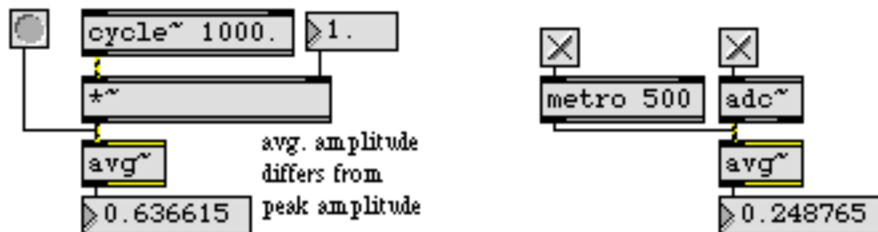
Arguments

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

average~	Multi-mode signal average
avg~	Signal average
meter~	Visual peak level indicator

begin~

Define a switchable part of a signal network

Input

None.

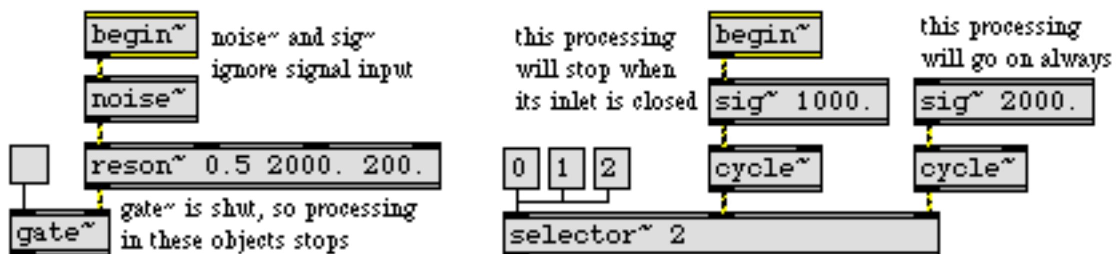
Arguments

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: $y_n = a_0 x_n + a_1 x_{n-1} + a_2 x_{n-2} - b_1 y_{n-1} - b_2 y_{n-2}$.

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

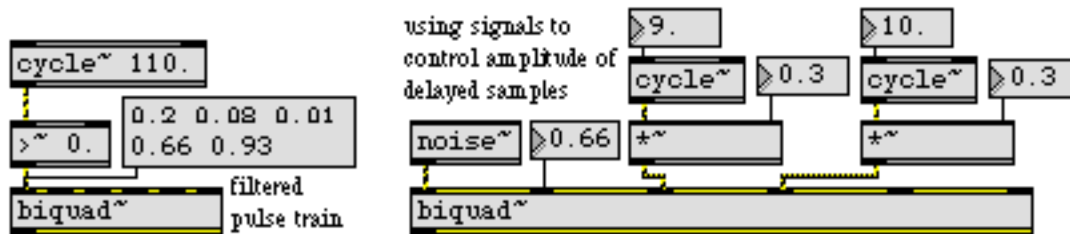
Arguments

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

buffir~	Buffer-based FIR filter
comb~	Comb filter
filtergraph~	Graphical filter editor
lores~	Resonant lowpass filter
onepole~	Single-pole lowpass filter
reson~	Resonant bandpass filter
teeth~	Comb filter with feedforward and feedback delay control

bitand~

Bitwise and of floating point signals

The **bitand~** object performs a bitwise intersection (a bitwise “and”) on two incoming floating-point signals as either raw 32-bit data or as integer values. The output is a floating-point signal composed of those bits which are 1 in *both* numbers.

Input

signal — In left inlet: The floating-point signal is compared, in binary form, with the floating-point signal in the right inlet. The signal can be treated as either a floating-point signal or as an integer.

In right inlet: The floating-point signal to be compared with the signal in the left inlet. The signal can be treated as either a floating-point signal or as an integer.

The raw floating-point signal bit values are expressed in the following form:

<1 sign bit> <8 exponent bits> <23 mantissa bits>

int — In right inlet: An integer value can be used as a bitmask when supplied to the right inlet of the **bitand~** object, provided that the proper mode is set.

bits — In left inlet: The word bits, followed by a list containing 32 ones or zeros, specifies a bitmask to be used by **bitand~**. Alternately, a bitmask value can be set by using an int value in the right inlet.

mode — In left inlet: The word mode, followed by a zero or one, specifies whether the floating signal or floating-point values will be processed as a raw 32-bit floating-point value or converted to an integer value for the bitwise operation. The modes of operation are:

Mode	*Description*
0	Treat both floating-point signal inputs as raw 32-bit values (default).
1	Convert both floating-point signal inputs to integer values.
2	Treat the floating-point signal in the left inlet as a raw 32-bit value and treat the value in the right inlet as an integer.
3	Convert the floating-point signal in the left inlet to an integer and treat the right input as a raw 32-bit value.

Note: If you convert the floating-point signal input to an int and then convert it back, the resulting floating-point value will retain only 24 bits of integer resolution.

Arguments

int — Optional. Sets the bitmask to be used by the **bitand~** object. The default is 0. An integer value can be used as a bitmask regardless of the mode; the binary representation of this integer is the bitmask.

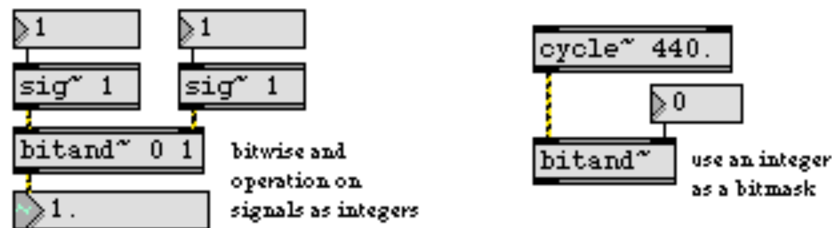
int — Optional. Specifies whether the floating-point signal or floating-point values will be processed as raw 32-bit floating-point values or converted to integer values for the bitwise operation. The modes of operation are:

Mode	*Description*
0	Treat both floating-point signal inputs as raw 32-bit values (default).
1	Convert both floating-point signal inputs to integer values.
2	Treat the floating-point signal in the left inlet as a raw 32-bit value and the value in the right inlet as an integer.
3	Convert the floating-point signal in the left inlet to an integer and treat the right input as a raw 32-bit value.

Output

signal — The two floating-point signals or ints received in the inlets are compared, one bit at a time. If a bit is 1 in both numbers, it will be 1 in the output number, otherwise it will be 0 in the output floating-point signal.

Examples



See Also

bitshift~	Bitwise shifting of a floating-point signal
bitor~	Bitwise "or" of floating-point signals
bitxor~	Bitwise "exclusive or" of floating-point signals
bitnot~	Bitwise inversion of a floating-point signal

bitnot~

Bitwise inversion of a floating point signal

The **bitnot~** object performs a bitwise inversion on an incoming floating-point signal as either raw 32-bit data or as an integer value. All bit values of 1 are set to 0, and vice versa.

Input

signal — The **bitnot~** object can perform bit inversion on either a floating-point signal as bits, or as an integer.

Floating-point signal bit values are expressed in the following form:

<1 sign bit> <8 exponent bits> <23 mantissa bits>

mode — In left inlet: The word mode, followed by a zero or one, specifies whether the floating signal or floating-point value will be processed as a raw 32-bit floating-point value or converted to an integer value for bit inversion. The modes of operation are:

Mode	*Description*
0	Treat floating-point signal input as a raw 32-bit value (default).
1	Convert the floating-point signal input to an integer value.

Note: If you convert the floating-point signal input to an int and then convert it back, the resulting floating-point value will retain only 24 bits of integer resolution.

Arguments

int — Optional. Specifies whether the floating-point signal or floating-point value will be processed as a raw 32-bit floating-point value or converted to an integer value for bit inversion. The modes of operation are:

Mode	*Description*
0	Treat floating-point signal input as a raw 32-bit value (default).
1	Convert the floating-point signal input to an integer value.

Output

signal — The resulting bit inverted floating-point signal.

bitnot~

Bitwise inversion of a floating point signal

Examples

See Also

bitshift~	Bitwise shifting of a floating-point signal
bitor~	Bitwise "or" of floating-point signals
bitxor~	Bitwise "exclusive or" of floating-point signals
bitand~	Bitwise "and" of floating-point signals

bitor~

Bitwise or of floating point signals

The **bitor~** object performs a bitwise "or" on two incoming floating-point signals as either raw 32-bit data or as integer values. The bits of both incoming signals are compared, and a 1 is output if *either* of the two bit values is 1. The output is a floating-point signal composed of the resulting bit pattern.

Input

signal — In left inlet: The floating-point signal is compared, in binary form, with the floating-point signal in the right inlet. The signal can be treated as either a floating-point signal or as an integer.

In right inlet: The floating-point signal to be compared with the signal in the left inlet. The signal can be treated as either a floating-point signal or as an integer.

The raw floating-point signal bit values are expressed in the following form:

<1 sign bit> <8 exponent bits> <23 mantissa bits>

int — In right inlet: An integer value can be used as a bitmask when supplied to the right inlet of the **bitor~** object, provided that the proper mode is set.

bits — In left inlet: The word bits, followed by a list containing 32 ones or zeros, specifies a bitmask to be used by **bitor~**. Alternately, a bitmask value can be set by using an int value in the right inlet.

mode — In left inlet: The word mode, followed by a zero or one, specifies whether the floating signal or floating-point values will be processed as raw 32-bit floating-point values or converted to integer values for the bitwise operation. The modes of operation are:

Mode	*Description*
0	Treat both floating-point signal inputs as raw 32-bit values (default).
1	Convert both floating-point signal inputs to integer values.
2	Treat the floating-point signal in the left inlet as a raw 32-bit value and the value in the right inlet as an integer.
3	Convert the floating-point signal in the left inlet to an integer and treat the right input as a raw 32-bit value.

Note: If you convert the floating-point signal input to an int and then convert it back, the resulting floating-point value will retain only 24 bits of integer resolution.

Arguments

int — Optional. Sets the bitmask to be used by the **bitor~** object. The default is 0. An integer value can be used as a bitmask regardless of the mode; the binary representation of this integer is the bitmask.

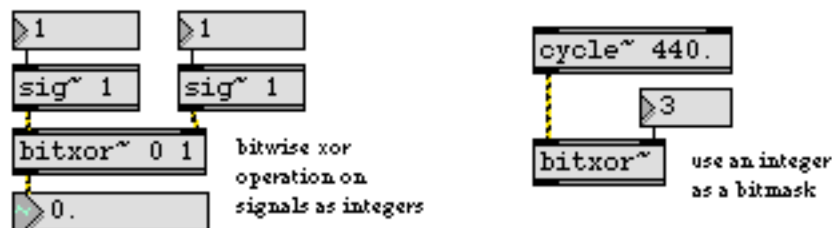
int — Optional. Specifies whether the floating-point signal or floating-point values will be processed as raw 32-bit floating-point values or converted to integer values for the bitwise operation. The modes of operation are:

Mode	*Description*
0	Treat both floating-point signal inputs as raw 32-bit values (default).
1	Convert both floating-point signal inputs to integer values.
2	Treat the floating-point signal in the left inlet as a raw 32-bit value and the value in the right inlet as an integer.
3	Convert the floating-point signal in the left inlet to an integer and treat the right input as a raw 32-bit value.

Output

signal — The two floating-point signals or ints received in the inlets are compared, one bit at a time. If a bit is 1 in either one of the numbers, it will be 1 in the output number, otherwise it will be 0 in the output number. The output is a floating-point signal composed of the resulting bit pattern.

Examples



See Also

bitshift~	Bitwise shifting of a floating-point signal
bitand~	Bitwise “and” of floating-point signals
bitxor~	Bitwise “exclusive or” of floating-point signals
bitnot~	Bitwise inversion of a floating-point signal

Input

signal — The **bitshift~** object performs bit shifting on a floating-point signal as either raw 32-bit data or as an integer value.

floating-point signal bit values are expressed in the following form:

<1 sign bit> <8 exponent bits> <23 mantissa bits>

mode — In left inlet: The word mode, followed by a zero or one, specifies whether the floating signal or floating-point value will be processed as a raw 32-bit floating-point value or converted to an integer value for bit shifting. The modes of operation are:

Mode	*Description*
0	Treat floating-point signal input as a raw 32-bit value (default).
1	Convert the floating-point signal input to an integer value.

Note: If you convert the floating-point signal input to an int and then convert it back, the resulting floating-point value will retain only 24 bits of integer resolution.

shift — In left inlet: The word shift, followed by a positive or negative number, specifies the number of bits to be shifted on the incoming floating-point signal. Positive number values correspond to left shifting that number of bits (i.e., Left shifting a number *n* places is the same as dividing it by 2^n). Negative numbers correspond to right shifting that number of bits (i.e., Right shifting a number n places is the same as dividing it by 2^n).

Arguments

int — Optional. Sets the number of bits to be shifted on the incoming floating-point signal. Positive shift values correspond to left shifting that number of bits, negative shift values correspond to right shifting that number of bits.

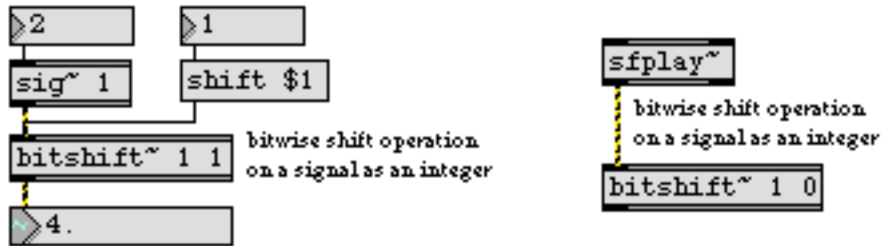
int — Optional. Specifies whether the floating signal or floating-point value will be processed as a raw 32-bit floating-point value or converted to an integer value for bit shifting. The modes of operation are:

Mode	*Description*
0	Treat floating-point signal input as a raw 32-bit value (default).
1	Convert the floating-point signal input to an integer value.

Output

signal — The resulting bit shifted floating-point signal.

Examples



See Also

bitand~	Bitwise “and” of floating-point signals
bitor~	Bitwise “or” of floating-point signals
bitxor~	Bitwise “exclusive or” of floating-point signals
bitnot~	Bitwise inversion of a floating-point signal

bitxor~

Bitwise exclusive or of floating point signals

The **bitxor~** object performs a bitwise "exclusive or" on two incoming floating-point signals as either raw 32-bit data or as integer values. The bits of both incoming signals are compared, and the corresponding output bit will be set to 1 if the two bit values are different, and 0 if the two values are the same. The output is a floating-point signal composed of the resulting bit pattern.

Input

signal — In left inlet: The floating-point signal is compared, in binary form, with the floating-point signal in the right inlet. The signal can be treated as either a floating-point signal or as an integer.

In right inlet: The floating-point signal to be compared with the signal in the left inlet. The signal can be treated as either a floating-point signal or as an integer.

The raw floating-point signal bit values are expressed in the following form:

<1 sign bit> <8 exponent bits> <23 mantissa bits>

int — In right inlet: An integer value can be used as a bitmask when supplied to the right inlet of the **bitxor~** object, provided that the proper mode is set.

bits — In left inlet: The word bits, followed by a list containing 32 ones or zeros, specifies a bitmask to be used by **bitxor~**. Alternately, a bitmask value can be set by using an int value in the right inlet.

mode — In left inlet: The word mode, followed by a zero or one, specifies whether the floating signal or floating-point values will be processed as raw 32-bit floating-point values or converted to integer values for the bitwise operation. The modes of operation are:

Mode	*Description*
0	Treat both floating-point signal inputs as raw 32-bit values (default).
1	Convert both floating-point signal inputs to integer values.
2	Treat the floating-point signal in the left inlet as a raw 32-bit value and treat the value in the right inlet as an integer.
3	Convert the floating-point signal in the left inlet to an integer and treat the right input as a raw 32-bit value.

Note: If you convert the floating-point signal input to an int and then convert it back, the resulting floating-point value will retain only 24 bits of integer resolution.

Output

signal — The two floating-point signals or ints received in the inlets are compared, one bit at a time. A 1 is output if the two bit values are different, 0 if they are the same. The output is a floating-point signal composed of the resulting bit pattern.

bitxor~

Bitwise exclusive or of floating point signals

Examples

See Also

bitshift~	Bitwise shifting of a floating-point signal
bitand~	Bitwise “and” of floating-point signals
bitor~	Bitwise “or” of floating-point signals
bitnot~	Bitwise inversion of a floating-point signal

Input

bang — Redraws the contents of the **buffer~** object's waveform display window. You can open the display window by double-clicking on the **buffer~** object.

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

clearlow — Erases the contents of the buffer like the clear message, but performs the clear as a low-priority task.

filetype — The word filetype, followed by symbol which specifies an audio file format, sets the file type used by the **buffer~** object. The default file type is AIFF.Supported file types are identified as follows:

aiff	Apple Interchange File Format (default)
sd2	Sound Designer II (Macintosh only)
wave	WAVE
raw	raw
au	NeXT/Sun

import — The word import, followed by a filename, reads that file into **buffer~** immediately if it exists in Max's search path without opening the Open Document dialog box. Without a filename, import brings up an Open Document dialog box allowing you to choose a file. The imported file retains the sampling rate and word size of the original file, but looping points and markers are not imported. 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 **buffer~** object uses QuickTime to convert a media file (including MP3 files) into the sample memory of a **buffer~**, and requires that QuickTime be installed on your system. If you are using Max on Windows, we recommend that you install QuickTime and choose a complete install of all optional components.

Since the import message uses QuickTime, which specifies units of time for all files as 1/600 of a second rather than milliseconds, importing is not guaranteed to start at the specified offset with millisecond accuracy. 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 audio 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 audio 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.

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.

open Opens the **buffer~** sample display window or brings it to the front if it is already open.

read Reads an AIFF, Next/Sun, WAV file, or Sound Designer II file (Macintosh only) into the sample memory of the **buffer~**. The word read, followed by a filename, reads that file into **buffer~** immediately if it exists in Max's search path without opening the Open Document dialog box. Without a filename, read brings up a standard Open Document dialog box allowing you to choose a file. 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 audio 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 audio 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 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 audio files. Additional arguments specify starting time, duration, and number of channels as with the read message.

samptype In left inlet: The word samptype, followed by a symbol, specifies the sample type to use when interpreting an audio file's sample data (thus overriding the audio file's actual sample type). This is sometimes called "header munging."

The following types of sample data are supported:

int8 8-bit integer
int16 16-bit integer
int24 24-bit integer
int32 32-bit integer

float32 32-bit floating-point
float64 64-bit floating-point
mulaw 8-bit μ-law encoding
alaw 8-bit a-law encoding

set The word set, 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.

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 audio file.

wclose Closes the **buffer~** sample display window if it is open.

write Saves the contents of **buffer~** into an audio 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 folder, using the symbol as the filename. Unless you change the format with the Format pop-up menu in the standard Save As dialog box, the file will be saved in the format specified by the most recently received filetype message, or the file type of the most recently opened audio file. By default, **buffer~** saves in AIFF format.

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

writeau Saves the contents of the **buffer~** as a NeXT/Sun file. A standard Save As dialog is opened for naming the file unless the word writeau is followed by a symbol, in which case the file is saved in the current default folder, using the symbol as the filename.

writeraw Saves the contents of the **buffer~** as a raw file with no header. The default sample format is 16-bit, but the output sample format can be set with the samptype message. A standard Save As dialog is opened for naming the file unless the word writeraw is followed by a symbol, in which case the file is saved in the current default folder, using the symbol as the filename.

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

writewave — Saves the contents of the **buffer~** into a WAV file. A standard Save As dialog is opened for naming the file unless the word writewave is followed by a symbol, in which case the file is saved in the current default folder, using the symbol as the filename.

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

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

Arguments

symbol — Obligatory. The first argument is a name used by other objects to refer to the **buffer~** to access its contents.

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

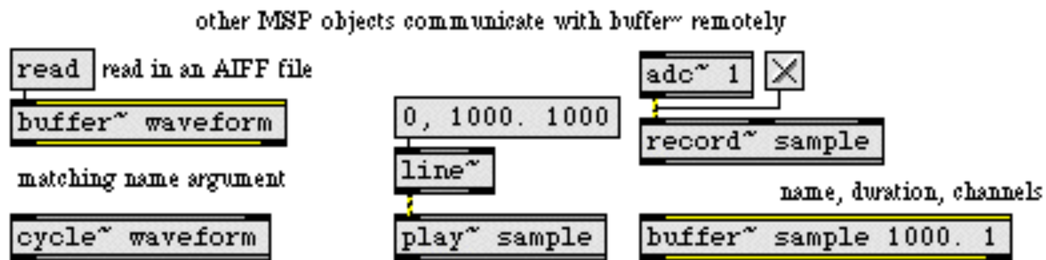
float or int — Optional. After the optional filename argument, a duration may be provided, in milliseconds, to set the size of the **buffer~**, which limits the amount of sound that will be stored in it. (A new duration can be specified as part of a read message, however.) If no duration is typed in,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

2d.wave~	Two-dimensional wavetable
buffir~	Buffer-based FIR filter
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 audio file from disk
sfrecord~	Record to audio file on disk
wave~	Variable-size wavetable
Tutorial 3	Fundamentals: Wavetable oscillator
Tutorial 12	Synthesis: Waveshaping
Tutorial 13	Sampling: Recording and playback

buffir~

buffer~-based FIR filter

The **buffir~** object implements a finite impulse response (FIR) filter that performs the convolution of an input signal and a set of coefficients which are derived from the samples stored in a **buffer~** object (referred to below as the filter **buffer~**) using the following equation:

$$y_n = b_0 x_n + b_1 x_{n-1} + b_2 x_{n-2} + \dots + b_q x_{n-q}$$

$$y_n = \sum_{j=0}^{q} b_j x_{n-j}$$

Input

signal — In left inlet: The signal to be convolved with samples from the **buffer~**.

In middle inlet: The offset (in samples) into the filter **buffer~** from which the **buffir~** object begins to read.

In right inlet: The size of the slice from the filter **buffer~** which is used to filter the input signal, in samples. The maximum is 256.

int or float — In middle inlet: The offset into the filter **buffer~** from which **buffir~** begins to read, in samples.

In right inlet: The size (in samples) of the slice from the filter **buffer~** which is used to filter the input signal (the maximum is 256).

clear — The word clear erases (zeroes) the current input history for the filter.

set — The word set, followed by the name of a **buffer~** object, an int which specifies sample offset, and an optional int which specifies a number of channels, specifies the name of a **buffer~** object which **buffir~** uses to filter its input signal.

Arguments

symbol — Obligatory. The name of a **buffer~** object which **buffir~** uses to filter the input signal.

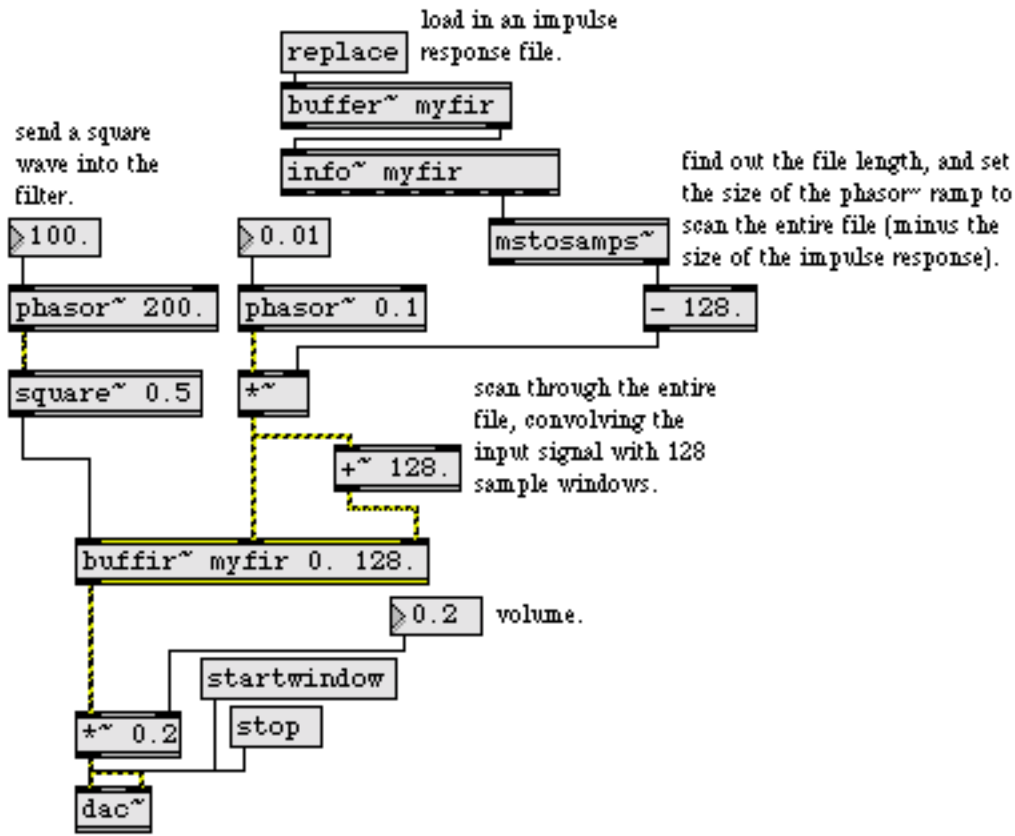
int or float — Optional. The offset, in samples, into the **buffer~** object before **buffir~** begins reading samples to construct the filter. The default is 0.

int or float — Optional. The size, in samples, of the slice in the **buffer~** which **buffir~** will use for the filter. The default is 0.

Output

signal — The filtered signal, based on a convolution of the input signal with samples in the **buffer~**.

Examples



buffir~ *lets you use slices of a* **buffer~** *as an impulse response for an FIR filter*

See Also

biquad~	Two-pole, two-zero filter
buffer~	Store audio samples

capture~

Store a signal to view as text

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.

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

open — Causes an editing and viewing window for the **capture~** object to become visible. The window is also brought to the front.

wclose — Closes the window associated with the **capture~** object.

(mouse) — Double-clicking on **capture~** opens a window for viewing and editing its contents. The numbers in the editing window can be copied and pasted into a graphic **buffer~** editing window.

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. 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.

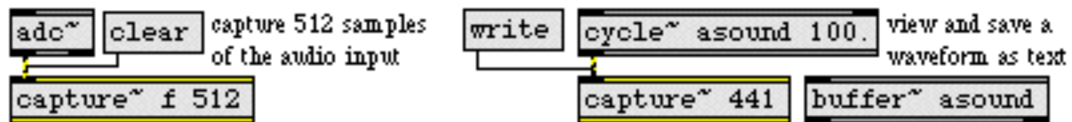
Output

None.

capture~

Store a signal to view as text

Examples



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

See Also

scope~	Signal oscilloscope

Input

signal — In left inlet: The real part of a frequency domain signal (such as that created by the **fft~** or **fftin~** objects) to be converted to a polar-coordinate signal pair consisting of amplitude and phase values.

In right inlet: The imaginary part of a frequency domain signal (such as that created by the **fft~** or **fftin~** objects) to be converted to a polar-coordinate signal pair consisting of amplitude and phase values.

Arguments

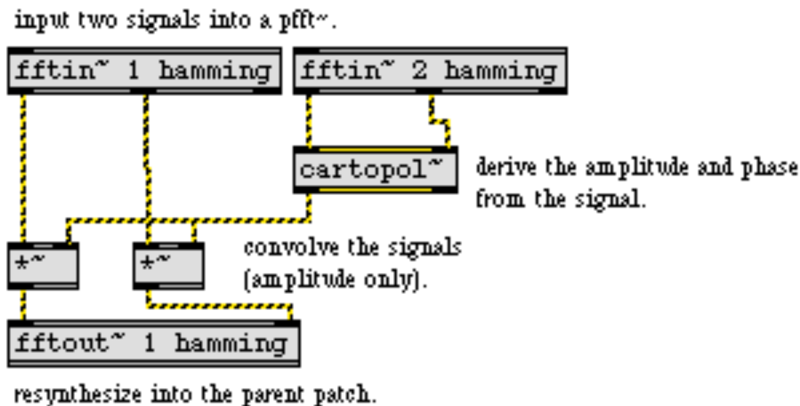
None.

Output

signal — Out left outlet: The magnitude (amplitude) of the frequency bin represented by the current input signals.

Out right outlet: The phase, expressed in radians, of the frequency bin represented by the current input signals. If only the left outlet is connected the phase computation will be bypassed, reducing the intensity of the computation.

Examples



Use **cartopol~** *to get amplitude/phase data from the real/imaginary data pair that* **fftin~** *outputs*

See Also

cartopol	Cartesian to Polar coordinate conversion
fft~	Fast Fourier transform
fftin~	Input for a patcher loaded by **pfft~**
fftinfo~	Report information about a patcher loaded by **pfft~**
fftout~	Output for a patcher loaded by **pfft~**
frameaccum~	Compute "running phase" of successive phase deviation frames
framedelta~	Compute phase deviation between successive FFT frames
ifft~	Inverse Fast Fourier transform
pfft~	Spectral processing manager for patchers
poltocar	Polar to Cartesian coordinate conversion
poltocar~	Signal Polar to Cartesian coordinate conversion
vectral~	Vector-based envelope follower
Tutorial 26	Frequency Domain Signal Processing with **pfft~**

change~

Report signal direction

Input

signal   Any signal.

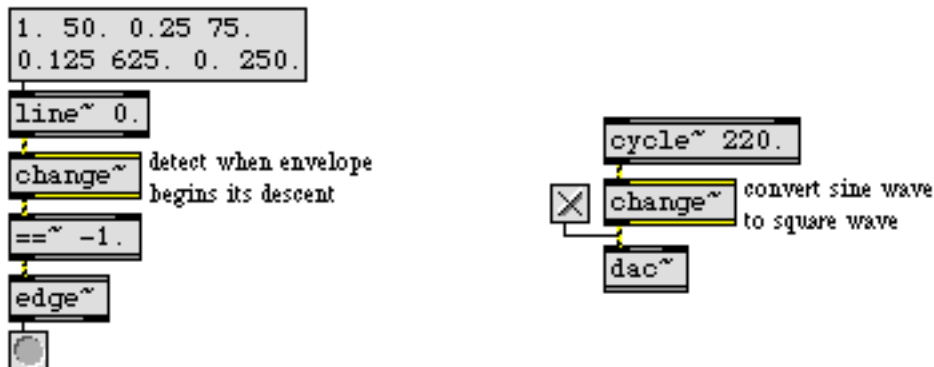
Arguments

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
zerox~	Zero-cross counter and transient detector

click~

Create an impulse

Input

bang Sends an impulse out the **click~** object's outlet. The default impulse consists of a single value (1.0), followed by a zero value.

set The word set, followed by a list of floating-point values in the range 0.0-1.0, specifies a impulse (i.e., a small wavetable) whose length is determined by the number of list elements. The maximum size for the list is 256 items.

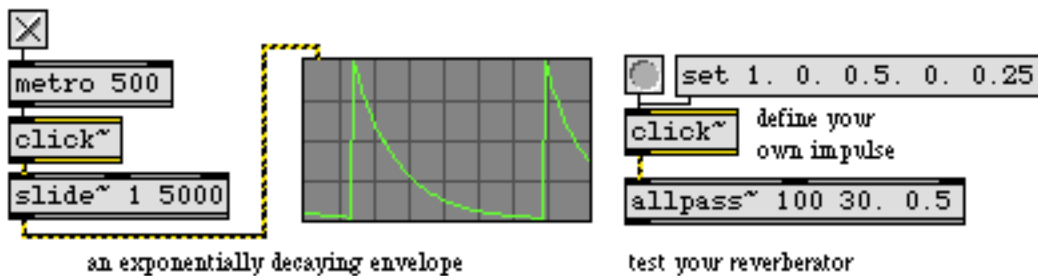
Arguments

list Optional. A list can be used to define the contents of a wavetable used for the impulse (see the set message). The maximum number of arguments is 256.

Output

signal An impulse.

Examples



Trigger an impulse signal

See Also

buffer~	Store a sound sample
buffir~	buffer-based FIR filter
line~	Linear ramp generator

clip~

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.

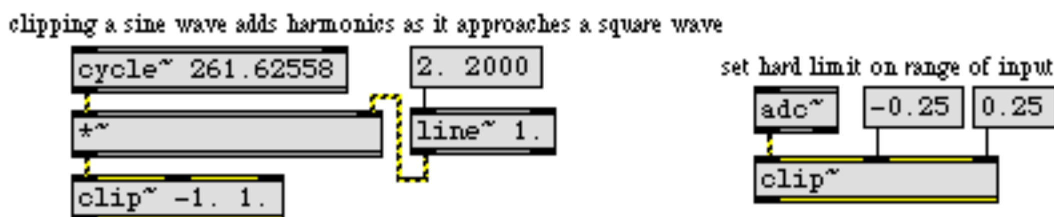
Arguments

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
trunc~	Truncate fractional signal values

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.

list — The three parameters can be provided as a list in the left inlet. The first number in the list is the delay time D, the next number is coefficient a, and the third number is coefficient b. If a signal is connected to a given inlet, the coefficient supplied in the list for that inlet is ignored.

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

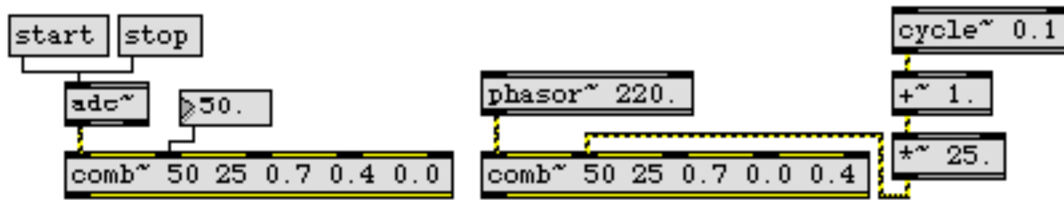
Arguments

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 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
teeth~	Comb filter with feedforward and feedback delay control

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.

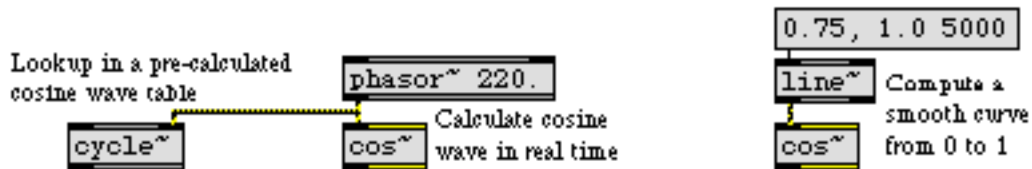
Arguments

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

acos~	Signal arc-cosine function
acosh~	Signal hyperbolic arc-cosine function
asin~	Signal arc-sine function
asinh~	Signal hyperbolic arc-sine function
atan~	Signal arc-tangent function
atanh~	Signal hyperbolic arc-tangent function
atan2~	Signal arc-tangent function (two variables)
cosh~	Signal hyperbolic cosine function
cosx~	Signal cosine function
cycle~	Table lookup oscillator
phasor~	Sawtooth wave generator
sinh~	Signal hyperbolic sine function
sinx~	Signal sine function
tanh~	Signal hyperbolic tangent function
tanx~	Signal tangent function
trapezoid~	Trapezoidal wavetable
triangle~	Triangle/ramp wavetable
wave~	Variable-size wavetable
2d.wave~	Two-dimensional wavetable

cosh~

Input

signal Input to a hyperbolic cosine function.

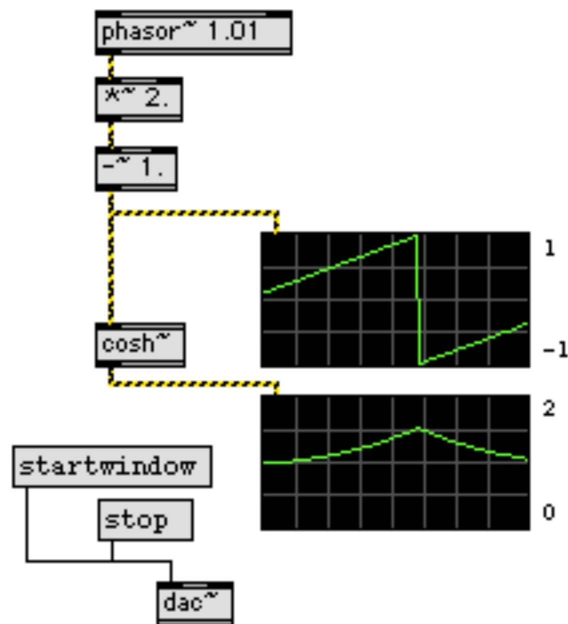
Arguments

None.

Output

signal The hyperbolic cosine of the input.

Examples



Exciting nautical motif audio control signals call for the cosh~ *object*

See Also

acos~	Signal arc-cosine function
acosh~	Signal hyperbolic arc-cosine function
asin~	Signal arc-sine function
asinh~	Signal hyperbolic arc-sine function
atan~	Signal arc-tangent function
atanh~	Signal hyperbolic arc-tangent function
atan2~	Signal arc-tangent function (two variables)
cos~	Signal cosine function (0-1 range)
cosx~	Signal cosine function
sinh~	Signal hyperbolic sine function
sinx~	Signal sine function
tanh~	Signal hyperbolic tangent function
tanx~	Signal tangent function

Input

signal — Output from a cosine function. Unlike the **cos~** object, whose output is based around 1 and intended for use as a lookup table with the **phasor~** object, the **cosx~** object is a true π-based function.

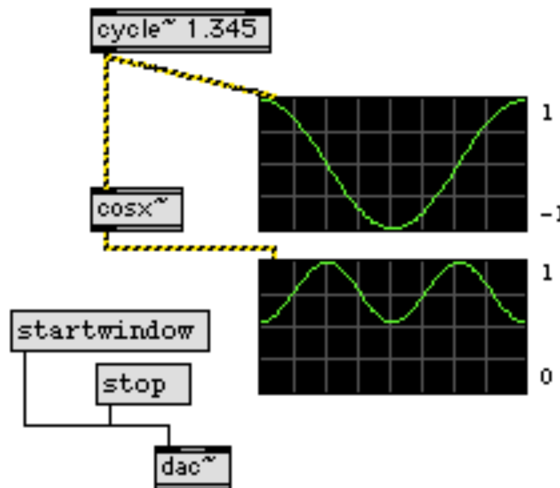
Arguments

None.

Output

signal — The cosine of the input.

Examples



cosx~ *can make your audio control signals less jumpy and more bumpy*

See Also

acos~	Signal arc-cosine function
acosh~	Signal hyperbolic arc-cosine function
asin~	Signal arc-sine function
asinh~	Signal hyperbolic arc-sine function
atan~	Signal arc-tangent function
atanh~	Signal hyperbolic arc-tangent function
atan2~	Signal arc-tangent function (two variables)
cos~	Signal cosine function (0-1 range)
cosh~	Signal hyperbolic cosine function
sinh~	Signal hyperbolic sine function
sinx~	Signal sine function
tanh~	Signal hyperbolic tangent function
tanx~	Signal tangent function

count~

Signal sample counter

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.

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

list — In left inlet: A list consisting of four numbers can be used to specify the behavior of the **count~** object. The first and second numbers specify the minimum and maximum values for the count, the third number specifies whether the **count~** object is off (0) or on (1) initially, and the fourth number sets the autoreset flag (see the autoreset message below).

float — In any inlet: Converted to int.

autoreset — In left inlet: The word autoreset, followed by a nonzero number, resets the counter to the minimum value when audio is turned on.

min — In left inlet: The word min, followed by a number, sets the count minimum on next loop without immediately affecting output.

set — In left inlet: The word set, followed by a number, sets the count minimum on the next loop without immediately affecting output.

stop — In left inlet: Causes **count~** to output a signal with its current minimum value.

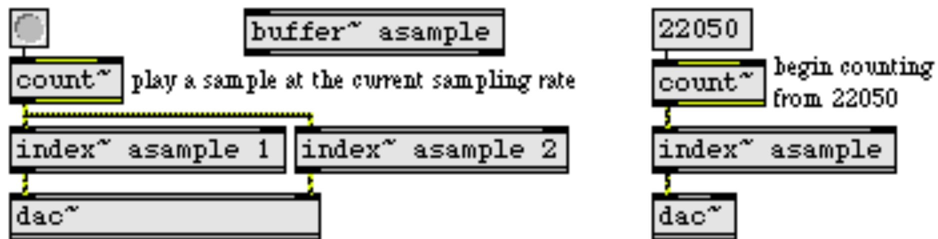
Arguments

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. The third argument specifies whether the **count~** object is off (0) or on (1) initially. The fourth argument sets the autoreset state of the object (see the autoreset message above).

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
+=~	Signal accumulator
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~	Linear 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.This is the case for the **cycle~** object's default cosine waveform. If this is what you want for other waveforms, you should make the 513th sample the same as the 512th sample, or omit the 513th 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.

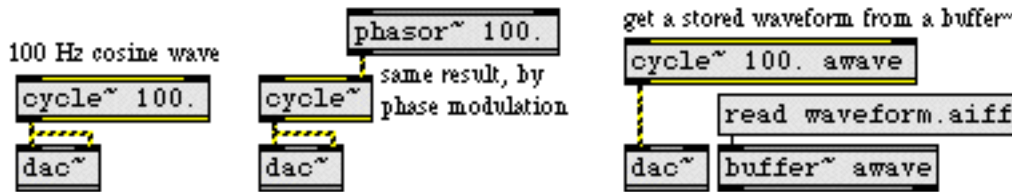
symbol — Optional. The name of a **buffer~** object used to store the oscillator's wavetable. If a float or int frequency argument is present, the **buffer~** name follows the frequency. (No frequency argument is required, however.) If no **buffer~** name is given, **cycle~** uses a stored cosine wave.

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 audio samples
buffir~	Buffer-based FIR filter
cos~	Cosine function
line~	Linear ramp generator
phasor~	Sawtooth wave generator
trapezoid~	Trapezoidal wavetable
triangle~	Triangle/ramp wavetable
wave~	Variable-size wavetable
2d.wave~	Two-dimensional wavetable
Tutorial 2	Fundamentals: Adjustable oscillator
Tutorial 3	Fundamentals: Wavetable oscillator

dac~

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.

open — Opens the DSP Status window.

set — In any inlet: The word set, followed by a number, sets the logical output channel for the signal inlet in which the set message was received. For instance, sending set 3 to the left inlet of **dac~** makes the signal coming in the left inlet to output to logical output channel 3.

Note that if the audio is on and you use the set message to change a **dac~** to use logical channels that are not currently in use, no sound will be heard from these channels until the audio is turned off and on again. For example, if you have a **dac~** object with arguments 1 2 3 4 and signals are only connected to the two leftmost inlets (for channels 1 and 2), the message set 1 3 will not immediately route the leftmost audio signal to logical channel 3, because it is not currently in use. A method to get around this is to connect a **sig~** 0 to each channel of a **dac~** you plan on using for a set message. At this point, you might as well use a **matrix~** or **switch~** object to do something similar before the audio signal reaches the **dac~**.

start — Turns on 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.

stop — Turns off audio processing in all loaded patches.

wclose — Closes the DSP Status window if it is open.

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. You can create a **dac~** object that uses one or more audio output channel numbers between 1 and 512. These numbers refer to *logical channels* and can be dynamically reassigned to physical device channels of a particular driver using either the DSP Status window, its I/O Mappings subwindow, or an **adstatus** object with an output keyword argument.Arguments, If the computer's built-in audio hardware is being used, there will be two input channels available. Other audio drivers and/or devices may have more than two channels. If no argument is typed in, **dac~** will have two inlets, for input channels 1 and 2.

Output

None. The signal received in the inlet is sent to its assigned logical audio output channel, which is mapped to a physical device output channel in the DSP Status window.

Examples

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

See Also

adc~	Audio input and on/off
adstatus	Access audio driver output channels
ezadc~	Audio on/off; analog-to-digital converter
ezdac~	Audio output and on/off button
Audio I/O	Audio input and output with MSP
Tutorial 1	Fundamentals: Test tone

degrade~

Signal quality reducer

Input

signal — In left inlet: The signal to be degraded.

float — In middle inlet: The ratio of frequency at which the input signal is resampled, effectively reducing its sampling rate. This ratio is the resampling rate divided by the system sampling rate. For example, if MSP's current sampling rate is 44100 Hz, and the ratio is 0.75, the effective sampling rate of the output signal will be 33075 Hz.

int — In right inlet: The number of bits used to quantize the input signal. This value must be in the range 1-24. Fewer bits mean lower signal quality.

Arguments

float — Optional. The first argument sets the resampling frequency ratio, as described above. If this argument is not supplied, the default value is 1.0.

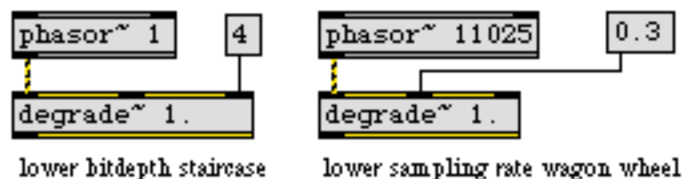
int — Optional. The second argument sets the number of bits used to quantize the input signal. If this argument is not supplied, the default value is 24.

Output

signal — The output signal is the input signal after being resampled and quantized. Note that this object deliberately does not use any interpolation when resampling, nor any dithering when quantizing. It is intended for creating “low-fi” effects.

Note: Use caution when listening to the output of this object. Quantizing to a small number of bits can create very loud, noisy signals.

Examples



Change a signal's effective sampling rate and bit depth

See Also

downsamp~	Downsample a signal
round~	Round an input signal value

delay~

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~**.

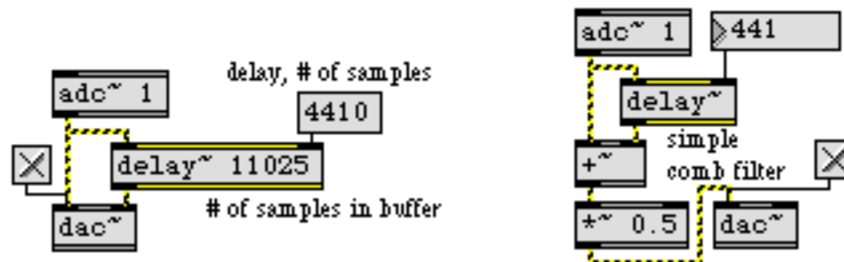
Arguments

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 differences between **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.

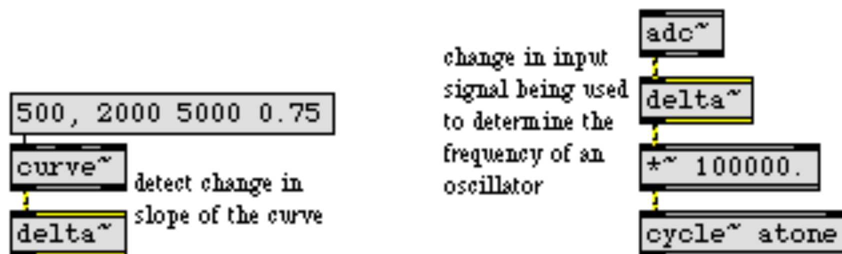
Arguments

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

average~	Multi-mode signal average
avg~	Signal average

deltaclip~ limits the change between samples in an incoming signal. It is similar to the **clip~** object, but it limits amplitude changes with respect to slope rather than amplitude.

Input

signal — In left inlet: Any signal.

float or int — In middle inlet: Minimum slope for the rate of change of the output signal. The minimum slope is typically negative.

In right inlet: Maximum slope for the rate of change of the output signal. The maximum slope is typically positive.

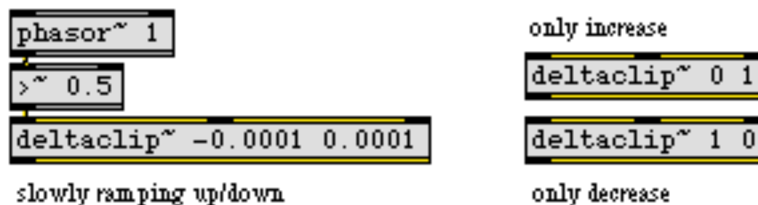
Arguments

float — Optional. Initial minimum and maximum slope values for the rate of change 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, with its change limited by the minimum and maximum slope values.

Examples



Limit a signal's rate of change

See Also

clip~ — Limit signal amplitude

downsamp~

Downsample a signal

Input

signal — In left inlet: A signal to be downsampled. The **downsamp~** object samples and holds a signal received in the left inlet at a rate set by an argument to the object of the value received in the right inlet, expressed in samples. No interpolation of the output is performed.

In right inlet: The rate, in samples, at which the incoming signal is to be downsampled.

int or float — In right inlet: Sets the sample rate used to downsample the input signal. You can specify the number of samples with floating-point values, but the **downsamp~** object will sample the input at most as frequently as the current sampling rate.

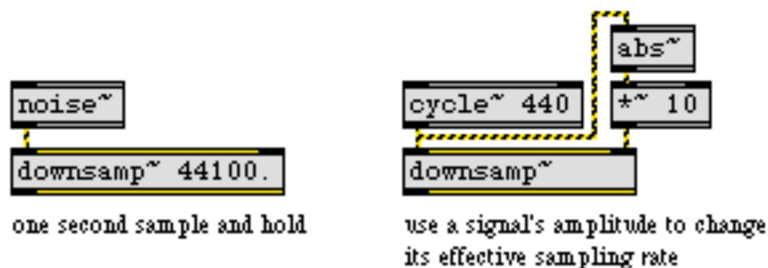
Arguments

int or float — Optional. Sets the sample rate.

Output

signal — The input signal, resampled at the rate set by argument or by the value received in the right inlet.

Examples



Sample and hold every n samples

See Also

degrade~	Signal quality reducer
sah~	Sample and hold

dspstate~

Report current DSP settings

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 the **dspstate~** object'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 second outlet: Sampling rate of the connected signal or the global sampling rate.

int — Out third outlet: Current DSP signal vector size.

int — Out fourth outlet: Current I/O 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 25	Analysis: Using the FFT

dsptime~

Report milliseconds of audio processed

Input

bang When **dsptime~** receives a bang, it reports the number of milliseconds corresponding to the number of audio samples that have currently been processed.

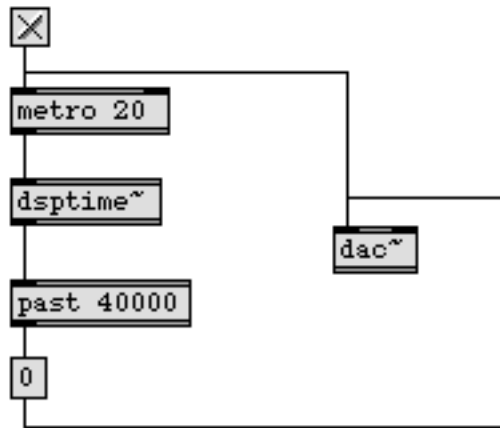
Arguments

None.

Output

float The number of milliseconds corresponding to the number of audio samples that have currently been processed. The value is based on the processed audio sample count, not the real time of the millisecond timer. This means you can use the **dsptime~** object as a sort of clock in conjunction with the NonRealTime audio driver.

Examples



Shut audio processing off automatically after 40 seconds have been processed

See Also

adstatus	Access audio driver output channels
dspstate~	Report current DSP setting

edge~

Input

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

Arguments

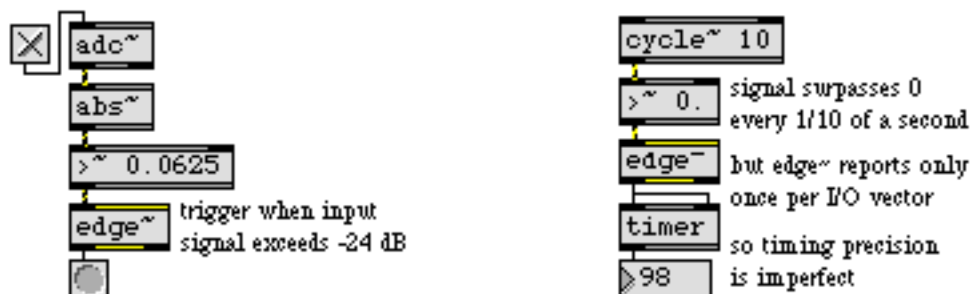
None.

Output

bang — Out left outlet: Sent when the input signal changes from zero to non-zero. The minimum time between bang messages will not be shorter than the minimum scheduler interval, which is generally equal to the signal vector size, but may be larger if Scheduler in Audio Interrupt mode is not enabled.

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 value
zerox~	Zero-cross counter and transient detector

Input

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

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

open Opens the DSP Status window. The window is also brought to the front.

start Turns on audio processing in all loaded patches.

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.

stop Turns off audio processing in all loaded patches.

wclose Closes the DSP Status window.

Arguments

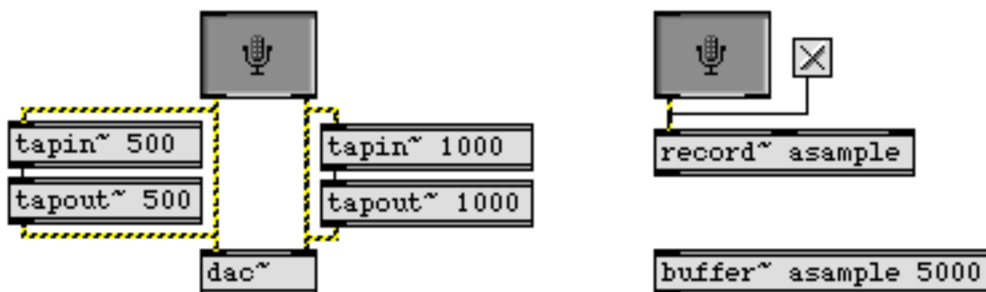
None.

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

adstatus	Access audio driver output channels
ezdac~	Audio output and on/off button
adc~	Audio input and on/off

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 highlighted.

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

open — Opens the DSP Status window. The window is also brought to the front.

start — Turns on audio processing in all loaded patches.

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.

stop — Turns off audio processing in all loaded patches.

stopwindow — Turns off audio processing only in the patch in which this **ezdac~** is located, and in subpatches of that patch.

wclose — Closes the DSP Status window.

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

adstatus	Access audio driver output channels
ezadc~	Audio input and on/off button
adc~	Audio output and on/off
Tutorial 3	Fundamentals: Wavetable oscillator

The **fffb~** object implements a bank of bandpass filter objects, each of which is similar to the **reson~** filter object. An input signal is applied to all filters, and the outputs of each filter are available separately. This object is more efficient than using a number of **reson~** objects, but for the sake of speed does not accept signals for parameter changes.

Input

signal — The signal present at the left inlet is sent to all of the filters.

freq — In left inlet: The word freq, followed by a list consisting of an int and one or more floats, sets the center frequencies of the filters starting with the filter whose index is given by the first number. This filter's frequency is set to the second number in the list. Any following numbers in the list set the frequencies of filters following the first designated one. Indices are zero-based.

For example, the message freq 3 1974.0 333.0 1234.0 sets the frequency of the fourth filter to 1974Hz, the fifth filter to 333Hz, and the sixth filter to 1234Hz.

freqAll — in left inlet: The word freqAll, followed by a float, sets the center frequencies of all of the filters to the given floating-point value.

freqRatio — In left inlet: The word freqRatio, followed by a list of two or more numbers sets the center frequency of the first filter to the first value in the list, and sets the frequencies of the remaining filters by repeatedly multiplying the first value by the second, so that the ratio of frequencies of successive filters is the second value—for example, the message freqRatio 440. 2. sets the frequency of the first filter to 440Hz, the frequency of the second to 880Hz, the frequency of the third to 1760Hz, and so on.

If the second item in the list is the letter H rather than a number, the filters will be tuned in a harmonic series. For example, the message freqRatio 100 H sets the frequencies of the filters to 100Hz, 200Hz, 300Hz, 400Hz, and so on.

gain — In left inlet: The word gain, followed by a list consisting of an int and one or more floats, sets the gains of the filters starting with the filter whose index is given by the first number. This filter's gain is set to the second number in the list. Any following numbers in the list set the gains of filters following the first designated one. Indices are zero-based.

gainAll — In left inlet: The word gainAll, followed by a float, sets the gain of all of the filters to the given floating-point value.

Q — In left inlet: The symbol Q, followed by a list consisting of an int and one or more floats, sets the Q factors of the filters, starting with the filter whose index is given by the first number. This filter's Q factor is set to the second number in the list. Any following numbers in the list set the Q factors of filters following the first designated one. Indices are zero-based.

QAll — In left inlet: The word QAll, followed by a float, sets the Q of all of the filters to the given floating-point value.

Arguments

int — Obligatory. The first argument specifies the number of filters.

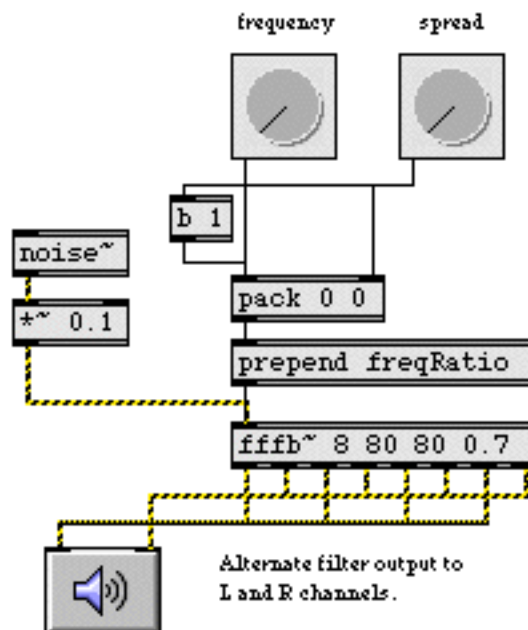
float — Optional. Three additional float arguments may be used to specify the frequency of the first filter, the ratio of frequencies between successive filters, and the Q factor for all of the filters.

symbol — Optional. If you use the letter H as the second argument rather than a float, the filters will be tuned to a harmonic series rather than with ratios of frequencies.

Output

signal — The output of each filter is provided at a separate outlet. The leftmost outlet is the output of the first filter.

Examples



Stereo expansion by altering the base frequency and frequency ratio

See Also

reson~ — Resonant bandpass filter

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 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

cartopol	Cartesian to Polar coordinate conversion
cartopol~	Signal Cartesian to Polar coordinate conversion
fftin~	Input for a patcher loaded by **pfft~**
fftinfo~	Report information about a patcher loaded by **pfft~**
fftout~	Output for a patcher loaded by **pfft~**
frameaccum~	Compute "running phase" of successive phase deviation frames
framedelta~	Compute phase deviation between successive FFT frames
ifft~	Inverse Fast Fourier transform
index~	Sample playback without interpolation
pfft~	Spectral processing manager for patchers
poltocar	Polar to Cartesian coordinate conversion
poltocar~	Signal Polar to Cartesian coordinate conversion
vectral~	Vector-based envelope follower
Tutorial 25	Analysis: Using the FFT

fftin~

Input for a patcher loaded by **pfft~**

The **fftin~** object provides an signal input to a patcher loaded by a **pfft~** object; it won't do anything if you try to use it anywhere other than inside a patcher loaded by the **pfft~** object. Where the **pfft~** object manages the windowing and overlap of the incoming signal, **fftin~** applies the windowing function (the envelope) and performs the Fast Fourier Transform.

Input

signal — Dummy inlet for the connection of a **begin~** object. The signal input for an **fftin~** object is an inlet in the **pfft~** subpatcher which contains the object.

Arguments

int — Obligatory. Determines the inlet number of the **pfft~** which will be routed into the **fftin~** object. Inlet assignment starts at one, for the leftmost inlet in the **pfft~**. Multiple **fftin~** objects will typically have different inlet numbers.

symbol — Specifies the window envelope function the **fftin~** object will apply to overlapping FFTs on the input signal. The options are *square* (i.e. no window envelope), *hanning* (the default), *triangle*, *hamming* and *blackman* (Note: The Blackman window should be used with an overlap of 4 or more). If the symbol nofft is used, then the **fftin~** object will not use a windowing envelope and will not perform a Fast Fourier Transform— it will echo the first half of its input sample window to its real output and the second half of its input sample window to its imaginary output. This can allow you to input raw control signals from outside the parent patcher through inlets in the **pfft~** object, provided its overlap is set to 2. Other overlap values may not yield useful results.

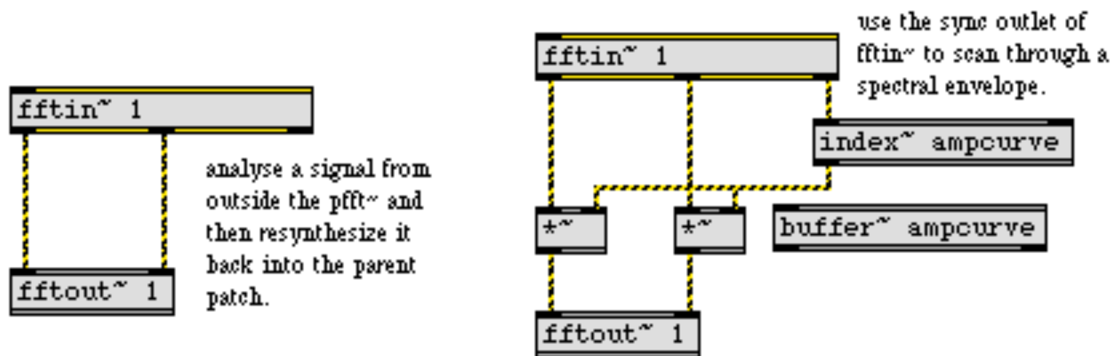
Output

signal — Out left outlet: This output contains the real-values resulting from the Fast Fourier transform performed on the corresponding inlet of the **pfft~**. This output frame is only half the size of the parent **pfft~** object's FFT size because the spectrum of a real input signal is symmetrical and therefore half of it is redundant. The real and imaginary pairs for one spectrum are called a spectral frame.

Out middle outlet: This output contains the imaginary-values resulting from the the Fast Fourier transform performed on the corresponding inlet of the **pfft~**. This output frame is only half the size of the parent **pfft~** object's FFT size because the spectrum of a real input signal is symmetrical and therefore half of it is redundant. The real and imaginary pairs for one spectrum are called a spectral frame.

Out right outlet: A stream of samples corresponding to the index of the current bin whose data is being sent out the first two outlets. This is a number from 0-(frame size - 1). The spectral frame size inside a **pfft~** object's subpatch is equal to half the FFT window size.

Examples



fftin~ *outputs a frequency/domain signal pair and a sync signal that indicates the bin number*

See Also

cartopol	Cartesian to Polar coordinate conversion
cartopol~	Signal Cartesian to Polar coordinate conversion
fft~	Fast Fourier transform
fftinfo~	Report information about a patcher loaded by **pfft~**
fftout~	Output for a patcher loaded by **pfft~**
frameaccum~	Compute “running phase” of successive phase deviation frames
framedelta~	Compute phase deviation between successive FFT frames
ifft~	Inverse Fast Fourier transform
in	Message input for a patcher loaded by **poly~** or **pfft**
out	Message output for a patcher loaded by **poly~** or **pfft~**
pfft~	Spectral processing manager for patchers
poltocar	Polar to Cartesian coordinate conversion
poltocar~	Signal Polar to Cartesian coordinate conversion
vectral~	Vector-based envelope follower
Tutorial 26	Frequency Domain Signal Processing with **pfft~**

fftinfo~

Report information about a patcher loaded by **pfft~**

Input

bang Causes the FFT window size, the FFT frame size (i.e., the signal vector size inside the patcher loaded by **pfft~**), and the FFT hop size to be sent out the object's outputs.

Arguments

None.

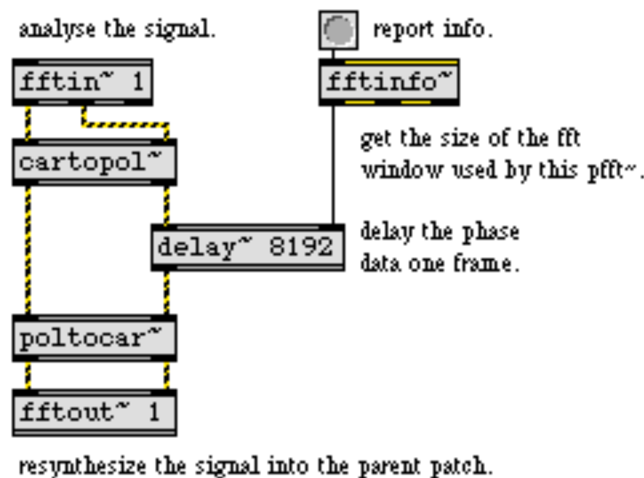
Output

int Out left outlet: The current FFT window size specified by argument to the **pfft~** object.

Out middle outlet: The current spectral frame size (half the FFT window size).

Out right outlet: The current FFT hop size (i.e., the window size divided by the overlap).

Examples



fftinfo~ *reports information about the FFT subpatcher in which it is located*

See Also

cartopol	Cartesian to Polar coordinate conversion
cartopol~	Signal Cartesian to Polar coordinate conversion
fft~	Fast Fourier transform
fftin~	Input for a patcher loaded by **pfft~**
fftout~	Output for a patcher loaded by **pfft~**
frameaccum~	Compute "running phase" of successive phase deviation frames
framedelta~	Compute phase deviation between successive FFT frames
ifft~	Inverse Fast Fourier transform
pfft~	Spectral processing manager for patchers
poltocar	Polar to Cartesian coordinate conversion
poltocar~	Signal Polar to Cartesian coordinate conversion
vectral~	Vector-based envelope follower
Tutorial 25	Analysis: Using the FFT
Tutorial 26	Frequency Domain Signal Processing with **pfft~**

fftout~

Output for a patcher loaded by **pfft~**

The **fftout~** object provides an signal output to a **pfft~** object; it won't do anything if you try to use it anywhere other than inside a patcher loaded by the **pfft~** object. The **fftout~** object performs an inverse Fast Fourier Transform and applies a windowing function (an envelope), allowing the **pfft~** object to manage the overlap-add of the output signal windows.

Input

signal — In left inlet: The real part of a signal that will be inverse-transformed back into the time domain.

In right inlet: The imaginary part of a signal that will be inverse-transformed back into the time domain.

Note that the real and imaginary inlets of **fftout~** expect only the first half of the spectrum, as output by **fftin~**. This half-spectrum is called a spectral frame in **pfft~** terminology.

Arguments

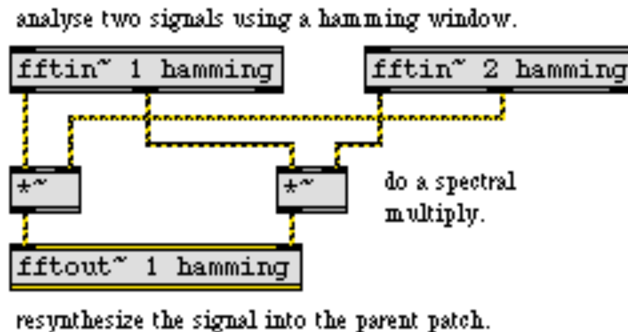
int — Obligatory. Determines the outlet number in the **pfft~** which will receive the output of the **fftout~** object. Outlet assignments start at 1 for the leftmost outlet of **pfft~**. Multiple **fftout~** objects will typically have different outlet numbers.

symbol — Optional. Tells **fftout~** which window envelope function to use when overlapping fft's on the input signal. The options are square (i.e. no window envelope), hanning (the default), and hamming. If the argument nofft is used, then the **fftout~** will echo its input signal to its output without performing a Fast Fourier transform. This allows you to output raw control signals from the **pfft~** to the parent patcher. Note that when the nofft option is used, overlap-adding is still being performed to create the output signal.

Output

signal — The **fftout~** object transforms frequency domain signals back into the time domain, at which point they are overlap-added and output by the corresponding outlet in the **pfft~** object in which the subpatcher is loaded. The **fftout~** object itself has no outlets.

Examples



fftout~ *converts frequency domain signal pairs into time domain signals and sends them to* **pfft~**

See Also

cartopol	Cartesian to Polar coordinate conversion
cartopol~	Signal Cartesian to Polar coordinate conversion
fft~	Fast Fourier transform
fftin~	Input for a patcher loaded by **pfft~**
fftinfo~	Report information about a patcher loaded by **pfft~**
frameaccum~	Compute "running phase" of successive phase deviation frames
framedelta~	Compute phase deviation between successive FFT frames
ifft~	Inverse Fast Fourier transform
out	Message output for a patcher loaded by **poly~** or **pfft~**
pfft~	Spectral processing manager for patchers
poltocar	Polar to Cartesian coordinate conversion
poltocar~	Signal Polar to Cartesian coordinate conversion
vectral~	Vector-based envelope follower
Tutorial 25	Analysis: Using the FFT
Tutorial 26	Frequency Domain Signal Processing with **pfft~**

filtergraph~

Graphical filter editor

The **filtergraph~** object is not a signal object per se, as it does not process audio signals by itself, but it does react to the current MSP current sampling rate in order to generate filter coefficients for the **biquad~** object from higher-level parameters such as frequency, amplitude and resonance (Q). Since the **filtergraph~** object needs to use the current sampling rate to calculate the filter response, Max/MSP must be using an audio driver in order for the object to properly display and calculate values.

The **filtergraph~** object was designed as both a display and a graphical user interface for a variety of second order (two-pole two-zero) filters implemented using the **biquad~** object. The horizontal axis of the **filtergraph~** object's display represents frequency (which can be displayed on either a linear or logarithmic scale), while its vertical axis represents amplitude. The curve displayed reflects the frequency response of the current filter model. The frequency response is essentially the amount that the filter will amplify or attenuate the frequencies present in an audio signal. The biquad~ object does the actual filtering, based on the coefficients that **filtergraph~** calculates and sends to it in a list. You probably do not need to interact with the coefficients themselves, but the mathematical equations are provided that describe their relationship to the higher-level parameters.

The *cutoff frequency* is the center frequency of the filter's activity. Its specific meaning is different for each filter type, but it can generally be identified as the transitional point of the graph's curve. In addition, it is marked in the display by a colored rectangle whose width corresponds to the bandwidth of the filter. The *bandwidth* is the range of a filter's effect, centered on the cutoff frequency. *Q* is another term for the same parameter of filter "width" although it is described in different units (octave reciprocals instead of plain Hz). Further names used to reference bandwidth include resonance, slope, S, and transitional band. For the most part, **filtergraph~** uses bandwidth or Q, which are inversely proportional to each other.

The interpretation of the gain parameter depends on the type of filter, but generally involves a linear scaling of values in the filter band, or across the entire spectrum.

High-level filter parameters can be changed by clicking and dragging on the object. By default, horizontal mouse dragging is mapped to cutoff frequency, and vertical mouse movement is mapped to gain (if gainmode is enabled). If the cursor is located directly over the edge of a filter band, however, the band rectangle is highlighted, indicating that clicking and dragging will map x-axis movement to adjust filter bandwidth, instead of cutoff frequency.

It is possible, especially in smaller **filtergraph~** objects, to create such a high Q value that the band is too narrow to click on it without selecting a bandwidth line for editing. For this and other purposes, double-clicking will reverse the mouse interpretation for the duration of that click/drag activity. Thus, if you double-click on a bandwidth line of a narrow filter, the mouse will be set to edit cutoff frequency, instead of resonance.

Input

float — In 1st-5th inlets: When in display mode, a float in one of the first five inlets changes the current value of the corresponding biquad~ filter coefficient (*a0*, *a1*, *a2*, *b1*, and *b2*, respectively), recalculates the filter's frequency response based on these coefficients and causes a list of the current filter coefficients to be output from the leftmost outlet.

In 6th inlet: Sets the center or cutoff frequency parameter for the filter and causes output.

In 7th inlet: Sets the gain parameter for the filter and causes output.

In 8th inlet: Sets the Q (resonance) or S (slope) parameter for the filter and causes output.

Note: Input to any one of the inlets will recalculate the current filter's graph and trigger the output.

int Converted to float.

list In left inlet: A list of five int values which correspond to **biquad~** filter coefficients sets the **filtergraph~** object's internal values for these coefficients and causes the object to output the list out its left outlet. If **filtergraph~** is in display mode, it will display the frequency response of the filter obtained from these coefficients.

in 6th inlet: A list of three values which correspond to center/cutoff frequency, gain and Q/S (resonance/slope), sets these values, recalculates the new filter coefficients and causes output. This is equivalent to the params message.

bang In left inlet: In display mode, bang causes the **filtergraph~** object to send its internally-stored biquad coefficients out the leftmost outlet. In the interactive filter modes, bang additionally causes the current filter parameters to be sent out their respective outlets (see Output).

aconstrain In left inlet: The word aconstrain, followed by two float values, allows you to constrain the amplitude values within the specified range. This is useful to constrain values obtained by clicking and dragging. Specifying aconstrain 0. 0. removes the limits.

amp In left inlet: The amp message sets a frequency amplitude display. It is equivalent to the spectrum 0 message.

autoout In left inlet: Toggles the automatic output on load feature. autoout 1 tells **filtergraph~** to automatically output its coefficients and parameters when a patch is loaded. **filtergraph~** saves its current state in a patcher. autoout 0 disables this feature. The default value is 1.

bandpass In left inlet: The word bandpass sets the filter type of the **filtergraph~** object to *bandpass* mode.It is equivalent to the mode 3 message. The frequency response of the filter is based on three parameters: *cf* (center frequency, or cutoff frequency) *gain*, and *Q* (resonance).

bandstop In left inlet: The word bandstop sets the filter type of the **filtergraph~** object to *bandstop* mode.It is equivalent to the mode 4 message. The frequency response of

the filter is based on three parameters: *cf* (center frequency, or cutoff frequency) *gain*, and *Q* (resonance).

brgb — In left inlet: The word brgb, followed by three numbers between 0 and 255, sets the color of the **filtergraph~** object background (i.e., the area above the filter curve) in RGB format. The default is 210 210 210.

cascade — In left inlet: The cascade message works in display mode only. The word cascade, followed by up to 24 groups of five float values corresponding to filter coefficients, will display a composite filter response which shows the multiplication of a group of biquad filters in cascade.

color — In left inlet: The word color, followed by a number from 0 to 15, sets the color of the **filtergraph~** object to one of the 16 object colors, which are also available using the Color submenu in the Object menu.

display — In left inlet: The word display sets the filter type of the **filtergraph~** object to display only. It is equivalent to the mode 0 message. In display mode, **filtergraph~** simply displays the frequency response for a set of five **biquad~** filter coefficients.

displaydot — In left inlet: The displaydot message, followed by a 0 or 1, toggles the display of the mousable bandwidth region when **filtergraph~** is in display mode. This allows you to use **filtergraph~** as an interface to design and display your own filter algorithms. The default is *disabled* (by default, display mode functions uniquely as a display).

domain — In left inlet: The domain message, followed by two integer frequencies in Hz, lets you change the frequency display range of the **filtergraph~**. The default display range is from 0 Hz to half the sampling rate (the Nyquist frequency).

frgb — In left inlet: The word frgb, followed by three numbers between 0 and 255, sets the color of the **filtergraph~** object foreground (i.e., the area below the filter curve) in RGB format. The default is 170 170 170.

fconstrain — In left inlet: The word fconstrain, followed by two float values, allows you to constrain the frequency values within the specified range. This is useful to constrain values obtained by clicking and dragging. Specifying fconstrain 0. 0. removes the limits.

fullspect — In left inlet: The word fullspect, followed by a 0 or 1, lets you select either a half-spectrum or full spectrum display. fullspect 0 (the default) specifies a half-spectrum from 0 Hz to the Nyquist frequency (half the sampling rate). fullspect 1 specifies a full (mirrored) spectrum from -Nyquist to +Nyquist (the spectrum is mirrored around 0 Hz). In full spectrum mode, the display has a red marker at DC (0 Hz).

gainmode In left inlet: The word gainmode, followed by a 0 or 1, toggles the gain parameter for the lowpass, highpass, bandpass, and bandstop filters. The traditional definitions of these filters have a fixed gain of 1.0, but by gain-enabling them, their amplitude response can be scaled without the additional use of a signal multiply at the filters output.The default is 0 (disabled).

highorder The highorder message works in display mode only. The word highorder, followed by a list of *n* groups of biquad filter "a" coefficients and *n-1* groups of biquad filter "b" coefficients, will display the response of an *nth* order filter.

highpass In left inlet: The word highpass sets the filter type of the **filtergraph~** object to *high-pass* mode.It is equivalent to the mode 2 message. The frequency response of the filter is based on three parameters: *cf* (center frequency, or cutoff frequency) *gain*, and *Q* (resonance) or *S* (slope - used for the shelving filters).

highshelf In left inlet: The word highshelf sets the filter type of the **filtergraph~** object to *highshelf* mode.It is equivalent to the mode 7 message. The frequency response of the filter is based on three parameters: *cf* (center frequency, or cutoff frequency) *gain*, and *S* (slope).

lin In left inlet: The lin message sets a linear frequency display scale. It is equivalent to the scale 0 message.

linmarkers In left inlet: The word linmarkers, followed by a list of up to 64 int values, will set markers for the linear frequency display (See the markers message). By default, the markers are set at ± SampleRate/4, SampleRate/2, and (3 * SampleRate)/4.

log In left inlet: The log message sets a log frequency display scale. It is equivalent to the scale 1 message.

logamp In left inlet: The logamp message, followed by a 0 or 1, sets the amplitude display scale. scale 0 sets a linear amplitude display (default), and scale 1 sets a log display scale.

logmarkers In left inlet: The word logmarkers, followed by a list of up to 64 int values, will set markers for the log frequency display (See the markers message). By default, the markers are set at± 50Hz, 500Hz and 5kHz at 44.1kHz. These values correspond to ± 0.007124, 0.071238, and 0.712379 radians for any sample rate.

lowpass In left inlet: The word lowpass sets the filter type of the **filtergraph~** object to *low-pass* mode.It is equivalent to the mode 1 message. The frequency response of the filter is based on three parameters: *cf* (center frequency, or cutoff frequency) *gain*, and *Q* (resonance).

lowshelf In left inlet: The word lowshelf sets the filter type of the **filtergraph~** object to *low-shelf* mode.It is equivalent to the mode 6 message. The frequency response of the

filter is based on three parameters: *cf* (center frequency, or cutoff frequency) *gain*, and *S* (slope).

markers — In left inlet: The word markers, followed by a list of up to 64 frequency values will place visual markers (vertical lines) at these frequencies behind the graph. The markers message will set the markers used for both linear and logarithmic frequency displays.

mode — In left inlet: The word mode, followed by a number from 0-7, sets the current filter type. The numbers and associated filter types are:

Number	*Filter type*
0	display only
1	lowpass
2	highpass
3	bandpass
4	bandstop
5	peaknotch
6	lowshelf
7	highshelf

In display mode, **filtergraph~** displays the frequency response for a set of five **biquad~** filter coefficients. In the other modes, it graphs the frequency response of a filter based on three parameters: *cf* (center frequency, or cutoff frequency) *gain*, and *Q* (resonance) or *S* (slope - used for the shelving filters).

mousemode — In left inlet: The word mousemode followed by two int arguments, specifies the interpretation of horizontal and vertical mouse movement. With one argument, only the horizontal mouse mode is affected. The mouse mode values are the same for both axes: (0 = off, 1 = normal, 2 = alternate).

For horizontal movement (specified by the first argument), normal behavior means that clicking on the filter band and dragging horizontally changes the filter's cutoff frequency. When set to the alternate mouse mode (2), horizontal movement affects Q, or resonance. When turned off (0), mouse activity along the x-axis has no effect.

For vertical movement (specified by the second argument), normal behavior means that the y-axis is mapped to gain during clicking and dragging activity. When the alternate mouse mode (2) is selected, vertical movement changes the Q (resonance) setting instead. When turned off (0), vertical mouse movement has no effect.

params — In left inlet: The word params, followed by three numbers specifying frequency, gain and Q, sets the filter parameters and triggers output.

peaknotch — In left inlet: The word peaknotch sets the filter type of the **filtergraph~** object to *peaknotch* mode.It is equivalent to the mode 5 message. The frequency response of the filter is based on three parameters: *cf* (center frequency, or cutoff frequency) *gain*, and *Q* (resonance).

phase — In left inlet: The phase message sets a frequency phase display. It is equivalent to the spectrum 1 message.

qconstrain — In left inlet: The word qconstrain, followed by two float values, allows you to constrain the Q (resonance) values within the specified range. This is useful to prevent Q settings that might be inappropriate in a specific context. It can also be used to "lock" Q to a specific value, by sending that value as both the minimum and the maximum (e.g., qconstrain 0.4 0.4). Specifying qconstrain 0. 0. removes the limits.

query — In left inlet: The word query, followed by a float value, will cause the amplitude and phase response of the current filter at that frequency to be sent out the rightmost outlet of the **filtergraph~** object as a list.

range — In left inlet: The range message, followed by a float value greater than 0, sets the amplitude display range of **filtergraph~**. The amplitude is displayed from 0 to the range value along the vertical axis of the graph. (default value 2.0)

rgb — In left inlet: The word rgb, followed by three numbers between 0 and 255, sets the color of the **filtergraph~** display. The background color for the object display will be automatically selected. The brgb, frgb, rgb2, rgb3, and rgb4 messages can be used to set the colors of individual portions of a **filtergraph~** object display.

rgb2 — In left inlet: The word rgb2, followed by three numbers between 0 and 255, sets the color of the **filtergraph~** object's curve line (i.e., the line that separated the areas above and below the filter curve) in RGB format. The default is 0 0 0 (black).

rgb3 — In left inlet: The word rgb3, followed by three numbers between 0 and 255, sets the color of the **filtergraph~** display markers in RGB format. The default is 0 0 0 (black).

rgb4 — In left inlet: The word rgb4, followed by three numbers between 0 and 255, sets the color of the rectangle that outlines the **filtergraph~** object display in RGB format. The default is 0 0 0 (black).

scale — In left inlet: The scale message, followed by a 0 or 1, sets the frequency display scale. scale 0 sets a linear frequency display (default), and scale 1 sets a log display scale.

set — In left inlet: The word set, followed by a list of five int values which correspond to **biquad~** filter coefficients, sets the **filtergraph~** object's internal values for these

coefficients but does not cause output. If **filtergraph~** is in display mode, it will display the frequency response of the filter obtained from these coefficients.

in 6th inlet: A list of three values which correspond respectively to center/cutoff frequency, gain and Q/S (resonance/slope), sets these values, recalculates the new filter coefficients but does not cause output. In display mode this message has no effect.

spectrum — In left inlet: The word spectrum, followed by a 0 or 1, specifies whether to display the amplitude or phase, with respect to frequency. spectrum 0 sets a frequency amplitude display (default), and spectrum 1 sets a phase frequency display scale.

(loadbang) — In left inlet: **filtergraph~** responds to a loadbang message sent to it when a patcher is loaded (See the autoout message).

(Get Info...) — Opens the **filtergraph~** object's Inspector window.

(preset) — You can save and restore the settings of **filtergraph~** using a **preset** object.

Inspector

The behavior of a **filtergraph~** object is displayed and can be edited using its Inspector. If you have enabled the floating inspector by choosing **Show Floating Inspector** from the Windows menu, selecting any **filtergraph~** object displays the **filtergraph~** Inspector in the floating window. Selecting an object and choosing **Get Info…** from the Object menu also displays the Inspector.

The **filtergraph~** Inspector lets you set the following attributes:

The *Filter Type* pop-up menu sets the kind of filter type to be displayed by the **filtergraph~** object. The filter types are *Display* (the default), *Lowpass*, *Highpass*, *Bandpass*, *Bandstop*, *Peak/Notch*, *Low Shelf*, or *High Shelf*. If you are operating in Display mode, a checkbox is used to enable the two red circles when in display mode. In any of the filter modes, you can used the *Gain-Enabled* checkbox to enable gain scaling in the display.

The *Constraint* options let you set maximum and minimum ranges for mousing and input constraints for both the frequency and amplitude axes.

The *Frequency Display* options allow you to set minimum and maximum frequency ranges to display (the default values are 0 and 22050 Hz.), let you choose linear (the default) or logarithmic display scales, and choose a full spectrum display.

The *Amplitude Display* options allow you to set the *Minimum* and *Maximum Display Range* (the default values 0.0625 and 16 correspond to +-12dB). Radio buttons allow you to also select *Amplitude Response* (the default) or **Phase Response**

(whose range is always -π to π). If you have selected amplitude response, you may also choose between linear (default) or logarithmic (i.e. deciBel) display of values.

Checking the *Output Coefficients on Load* checkbox will cause the **filtergraph~** objects to output its **biquad~** filter coefficients when the object receives messages which configure the filter.

Checking the *Show Numerical Display* option makes **filtergraph~** display the numerical values for frequency, gain and Q values while clicking and dragging the bandwidth rectangle with the mouse.

Checking the *Show deciBel Values* option sets the **filtergraph~** object to display the numerical values for gain change in deciBels represented by the small ticks at the right-hand side of the object's display.

The *Color* pop-up menu lets you use a swatch color picker or RGB values to specify the colors used for display by the **filtergraph~** object. These are described above, in the Input section.

The *Revert* button undoes all changes you've made to an object's settings since you opened the Inspector. You can also revert to the state of an object before you opened the Inspector window by choosing **Undo Inspector Changes** from the Edit menu while the Inspector is open.

Arguments

None.

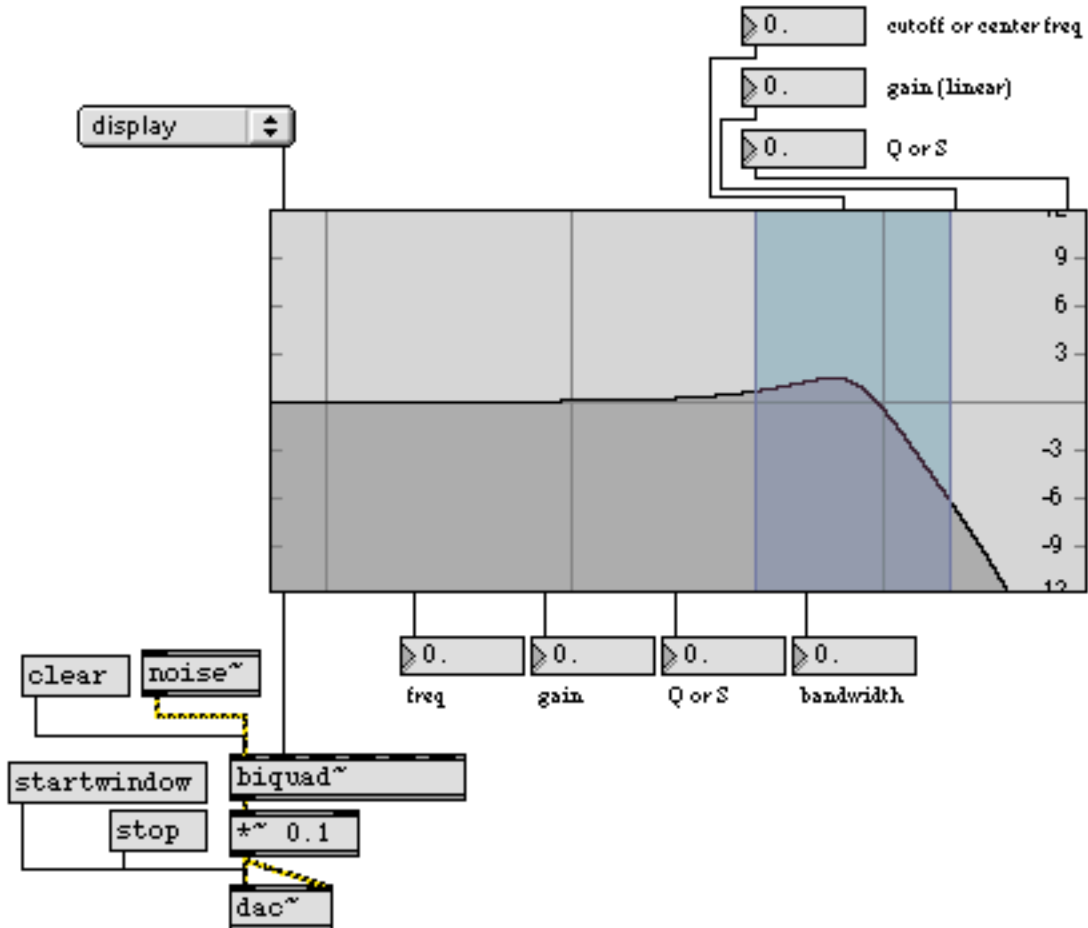
Output

list — Out leftmost outlet: a list of 5 floating-point filter coefficients for the **biquad~** object. Coefficients output in response to mouse clicks and changes in the coefficient or filter parameter inlets. They are also output when the audio is turned on, and optionally when the patch is loaded if the automatic output option is turned on (see autoout message, above).

Out rightmost (sixth) outlet: a list of 2 floating-point values (amplitude, phase) output in response to the query message (see above).

float — Out middle four outlets: Frequency, Gain (linear), Resonance (Q) and Bandwidth output in response to clicks on the **filtergraph~** object

Examples



The **filtergraph~** *object greatly simplifies working with the* **biquad~** *object*

See Also

allpass~	Allpass filter
biquad~	Two-pole, two-zero filter
delay~	Delay line specified in samples
lores~	Resonant lowpass filter
reson~	Resonant bandpass filter
teeth~	Comb filter with feedforward and feedback delay control

frameaccum~

Compute "running phase" of successive phase deviation frames

Input

signal The input to be accumulated.

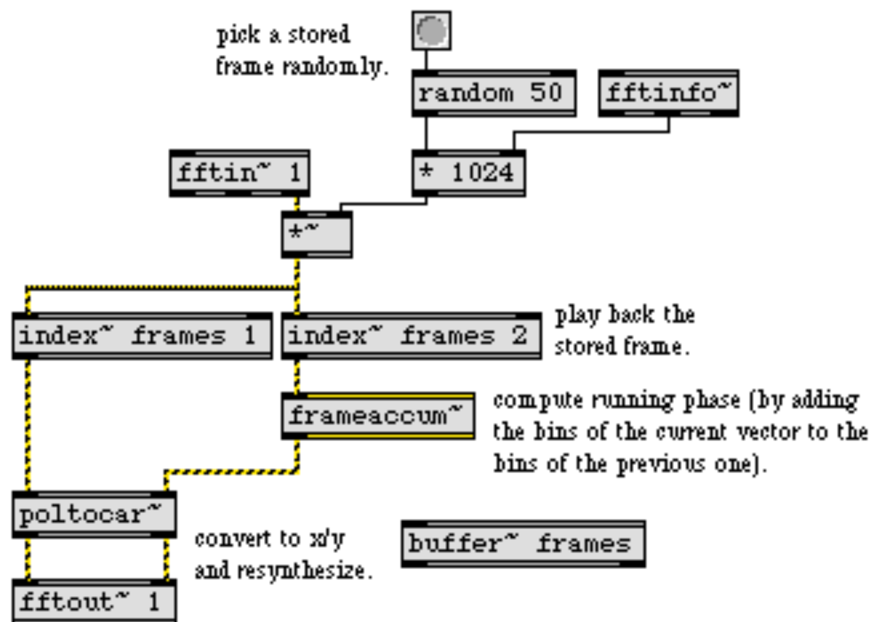
Arguments

None.

Output

signal The **frameaccum~** object computes a running phase by keeping a sum of the values in each position of its incoming signal vectors. In other words, for each signal vector, the first sample of its output will be the sum of all of the first samples in each signal vector it has received, the second sample of its output will be the sum of all the second samples in each signal vector, and so on. When used inside a **pfft~** object, it can keep a running phase of the FFT because the FFT size is equal to the signal vector size.

Examples



frameaccum~ *computes the running phase between frames of spectral data*

See Also

framedelta~	Compute phase deviation between successive FFT frames
Tutorial 26	Frequency Domain Signal Processing with **pfft~**

framedelta~

Compute phase deviation between successive FFT frames

Input

signal The input on which the deviation will be computed.

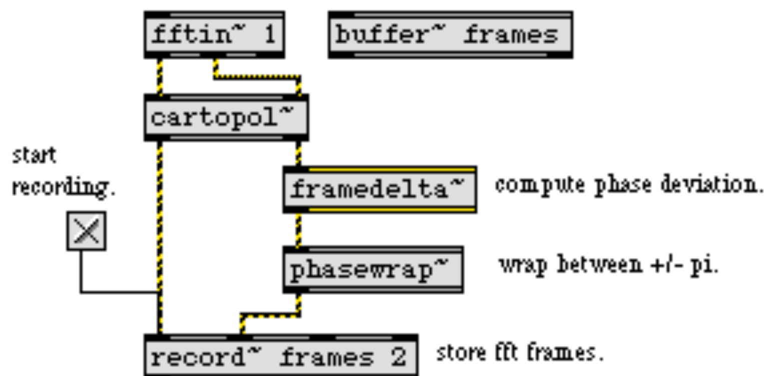
Arguments

None.

Output

signal The **framedelta~** object computes a running phase deviation by subtracting values in each position of its previously received signal vector from the current signal vector. In other words, for each signal vector, the first sample of its output will be the first sample in the current signal vector minus the first sample in the previous signal vector, the second sample of its output will be the second sample in the current signal vector minus the second sample in the previous signal vector, and so on. When used inside a **pfft~** object, it keeps a running phase deviation of the FFT because the FFT size is equal to the signal vector size.

Examples



framedelta~ *computes the difference between successive frames of FFT data*

See Also

frameaccum~	Compute "running phase" of successive phase deviation frames
Tutorial 26	Frequency Domain Signal Processing with **pfft~**

Input

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

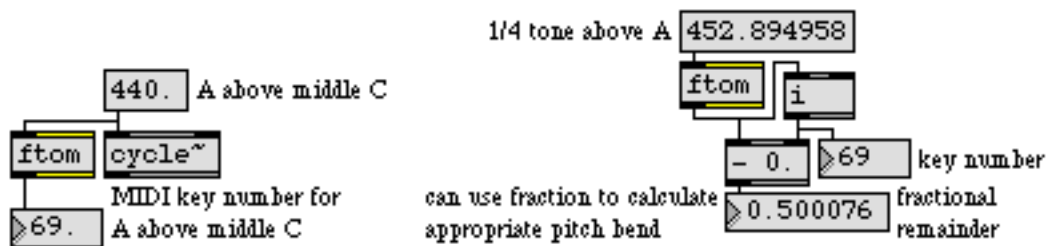
Arguments

float — Optional. If a float value is present, the **ftom** object outputs floating-point values. By default, it outputs int values.

Output

int or float — The MIDI note value that corresponds to the input frequency. When an input frequency falls *between* two equal tempered pitches, the value is rounded to the nearest int when **ftom** is used in its default int mode. When **ftom** is used in the optional float mode, the fractional part of the float is included. The fractional part could 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

expr	Evaluate a mathematical expression
mtof	Convert a MIDI note number to frequency

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 Macintosh or Control-clicking on Windows 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 the optional output mode is enabled, the output would be 0 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.

brgb The word brgb, followed by three numbers between 0 and 255, sets the RGB values for the background color of the **function** object. The default value is light gray (brgb 204 204 204).

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.

clickadd The message clickadd 0 prevents the user from creating new breakpoints by dragging them with the mouse. clickadd 1 allows the user to create new breakpoints. The default behavior allows the user to create 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.

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.

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.

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. An optional symbol argument can be used to specify a **receive** objects as a destination.

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.

frgb — The word frgb, followed by three numbers between 0 and 255, sets the RGB values for the breakpoints displayed by the **function** object. The default value is grey (frgb 82 82 82).

legend — The word legend, followed by a 1 or 0, enables (1) or disables (0) the numerical display (legend) of the **function** object, displayed when a point is highlighted or updated. The default value is on (legend 1).

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.)

listdump — Outputs a single list which contains all X and Y values for each of the breakpoints out the **function** object's middle-right outlet. An optional symbol argument can be used to specify a **receive** objects as a destination.

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.

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.

outputmode — The word outputmode, followed by a 1 or 0, enables (1) or disables (0) the optional output mode. If on, when the function object receives a bang, it sends its values in single list in which the first Y value is followed by a 0, followed by any additional Y values and associated times. When off, the object outputs its values as described above in the description of the bang message. The optional output mode is off by default.

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.

rgb2 — The word rgb2, followed by three numbers between 0 and 255, sets the RGB values for the line segments displayed by the **function** object. The default value is dark gray (rgb2 85 85 85).

rgb3 — The word rgb3, followed by three numbers between 0 and 255, sets the RGB values for the sustain points displayed by the **function** object. The default value is white (rgb3 255 255 255).

rgb4 — The word rgb4, followed by three numbers between 0 and 255, sets the RGB values for the numerical display (legend) of the **function** object when it is highlighted or being updated. The default value is black (rgb4 0 0 0).

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.

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.

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. Command-clicking on Macintosh or Control-clicking on Windows also toggle the sustain property of a point.

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

Inspector

The behavior of a **function** object is displayed and can be edited using its Inspector. If you have enabled the floating inspector by choosing **Show Floating Inspector** from the Windows menu, selecting any **function** object displays the **function** Inspector in the floating window. Selecting an object and choosing **Get Info…** from the Object menu also displays the Inspector.

The **function** Inspector lets you set the following attributes:

The *Graph Range* options allow you to set the minimum (default 0.) and maximum (default 1.0) display ranges for Y values.

The *Graph Domain* option sets the maximum (default 1000.) maximum displayed X value in milliseconds. The minimum value is always 0.

Checking the *Enable Dragging Points* checkbox will allow the user to create new breakpoints by clicking with the mouse. The default enables behavior allows the user to create new breakpoints. The current setting is saved with the object when its patcher is saved. Checking the *Enable Dragging Points* checkbox will allow the user to move existing breakpoints by dragging them with the mouse. The default behavior allows the user to drag breakpoints. Checking the *Show Legend* checkbox enables the numerical display (legend) of the **function** object, displayed when a point is highlighted or updated. The default value is enabled. Checking the *Output Only List for line~* checkbox enables the optional output mode of the **function** object. When enabled, the function object will output a single list which consists of all breakpoints when the object receives a bang. The optional output mode is off by default.

The *Color* pop-up menu lets you use a swatch color picker or RGB values to specify the colors used for display by the **function** object. *Points* sets the color for the breakpoints displayed (default 82 82 82), and *Background* sets the color for the message area in which the hint appears (default 204 204 204). *Line Segments* sets the color for the line segments that connect the breakpoints (default 85 85 85). *Sustain Points* sets the color used to display sustain points (default 255 255 255). *Legend Text* sets the color for the legend text (default 0 0 0).

The *Revert* button undoes all changes you've made to an object's settings since you opened the Inspector. You can also revert to the state of an object before you opened the Inspector window by choosing **Undo Inspector Changes** from the Edit menu while the Inspector is open.

Arguments

None.

function

Graphical breakpoint function editor

Output

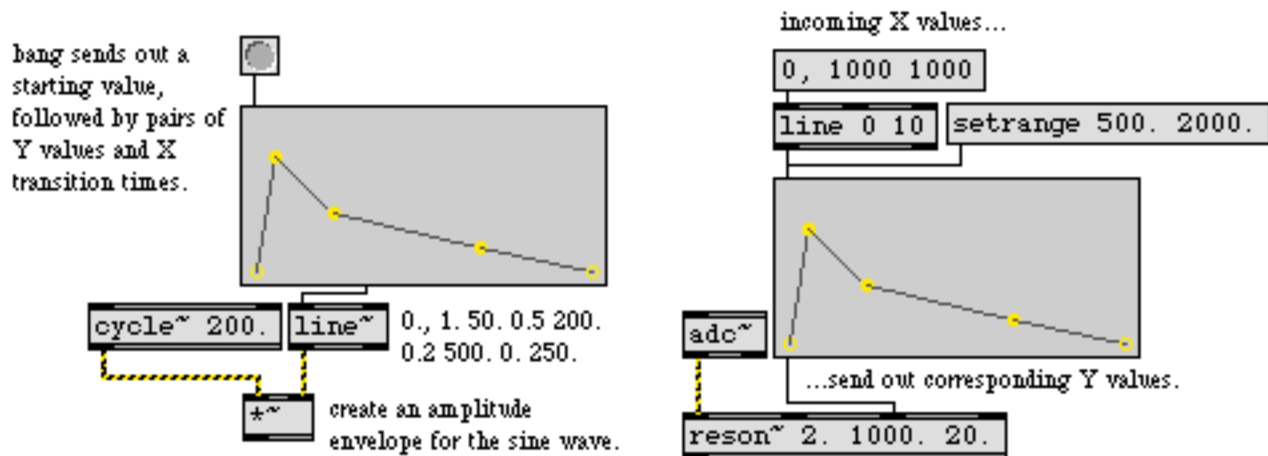
float Out left outlet: The interpolated Y value is sent out in response to a float or int X value received in the inlet; or a stored Y value is sent out in response to an nth message.

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** object'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.

Examples



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

See Also

line~	Linear ramp generator
Tutorial 7	Synthesis: Additive synthesis

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.

color — In left inlet: 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.

inc — The word inc, followed by a float, sets the increment value used to calculate the output scale factor based on the input value. The default value is 1.071519. See the Inspector section for an explanation of the calculation.

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

scale — The word scale, followed by a float, sets the base output value used to calculate the output scale factor based on the input value. The default value is 7.94231. See the Inspector section for an explanation of the calculation.

set — In left inlet: The word set, followed by a number, 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.

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 — In left inlet: 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. The default value is 158.

Inspector

The behavior of a **gain~** object is displayed and can be edited using its Inspector. If you have enabled the floating inspector by choosing **Show Floating Inspector** from the Windows menu, selecting any **gain~** object displays the **gain~** Inspector

in the floating window. Selecting an object and choosing **Get Info…** from the Object menu also displays the Inspector.

The **gain~** Inspector lets you set four parameters—the *Range*, the second is the *Base Value*, and 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 value (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.

You can also set the *Interpolation Time* by entering a value which will set the interpolation time for the gain~ object The default value is 10 milliseconds.

The *Revert* button undoes all changes you've made to an object's settings since you opened the Inspector. You can also revert to the state of an object before you opened the Inspector window by choosing **Undo Inspector Changes** from the Edit menu while the Inspector is open.

Arguments

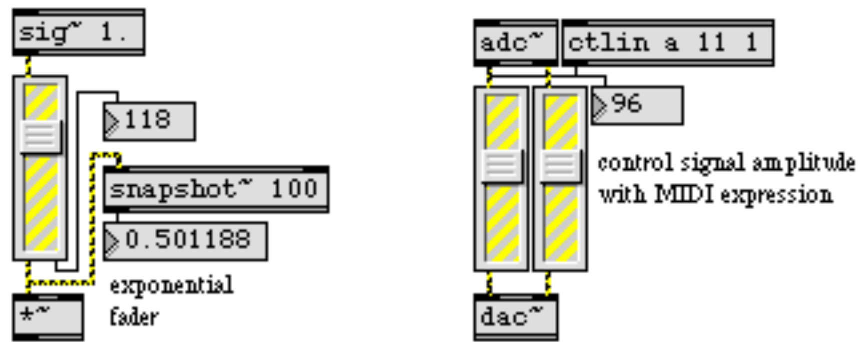
None.

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~**

gate~

Route a signal to one of several outlets

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 int or float messages 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.

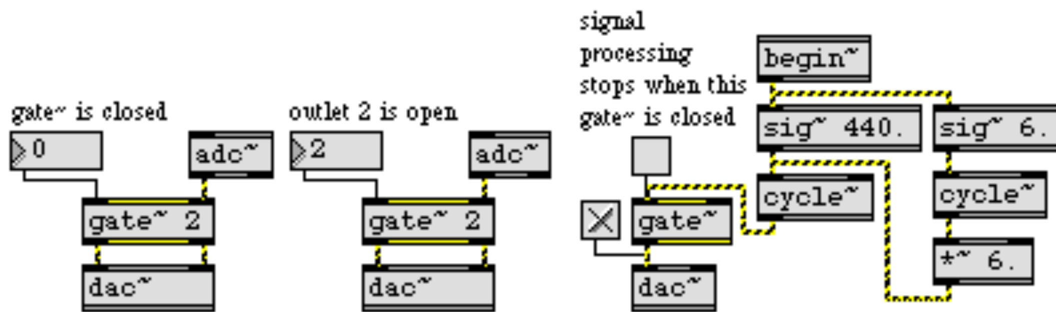
Arguments

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. The **groove~** object uses the **buffer~** sampling rate to determine playback speed.

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.

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

loopinterp — The word loopinterp, followed by 1, enables interpolation about start and end points for a loop. loop 0 turns off loop interpolation. By default, loop interpolation is off.

reset — Clears the start and end loop points.

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

setloop — The word setloop, followed by two numbers, sets the start and end loop points in milliseconds.

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.

(mouse) — Double-clicking on a **groove~** object opens the sample display window of the **buffer~** object associated with the **groove~** object.

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.

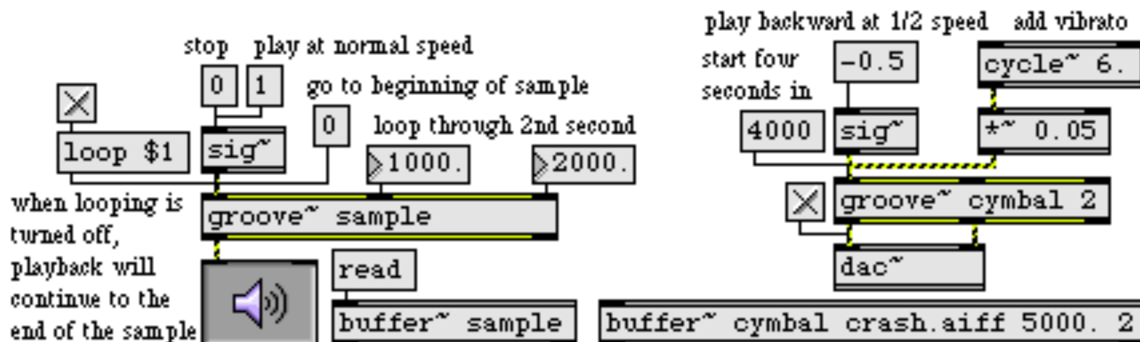
Output

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

2d.wave~	Two-dimensional wavetable
buffer~	Store audio samples
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.

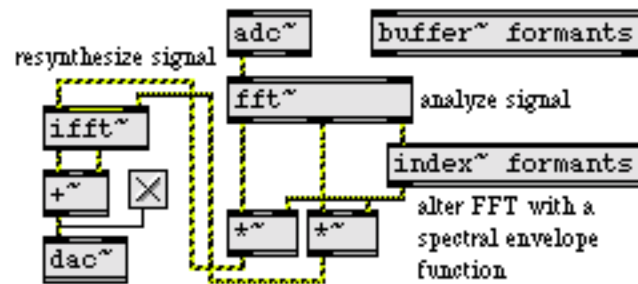
Output

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

cartopol	Cartesian to Polar coordinate conversion
cartopol~	Signal Cartesian to Polar coordinate conversion
fft~	Fast Fourier transform
fftin~	Input for a patcher loaded by **pfft~**
fftinfo~	Report information about a patcher loaded by **pfft~**
fftout~	Output for a patcher loaded by **pfft~**
frameaccum~	Compute "running phase" of successive phase deviation frames
framedelta~	Compute phase deviation between successive FFT frames
pfft~	Spectral processing manager for patchers
poltocar	Polar to Cartesian coordinate conversion
poltocar~	Signal Polar to Cartesian coordinate conversion
vectral~	Vector-based envelope follower
Tutorial 25	Analysis: Using the FFT

Input

None.

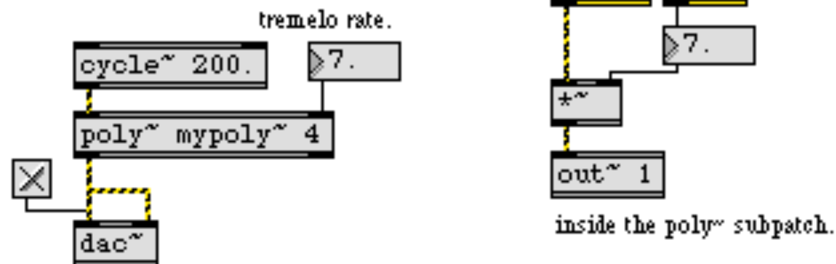
Arguments

int — Obligatory. Each **in** object is identified by a unique index number which specifies which message inlet in a **poly~** or **pfft~** object it corresponds to. The first outlet is 1.

Output

message — Each **in** object in a patcher loaded by the **poly~** or **pfft~** objects appears as an inlet at the top of the object. Messages received at the first message inlet of the **poly~** or **pfft~** object are sent to the first **in** object (i.e., the **in 1** object) in the loaded patcher, and so on. The number of total inlets in a **poly~** or **pfft~** object is determined by whether there are a greater number of **in~** or **in** objects in the loaded patch (e.g., if your loaded **poly~** patcher has three **in~** objects and only two **in** objects, the **poly~** object will have three inlets, two of which will accept both signals and anything else, and a third inlet which only takes signal input).

Examples



Message inlets of the **poly~** *object correspond to the* **in** *objects inside the loaded patcher*

See Also

in~	Signal input for a patcher loaded by **poly~**
out	Message output for a patcher loaded by **poly~**
out~	Signal output for a patcher loaded by **poly~**
pfft~	Spectral processing manager for patchers
poly~	Polyphony/DSP manager for patchers
thispoly~	Control **poly~** voice allocation and muting
Tutorial 21	MIDI control: Using the **poly~** object

Input

None.

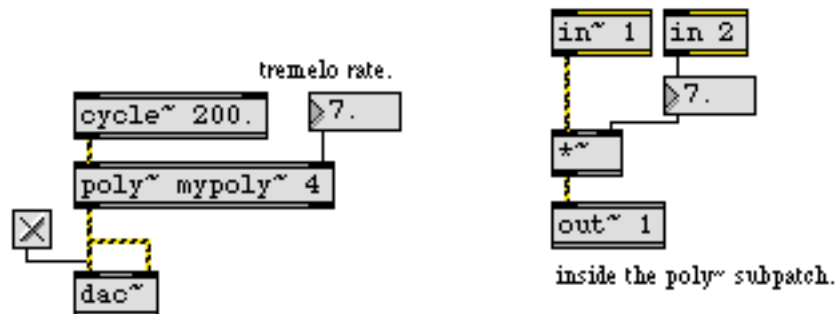
Arguments

int — Obligatory. Each **in~** object is identified by a unique index number which specifies which signal inlet in a **poly~** object it corresponds to. The first inlet is 1.

Output

signal — Each **in~** object in a patcher loaded by the **poly~** object appears as an inlet at the top of the **poly~** object. Signals received at the first message inlet of the **poly~** object are sent to the first **in~** object (i.e., the **in~** 1 object) in the loaded patcher, and so on. The number of total inlets in a **poly~** object is determined by whether there are a greater number of **in~** or **in** objects in the loaded patch (e.g., if your loaded patcher has three **in~** objects and only two **in** objects, the **poly~** object will have three inlets, two of which will accept both signals and anything else, and a third inlet which only takes signal input).

Examples



Signal inlets of the **poly~** *object correspond to the* **in** *objects inside the loaded patcher*

See Also

in	Message input for a patcher loaded by **poly~**
out	Message output for a patcher loaded by **poly~**
out~	Signal output for a patcher loaded by **poly~**
poly~	Polyphony/DSP manager for patchers
thispoly~	Control **poly~** voice allocation and muting
Tutorial 21	MIDI control: Using the **poly~** object

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

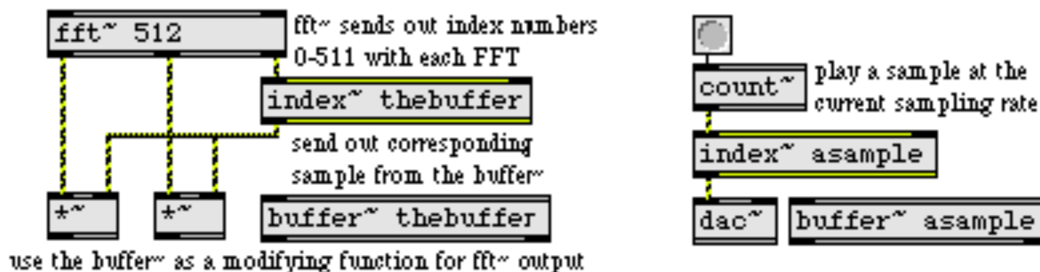
symbol — Obligatory. Names the **buffer~** object whose sample memory is used by **index~** for playback.

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

2d.wave~	Two-dimensional wavetable
cycle~	Table lookup oscillator
buffer~	Store audio samples
buffir~	Buffer-based FIR filter
fft~	Fast Fourier transform
wave~	Variable-size wavetable
Tutorial 13	Sampling: Recording and playback

info~

Report information about a sample

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 audio 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.

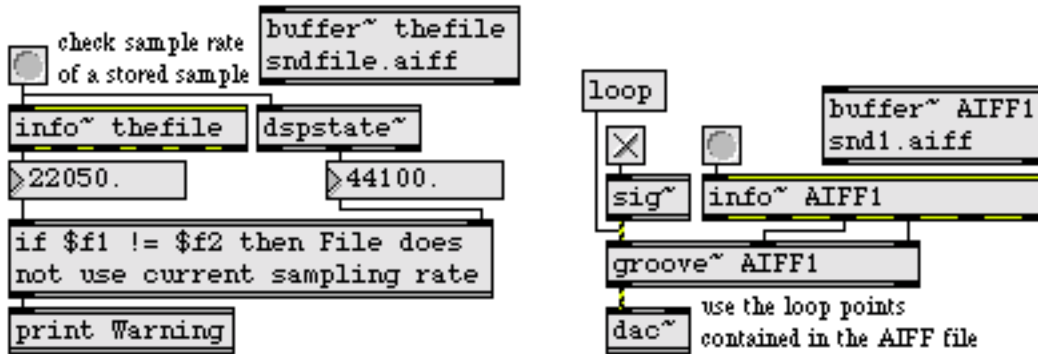
Out 7th outlet: Total time of the associated **buffer~** object, in milliseconds.

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 audio samples
mstosamps~	Convert milliseconds to samples
sfinfo~	Report audio file information
Tutorial 14	Sampling: Playback with loops
Tutorial 20	MIDI control: Sampler

Input

signal or float — In left inlet: Sets the frequency of the oscillator whose index is currently referenced to the current floating-point value of the signal. The default value is 0.

In 2nd inlet: Sets the magnitude (amplitude) of the oscillator whose index is currently referenced.

In 3rd inlet: If frame sync is enabled using the framesync 1 message, a signal in the range 0-1.0 sets the phase of the oscillator currently being referenced.

In 4th inlet: Sets the index of the oscillator currently being referenced.

float — In 3rd inlet: A float in the range 0-1.0 sets the phase of the oscillator currently being referenced.

clear — The word clear sets the frequency of all oscillators to zero and zeros all amplitudes.

copybuf — In left inlet: The word copybuf, followed by a symbol that specifies a buffer, copies 4096 samples from the buffer into the **ioscbank~** object's internal wavetable. An optional second integer argument specifies the position in the buffer at which samples are loaded (offset).

framesync — The word framesync, followed by a non-zero number, enables frame synchronous operation. When frame synchronous operation is enabled, a given index's values will only change or begin their interpolated ramps to the next value when the index input signal is 0 (or once per *n* sample frame). Otherwise, a given index's values will change or begin their interpolated ramps to the next value when the index input signal is equal to that index. The default is off.

freqsmooth — The word freqsmooth, followed by a number, sets the number of samples across which frequency smoothing is done. The default is 1 (no smoothing).

magsmooth — The word magsmooth, followed by a number, sets the number of samples across which magnitude (amplitude) smoothing is done on an oscillator. The default is 0 (no amplitude smoothing).

set — The word set, followed by pairs of floating-point values, sets the frequency and amplitude of an oscillator in the oscillator bank. A list of *n* pairs will set the first *n* oscillators in the **ioscbank~** object and zero the amplitude of all others.

silence — The word silence zeros the amplitude of all the oscillators.

size — The word size, followed by a number, sets the number of oscillators. The default is 64.

Arguments

int Optional. The number of oscillators. The default is 1.

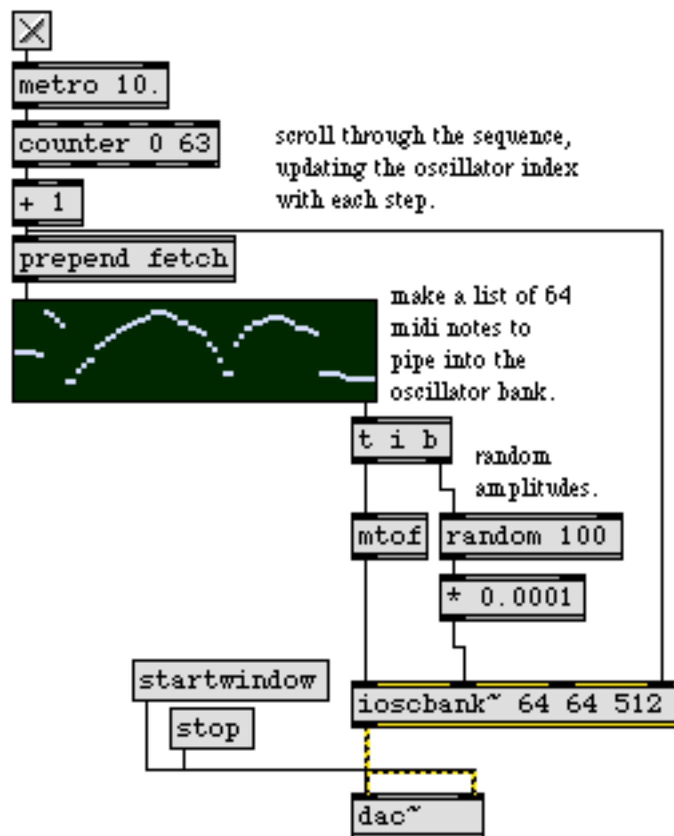
int Optional. The number of samples across which frequency smoothing is done.

int Optional. The number of samples across which amplitude smoothing is done.

Output

signal A waveform consisting of the sum of the specified frequencies and amplitudes.

Examples



ioscbank~ *lets you sound multiple interpolated oscillators with one object*

See Also

oscbank~	Non-interpolating oscillator bank

kink~

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.

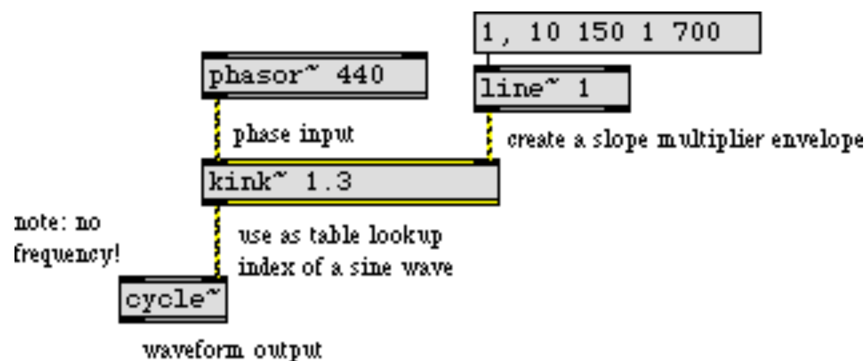
Arguments

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~	Sawtooth wave generator
triangle~	Triangle/ramp wavetable

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

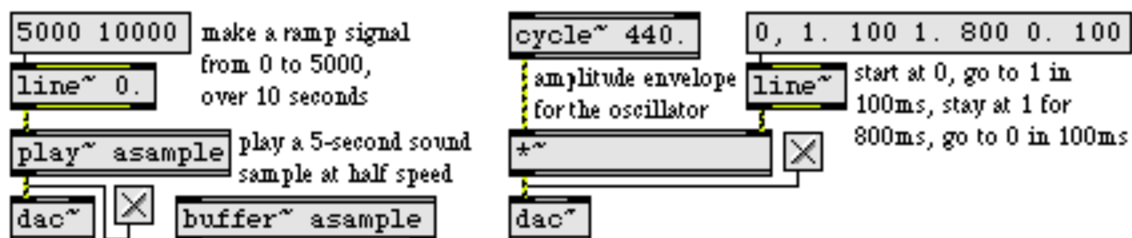
float or int — Optional. Sets an initial value for the signal output. The default value is 0.

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

click~	Create an impulse
curve~	Exponential ramp generator
Tutorial 2	Fundamentals: Adjustable oscillator

linedrive

Scale numbers exponentially for use with **line~**

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 (approximately 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 time interval.

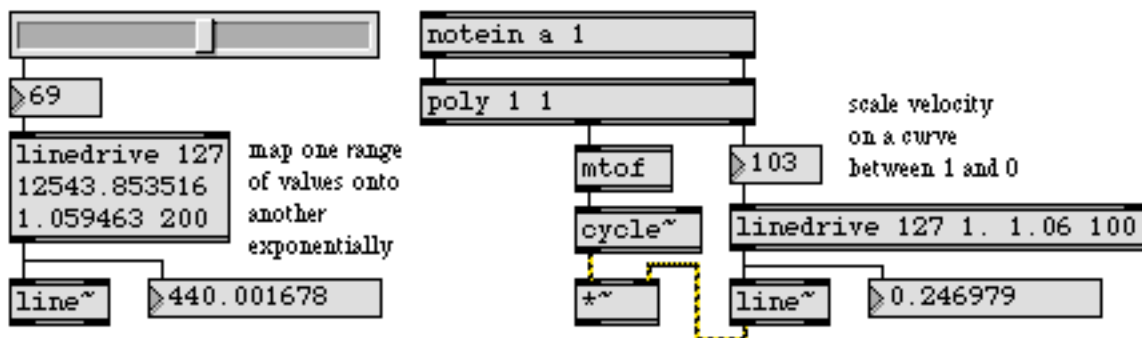
Arguments

int or float — Obligatory. The first argument is the maximum input value, followed by the maximum output value. The third argument specifies the nature of the scaling curve. The third argument must be greater than 1. The larger the value, the more steeply exponential the curve is. An appropriate value for this argument is 1.06. 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



Use **linedrive** *for exponential value scaling*

See Also

expr	Evaluate a mathematical expression
line~	Linear ramp generator

log~

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.

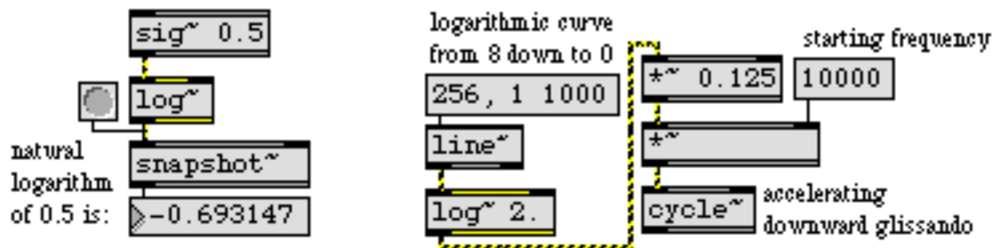
Arguments

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

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

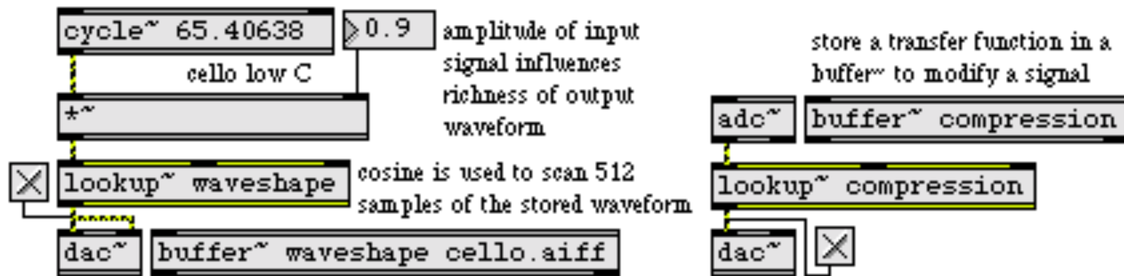
symbol — Obligatory. Names the **buffer~** object whose sample memory is used by **lookup~** for table lookup.

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 audio samples
peek~	Read and write sample values
Tutorial 12	Synthesis: Waveshaping

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.

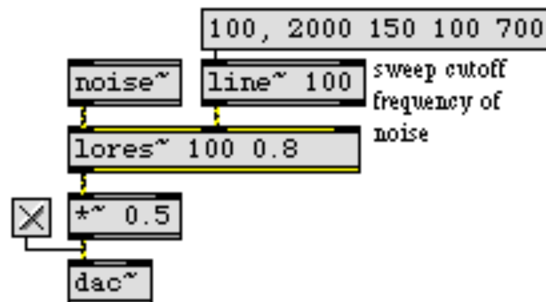
Output

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
buffir~	Buffer-based FIR filter
comb~	Comb filter
filtergraph~	Graphical filter editor
onepole~	Single-pole lowpass filter
reson~	Resonant bandpass filter
teeth~	Comb filter with feedforward and feedback delay control

The **matrix~** object is an array of signal connectors and mixers (adders). It can have any number of inlets and outlets. Signals entering at each inlet can be routed to one or more of the outlets, with a variable amount of gain. If an outlet is connected to more than one inlet, its output signal is the sum of the signals from the inlets.

The **matrix~** object has two modes of operation: "*binary*" and *non-binary.* In *binary* mode, connections act like simple switches, and always have unity gain. This mode is faster, but audible clicks will occur if you're listening to the outputs of this object when connections are made and broken. In *non-binary* mode, connections are gain stages, i.e. they can scale the signal by some amount other than zero and one. They also provide an adjustable ramping time over which the gain values are changed. This allows signals to be switched without creating audible clicks.

Input

signal — In any inlet: Signals present at an inlets are sent to the outlets to which they are connected, scaled by the gain values of the connections.

list — In left inlet: A list of three ints may be used to connect inlets and outlets when the **matrix~** object is in binary mode. The first int specifies the inlet, the second int specifies the outlet, and a third int is used to specify connection or disconnection. If the third int is nonzero value, the inlet of the first int will be connected to the outlet specified by the second int. A zero value for the third int in the list disconnects the inlet-outlet pair.

If the **matrix~** object is operating in non-binary mode, A list of two ints followed by a float sets the gain of the connection between inlet i and outlet o to the value specified by the float.

Note: To specify the gain of individual connections, you must use three-element list messages rather than the connect message. Connections formed with the connect message always have a gain specified by the third argument initially given to the **matrix~** object. However, subsequent list messages can alter the gain of connections formed with the connect message. The addition of an optional fourth element to the list message can be used to specify a ramp time, in milliseconds, for the individual connection (e.g., 1 2 .8 500 would connect the first inlet to the second outlet and specify a gain of .8 and a ramp time of .5 seconds).

print — In left inlet: The word print causes the current state of all **matrix~** object connections to be printed in the Max window in the form of a list for each connection. The list consists of two numbers which specify the inlet and outlet, followed by a float which specifies the gain for the connection.

dump — In left inlet: The word dump causes the current state of all **matrix~** object connections to be sent out the rightmost outlet of the object in the form of a list for each connection. The list consists of two numbers which specify the inlet and outlet, followed by a float which specifies the gain for the connection.

clear — In left inlet: The word clear removes all connections.

connect — In left inlet: The word connect, followed by one or more pairs of ints, will connect any inlet specified by the first int from the outlet specified by the second int. Multiple disconnections may be made by adding additional int pairs to the message. Inlets and outlets are numbered from left to right, starting at zero. For example, the message connect 1 0 1 1 would connect the second inlet from the left to the left-most outlet and the second outlet from the left.

disconnect — In left inlet: The word disconnect, followed by one or more pairs of ints, will disconnect any inlet specified by the first int from the outlet specified by the second int. Multiple disconnections may be made by adding additional int pairs to the message.

ramp — In left inlet: The word ramp, followed by a number, sets the time in milliseconds use to change gain values when the **matrix~** object is in non-binary mode. The default millisecond value is 10.

Arguments

int — Obligatory. The first argument specifies the number of inlets.

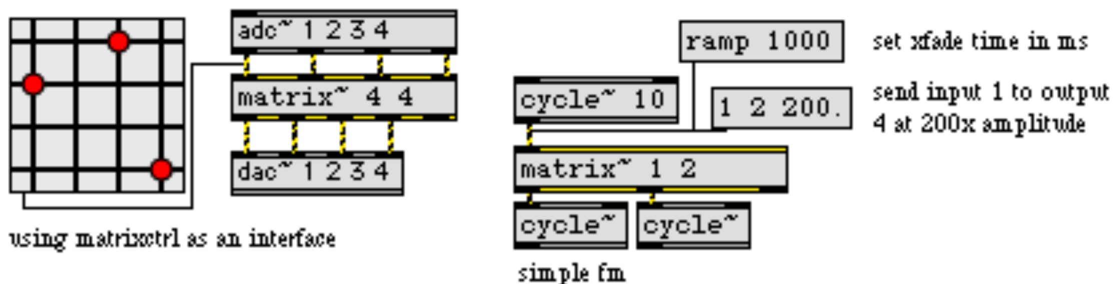
int — Obligatory. The second argument specifies the number of outlets.

float — Optional. If a float value is provided as a third argument, **matrix~** operates in its non-binary mode. The argument sets the gain value that will be used when connections are created.

Output

signal — The output signals for each outlet are the sum of their connected inputs, scaled by the gain values of the connections.

Examples



Multichannel audio routing

See Also

gate~	Route a signal to one of several outlets
matrixctrl	Matrix switch control
receive~	Receive signals without patch cords
selector~	Assign one of several inputs to an outlet
send~	Transmit signals without patch cords

maximum~

Compare two signals, output the maximum

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. The greater of the two values is sent out the outlet.

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.

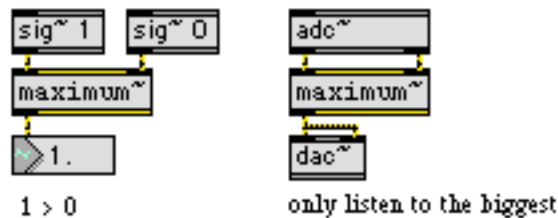
Arguments

float or int — Optional. Sets an initial comparison value for the signal coming into the left inlet. If a signal is connected to the right inlet, the argument is ignored.

Output

signal — The greater of the two signal values received in the left and right inlets is sent out.

Examples



Find the maximum of two signals

See Also

<=~	*Is less than or equal to,* comparison of two signals
>~	*Is greater than,* comparison of two signals
>=~	*Is greater than or equal to,* comparison of two signals
==~	*Is equal to,* comparison of two signals
!=~	*Not equal to,* comparison of two signals
minimum~	Compare two signals, output the minimum

Input

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

brgb — The word brgb, followed by three numbers between 0 and 255, sets the RGB values for the background color of the **meter~** object. The default value is set by brgb 104 104 104.

dbperled — The word dbperled, followed by a number between 1 and 12, sets the amount of signal level in deciBels represented by each LED. By default each LED represents a 3dB change in volume from its neighboring LEDs.

frgb — The word frgb, followed by three numbers between 0 and 255, sets the RGB values for the LED color for the lowest "cold" range of the **meter~** object. The default value is set by frgb 0 168 0.

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.

numhot — The word numhot, followed by a number between 0 and 20, sets the total number "hot" warning LEDs displayed on the **meter~** object (corresponding to the color set by the rgb2 message). The default number is 3.

numleds — The word numleds, followed by a number between 10 and 20, sets the total number of LEDs displayed on the **meter~** object. The default number of LEDs is 12.

numtepid — The word numtepid, followed by a number between 0 and 20, sets the total number "tepid" mid-range LEDs displayed on the **meter~** object (corresponding to the color set by the rgb5 message). The default number is 3.

numwarm — The word numwarm, followed by a number between 0 and 20, sets the total number "warm" lower-mid-range LEDs displayed on the **meter~** object (corresponding to the color set by the rgb4 message). The default number is 3.

rgb2 — The word rgb2, followed by three numbers between 0 and 255, sets the RGB values for the LED color for the upper "hot" range of the **meter~** object. The default value is set by rgb2 255 153 0.

rgb3 — The word rgb3, followed by three numbers between 0 and 255, sets the RGB values for the LED color for the "over" indicator of the **meter~** object. The default value is set by rgb3 255 0 0.

rgb4 — The word rgb4, followed by three numbers between 0 and 255, sets the RGB values for the LED color for upper-middle "warm" range of the **meter~** object. The default value is set by rgb4 153 186 0.

rgb5 The word rgb5, followed by three numbers between 0 and 255, sets the RGB values for the LED color for the lower-middle “tepid” range of the **meter~** object. The default value is set by rgb5 217 217 0.

(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.

Inspector

The behavior of a **meter~** object is displayed and can be edited using its Inspector. If you have enabled the floating inspector by choosing **Show Floating Inspector** from the Windows menu, selecting any **meter~** object displays the **meter~** Inspector in the floating window. Selecting an object and choosing **Get Info…** from the Object menu also displays the Inspector.

The **meter~** Inspector lets you set the update time interval, in milliseconds, of the display by typing a number into the *Interval* box. The default interval is 100 ms.

The various Appearance options in the **meter~** Inspector let you set the *Total Number of LEDs* displayed on the meter~ object. The meter~ object can have a minimum of 10 and a maximum of 20 LEDs; there are 12 LEDs by default. You can also set how much volume each LED represents by changing the *dB Per LED* value. By default each LED represents a 3dB change in volume. The *Number of Hot LEDs*, *Number of Hot LEDs*, and *Number of Hot LEDs* boxes let you set the number of LEDS in each of the volume ranges, corresponding to the *Warning (Hot)*, *Tepid* and *Warm* colors, respectively (see Color, below). By default there are three LEDs in each of these color regions - all remaining LEDs use the color of the *Foreground (Cold)* color region.

The *Color* pull-down menu lets you use a swatch color picker or RGB values to specify the colors used for display by the **meter~** object. *Background* sets the **meter~** object’s background color. The default background color is 104 104 104. The remaining menu choices set the colors of the various ranges of LEDs, from lowest to highest. *Foreground (Cold)* sets the color for the lowest range of LEDs on the **meter~** object. The default value is 0 168 0. *Tepid* sets the LED color for the lower-midrange range group of LEDs. The default value is 153 186 0. *Warm* sets the LED color for the upper-mid range of LEDs. The default value is 217 217 0. *Warning (Hot)* sets the LED color for the upper range of the **meter~** object. The default value is 255 153 0. *Overload* sets the LED color for the “over” indicator of the **meter~** object. The default value is 255 0 0.

The *Revert* button undoes all changes you've made to an object's settings since you opened the Inspector. You can also revert to the state of an object before you opened the Inspector window by choosing **Undo Inspector Changes** from the Edit menu while the Inspector is open.

meter~

Visual peak level indicator

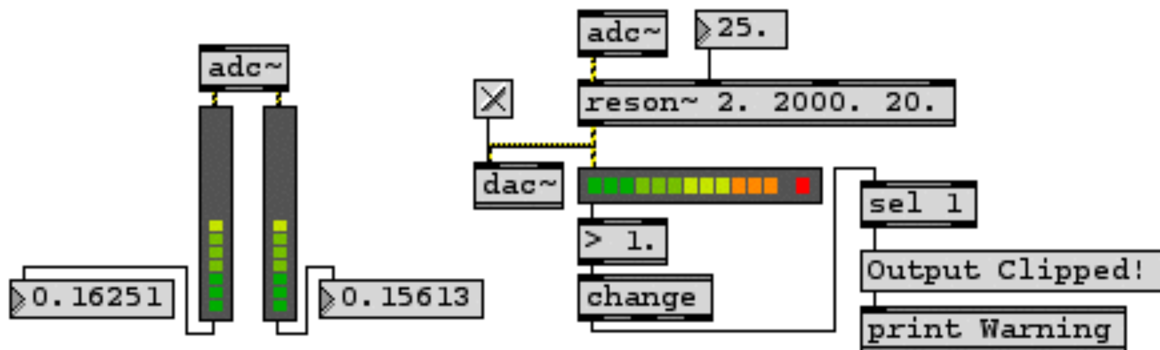
Arguments

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

average~	Multi-mode signal average
scope~	Signal oscilloscope
Tutorial 23	Analysis: Viewing signal data

minimum~

Compare two signals, output the minimum

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. The lesser of the two values is sent out the outlet.

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.

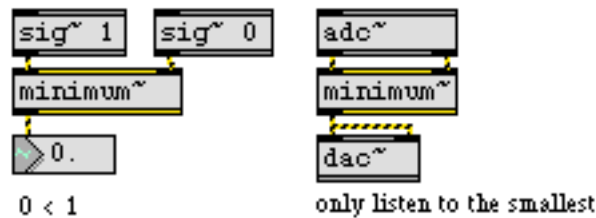
Arguments

float or int — Optional. Sets an initial comparison value for the signal coming into the left inlet. If a signal is connected to the right inlet, the argument is ignored.

Output

signal — The lesser of the two signal values received in the left and right inlets is sent out.

Examples



Find the minimum of two signals

See Also

<=~	*Is less than or equal to,* comparison of two signals
>~	*Is greater than,* comparison of two signals
>=~	*Is greater than or equal to,* comparison of two signals
==~	*Is equal to,* comparison of two signals
!=~	*Not equal to,* comparison of two signals
maximum~	Compare two signals, output the maximum

minmax~

Compute the minimum and maximum values of a signal

Input

signal — Signal to be evaluated for maximum and minimum values.

bang — Sends floating-point values corresponding to the minimum and maximum signal values out the 3rd and 4th outputs.

reset — Resets the current minimum and maximum values to the default (0).

Arguments

None.

Output

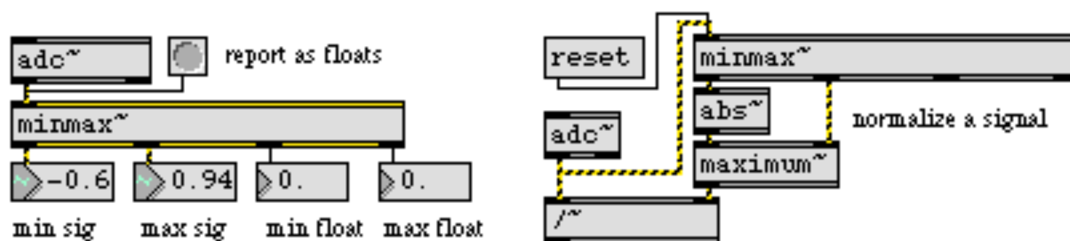
signal — Out 1st outlet: Signal value which corresponds to the minimum signal value received since startup or the last reset message.

Out 2nd outlet: Signal value which corresponds to the maximum signal value received since startup or the last reset message.

float — Out 3rd outlet: When **minmax~** receives a bang message, a floating-point value which corresponds to the minimum signal value received since startup or the last reset message is output.

Out 4th outlet: When **minmax~** receives a bang message, a floating-point value which corresponds to the maximum signal value received since startup or the last reset message is output.

Examples



Find the hi/low peaks of a signal

See Also

meter~	Visual peak level indicator
peakamp~	See the maximum amplitude of a signal
snapshot~	Convert signal values to numbers

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

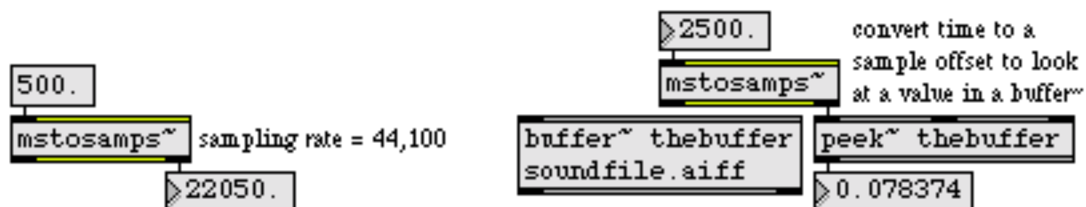
None.

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

mtof

Convert a MIDI note number to frequency

Input

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

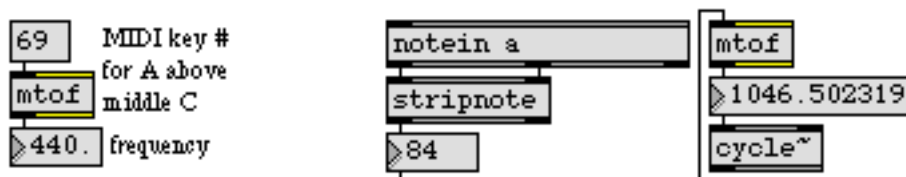
Arguments

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

expr — Evaluate a mathematical expression
ftom — Convert frequency to a MIDI note number

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.

list Sending the list 1 1 to the **mute~** object will mute any subpatchers of the **patcher** object to which the message is sent. Similarly, sending the list 0 1 to the **mute~** object will unmute any subpatchers of the **patcher** object.

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 patchers inside **bpatcher** objects.

Examples

You can mute all processing in any patcher or other subpatch

See Also

begin~	Define a switchable part of a signal network
pass~	Eliminate noise in a muted subpatcher
Tutorial 5	Fundamentals: Turning signals on & off

noise~

Input

None.

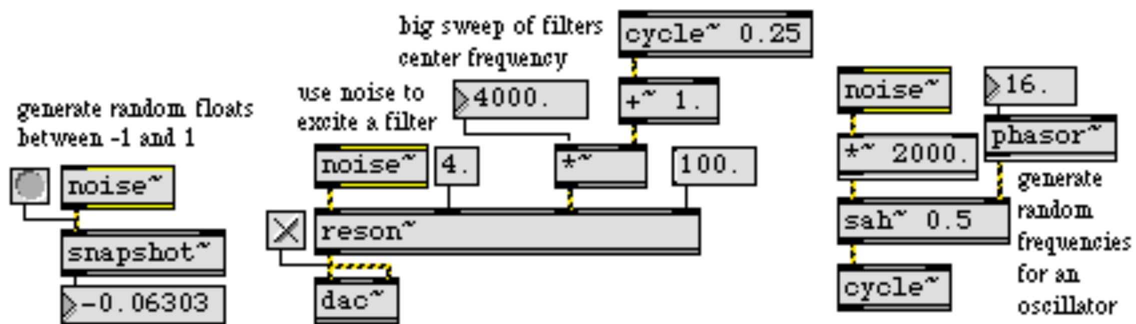
Arguments

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
pink~	Pink noise generator
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.

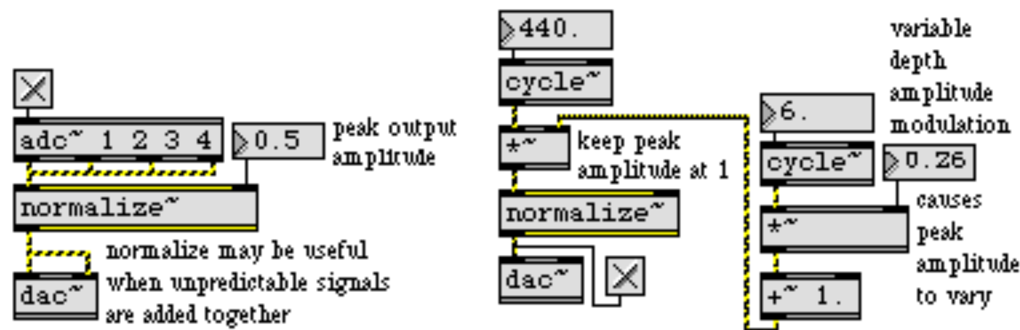
Arguments

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~

Signal monitor and constant generator

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~**.

brgb — The word brgb, followed by three numbers between 0 and 255, sets the RGB values for the background color of the **number~ box**. The default value is white (brgb 255 255 255).

frgb — The word frgb, followed by three numbers between 0 and 255, sets the RGB values for the number values displayed by the **number~** box. The default value is black (frgb 0 0 0).

rgb2 The word rgb2, followed by three numbers between 0 and 255, sets the RGB values for the number values displayed by the **number~** box when it is highlighted or being updated. The default value is black (rgb2 0 0 0).

rgb3 The word rgb3, followed by three numbers between 0 and 255, sets the RGB values for the background color of the **number~** box when it is highlighted or being updated. The default value is white (rgb3 255 255 255).

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 desiƒred 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.

Inspector

The behavior of a **number~** object is displayed and can be edited using its Inspector. If you have enabled the floating inspector by choosing Show Floating Inspector… from the Windows menu, selecting any **number~** object in the patcher window opens an Inspector panel which lets you change the behavior of that object. Selecting an object and choosing **Get Info…** from the Object menu also displays the Inspector.

The **number~** Inspector lets you set the following attributes:

You can set the range for stored, displayed, typed, and passed-through values by typing values into the *Range Min.* and *Max.* boxes. If the *No Min.* and *No Max.* checkboxes are checked (the default state), the **number~** objects will have their

minimum and maximum values set to "None." Unchecking these boxes sets the minimum and maximum values to 0.

The Options section of the Inspector lets you set the display attributes of the **number~** object. Other options available in the Inspector are: *Bold* (to display in bold typeface), *Draw Triangle* (to have an arrow pointing to the number, giving it a distinctive appearance), *Output Only on Mouse-Up* (to send a number only when the mouse button is released, rather than continuously), *Can't Change* (to disallow changes with the mouse or the computer keyboard), and *Transparent* (to display only the number in the **number~** object and not the box, so that the number box resembles a **comment** object).

The *Display Style* pull-down menu lets you select the way that number values are represented. *Decimal* is the default method of displaying numbers. *Hex* shows numbers in hexadecimal, useful for MIDI-related applications. *Roland Octal* shows numbers in a format used by some hardware devices where each digit ranges from 1 to 8; 11 is 0 and 88 is 63. *Binary* shows numbers as ones and zeroes. *MIDI Note Names* shows numbers according to their MIDI pitch value, with 60 displayed as C3. *Note Names C4* is the same as *MIDI Note Names* except that 60 is displayed as C4. With all display modes, numbers must be typed in the format in which they are displayed.

Mode lets you check boxes to select *Signal Monitor* or *Signal Output* modes. Both modes are checked by default, but at least one mode must be checked.

Interval 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. The default is 250 ms.

The *Color* option lets you use a swatch color picker or RGB values used to display the **number~** box and its background in its normal and highlighted forms. *Number* sets the color for the number displayed (default 0 0 0), *Background* sets the color for the **number~** box object itself (default 221 221 221), *Highlighted Number* sets the color of the number display when the number box is selected or its values are being updated (default 0 0 0), and *Highlighted Background* sets the color of the **number~** box when it is highlighted or being updated (default 221 221 221).

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

The *Revert* button undoes all changes you've made to an object's settings since you opened the Inspector. You can also revert to the state of an object before you opened the Inspector window by choosing **Undo Inspector Changes** from the Edit menu while the Inspector is open.

Arguments

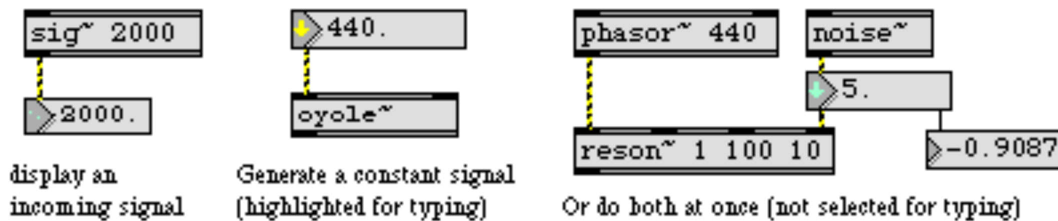
None.

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~	Linear ramp generator
sig~	Constant signal of a number
snapshot~	Convert signal values to numbers
Tutorial 23	Analysis: Viewing signal data

onepole~

Single-pole lowpass filter

The **onepole~** implements the simple filter equation

*output = previous input + cf * (input - previous input)*

where *cf* represents the cutoff frequency of the filter expressed in radians. The values for *cf* lie in the range -1.0-0. This produces a single-pole lowpass filter with a 6dB/octave attenuation, which can be useful to gently roll off harsh high end (e.g., the digital artifacts of downsampling). **onepole~** is equivalent to a **biquad~** object with the coefficients,

[a0 = 1 + cf, a1 = 0, a2 = 0, b1 = cf, b2 = 0]

If you substitute these values into the **biquad~** equation, you are left with the **onepole~** object's algorithm. However, **onepole~** will execute much faster, since **biquad~** will still compute the unused portion of its equation.

Input

signal — In left inlet: Signal to be filtered.

In right inlet: A signal can be used to set the frequency for the filter, with the same effect as a float. If a signal is connected to this inlet, its value is sampled once every signal vector.

float — In right inlet: Sets the frequency for the filter (if no signal is connected). By default, frequency is expressed in Hz, where the allowable range is from 0 to one fourth of the current sampling rate. For convenience, **onepole~** has two additional input modes that use the more conventional input range, 0 - 1 (see the linear and radians messages).

clear — In either inlet: Clears the internal state of onepole~. Since onepole~ does not have the inherent instability of other filter types, this should never be necessary.

Hz — In either inlet: Sets the frequency input mode to Hz (the default).

linear — In either inlet: Sets the frequency input mode to linear (0 - 1). Linear mode is simply a scaled version of the standard Hz mode, except that values in the 0-1 range traverses the full frequency range.

radians — In either inlet: Sets the frequency input mode to radians (0 - 1). Radians mode lets you set the center frequency (*cf*) of the equation directly—while the input has the same range (0-1), the output has a curved frequency response that is closer to the exponential pitch scale of the human ear.

Arguments

float — Optional. Sets the center frequency for the filter, as described above.

Hz Optional. Sets the frequency input mode to Hz (the default mode - hence this is the same as providing no mode argument).

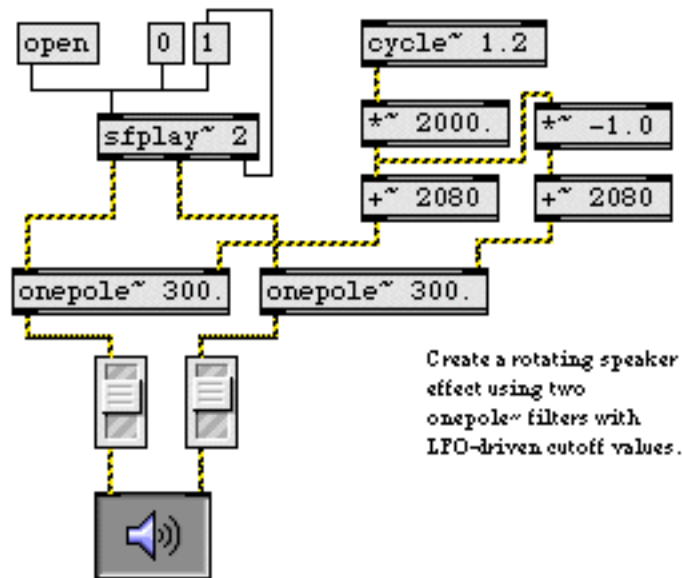
linear Optional. Sets the frequency input mode to linear (0 - 1).

radians Optional. Sets the frequency input mode to radians (0 - 1).

Output

signal The filtered signal.

Examples



onepole~ *provides efficient filtering for a simple sample player*

See Also

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

Input

signal or float	In left inlet: Sets the frequency of the oscillator whose index is currently referenced to the current floating-point value of the signal. The default value is 0.
	In 2nd inlet: Sets the magnitude (amplitude) of the oscillator whose index is currently referenced.
	In 3rd inlet: If frame sync is enabled using the framesync 1 message, a signal in the range 0-1.0 sets the phase of the oscillator currently being referenced.
	In 4th inlet: Sets the index of the oscillator currently being referenced.
float	In 3rd inlet: A float in the range 0-1.0 sets the phase of the oscillator currently being referenced.
clear	The word clear sets the frequency of all oscillators to zero and zeros all amplitudes.
copybuf	In left inlet: The word copybuf, followed by a symbol that specifies a buffer, copies samples from the buffer into the **oscbank~** object's internal wavetable. The number of samples is set using the tabpoints message. An optional second integer argument specifies the position in the buffer at which samples are loaded (offset).
framesync	The word framesync, followed by a non-zero number, enables frame synchronous operation. When frame synchronous operation is enabled, a given index's values will only change or begin their interpolated ramps to the next value when the index input signal is 0 (or once per *n* sample frame). Otherwise, a given index's values will change or begin their interpolated ramps to the next value when the index input signal is equal to that index. The default is off.
freqsmooth	The word freqsmooth, followed by an int, sets the number of samples across which frequency smoothing is done. The default is 1 (no smoothing).
magsmooth	The word magsmooth, followed by an int, sets the number of samples across which magnitude (amplitude) smoothing is done on a oscillator. The default is 0 (no amplitude smoothing).
set	The word set, followed by pairs of floating-point values, sets the frequency and amplitude of an oscillator in the oscillator bank. A list of *n* pairs will set the first *n* oscillators in the **oscbank~** object and zero the amplitude of all others.
silence	The word silence zeros the amplitude of all the oscillators.
size	The word size, followed by a number, sets the number of oscillators. The default is 64.
tabpoints	The word tabpoints, followed by a number, sets the number of wavetable points (samples) in the **oscbank~** object's internal wavetable. The default is 4096. The

number of wavetable points should be a power or two between 2^2 and 2^{16}. Any other value will be rounded to the nearest power of two.

Arguments

int Optional. The number of oscillators.

int Optional. The number of samples across which frequency smoothing is done.

int Optional. The number of samples across which amplitude smoothing is done.

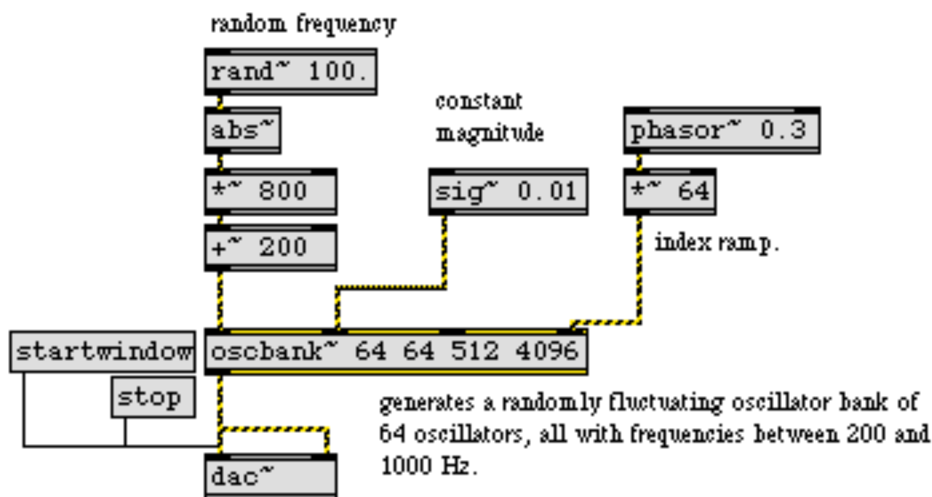
int Optional. The size, in samples, of the sinewave lookup table used by the **oscbank~** object. The default is 4096. Since **oscbank~** uses uninterpolated oscillators, you can choose to use a sinetable of larger size at the expense of CPU.

Note: There is only one wavetable for *all* oscillators in a given **oscbank~** object,

Output

signal A waveform consisting of the sum of the specified frequencies and amplitudes.

Examples



oscbank~ *creates a bank of oscillators that you can control with one object*

See Also

ioscbank~ Interpolating oscillator bank

Input

message — Each **out** object in a patcher loaded by a **poly~** or **pfft~** object appears as an outlet at the bottom of the **poly~** or **pfft~** object. Messages received in the **out** object in the loaded patcher will be sent out the corresponding outlet of the **poly~** or **pfft~** object. The message outputs are a mix of the outputs of all instances of the patcher's outputs.

Output

None.

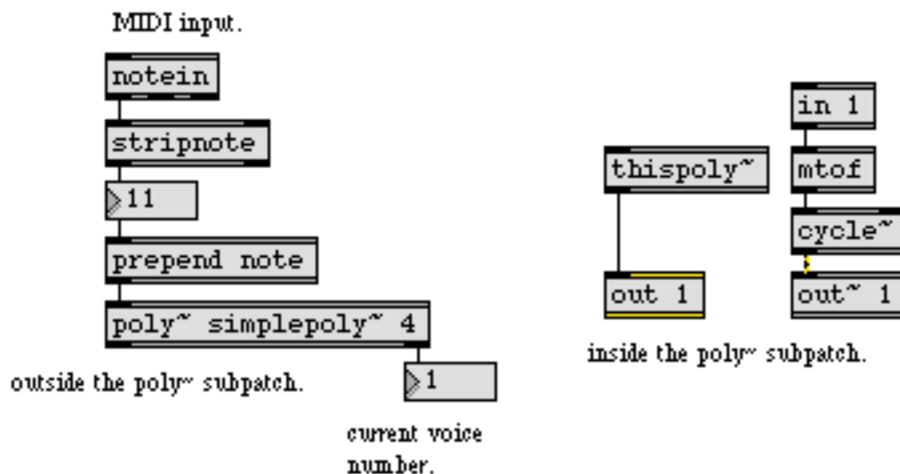
Arguments

int — Obligatory. Each **out** object is identified by a unique index number which specifies which message outlet in a **poly~** or **pfft~** object it corresponds to. The first outlet is 1.

Output

(patcher) — Any messages received by an **out** object in a loaded patcher appear at the signal outlet of the **poly~** or **pfft~** object which corresponds to the number argument of the **out** object. The signal outputs in a **poly~** or **pfft~** object are a mix of the outputs of all instances of the patcher's outputs which correspond to that number.

Examples



Message outlets of the **poly~** *object correspond to the* **out** *objects inside the loaded patcher*

See Also

in	Message input for a patcher loaded by **poly~** or **pfft**
in~	Signal input for a patcher loaded by **poly~**
out	Message output for a patcher loaded by **poly~** or **pfft~**
out~	Signal output for a patcher loaded by **poly~**
poly~	Polyphony/DSP manager for patchers
thispoly~	Control **poly~** voice allocation and muting
Tutorial 21	MIDI control: Using the **poly~** object

out~

Input

signal — Each **out~** object in a patcher loaded by the **poly~** object appear as an outlet at the bottom of the **poly~** object. Signals received by the **out~** object in the loaded patcher will be sent out the corresponding outlet of the **poly~** object. The message outputs are a mix of the outputs of all instances of the patcher's outputs.

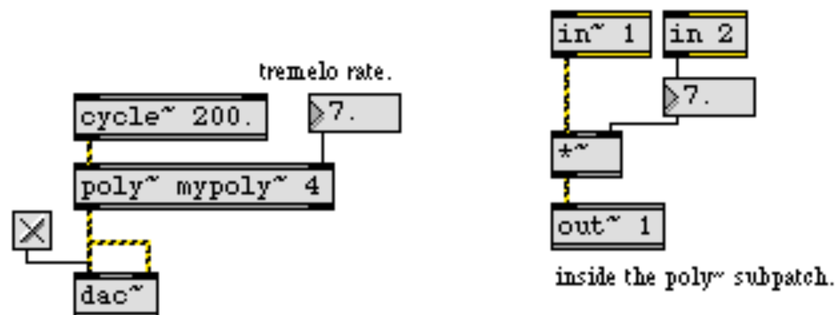
Arguments

int — Obligatory. Each **out~** object is identified by a unique index number which specifies which outlet in a **poly~** object it corresponds to. The first outlet is 1.

Output

(patcher) — Any signals received by an **out~** object in a loaded patcher appear at the signal outlet of the **poly~** object which corresponds to the number argument of the **out~** object. The signal outputs in a **poly~** object are a mix of the outputs of all instances of the patcher's outputs which correspond to that number.

Examples



Signal outlets of the **poly~** *object correspond to the* **out~** *objects inside the loaded patcher*

See Also

in	Message input for a patcher loaded by **poly~** or **pfft**
in~	Signal input for a patcher loaded by **poly~**
out	Message output for a patcher loaded by **poly~** or **pfft~**
poly~	Polyphony/DSP manager for patchers
thispoly~	Control **poly~** voice allocation and muting
Tutorial 21	MIDI control: Using the **poly~** object

overdrive~

Soft-clipping signal distortion

The **overdrive~** object uses a waveshaping function to distort audio signals. It amplifies signals, limiting the maximum value of the signal to ±1. Values outside of this range are removed using "soft clipping" somewhat like that of an overdriven tube-based circuit.

Input

signal In left inlet: the signal to be distorted.

float In right inlet: The **overdrive~** object accepts a floating-point "drive factor". The drive factor should usually be in the range 1.0-10.0. Using a factor of 1.0 creates a linear response without distortion, and higher values increase the distortion. Values less than 1, including negative values, produce very heavily distorted signals. Use with caution—this behavior was originally considered a bug until friends of the object's creator insisted that it should be considered a feature and left intact.)

int Converted to float.

Arguments

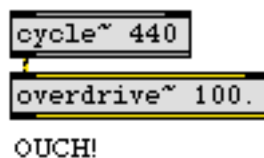
float Optional. A single number can be provided to set the drive factor. If no argument is provided, the drive factor is set to 1.0.

int Converted to float.

Output

signal The distorted signal.

Examples



Waveshape a signal similar to an overdriven amplifier

See Also

kink~	Distort a sawtooth waveform
lookup~	Transfer function lookup table

pass~

Input

signal Use a **pass~** above any **outlet** object that will handle a signal. When the audio in the subpatch is enabled, the **pass~** object will pass its input to its 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.

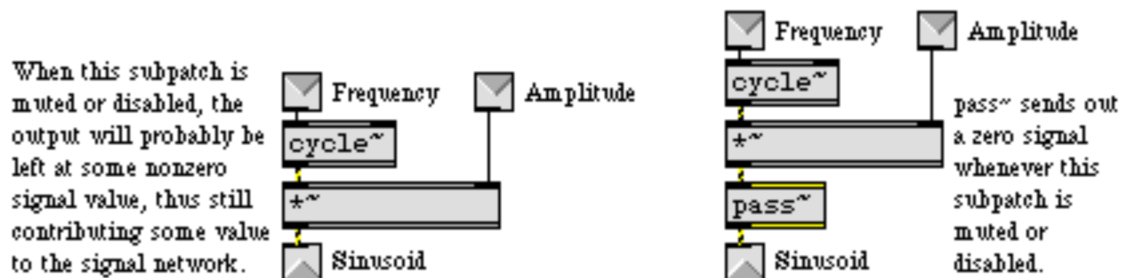
Arguments

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 signals on & off

peakamp~

See the maximum amplitude of a signal

Input

signal In left inlet: Signal to be evaluated for its peak amplitude.

bang In left inlet: Sends out a report of the greatest (absolute value) signal amplitude received since the previous report.

int In right inlet: Sets the interval in milliseconds for an internal clock that triggers the automatic output of peak amplitude 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. Time intervals shorter than the duration of one signal vector may be specified, but the peak amplitude will be checked only once per vector.

float In right inlet: Same as int.

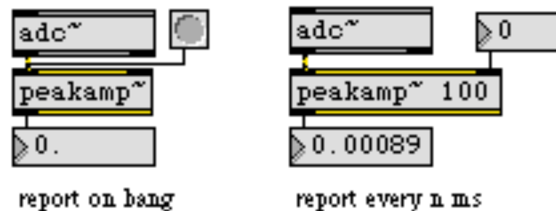
Arguments

int Optional. Sets the internal clock interval, in milliseconds. If it is 0, the internal clock is not used, so **peakamp~** will only output data when it receives a bang message. If it is non-zero, **peakamp~** will repeatedly send out the peak amplitude received in that interval of time. By default, the interval is 0.

Output

float When **peakamp~** receives a bang or its internal clock is on, the absolute value of the peak signal value from the input signal is sent out its outlet.

Examples



Report the maximum of a signal's absolute value

See Also

meter~	Visual peak level indicator
snapshot~	Convert signal values to numbers

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 the **peek~** object'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~** object'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.

clip — In left inlet: The word clip, followed by a non-zero number, enables -1.0-1.0 clipping. Clipping is enabled by default. Clipping can be disabled with the message clip 0.

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~** object'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.

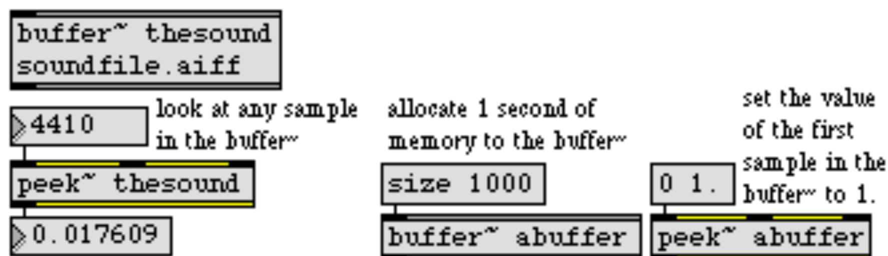
int — Optional. Following the **buffer~** name, you can type in a number to specify the channel in a multi-channel **buffer~** to use for subsequent reading or writing operations. The default is 1.

int — Optional. An optional third argument after buffer name and channel can be used to enable clipping. If the third argument is a one, then -1.0-1.0 clipping is enabled. You can also change this setting using the clip message.

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 the **peek~** object's outlet.

Examples



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

See Also

buffer~	Store audio samples
buffir~	Buffer-based FIR filter
poke~	Write sample values by index
table	Store and graphically edit an array of numbers

pfft~

Spectral processing manager for patchers

The **pfft~** object is designed to simplify spectral audio processing using the Fast Fourier Transform (FFT). In addition to performing the FFT and the Inverse Fast Fourier Transform (IFFT), **pfft~** (with the help of its companion **fftin~** and **fftout~** objects) manages the necessary signal windowing, overlapping and adding needed to create a real-time Short Term Fourier Transform (STFT) analysis/resynthesis system.

Input

signal — The number of inlets on the **pfft~** object is determined by the number of **fftin~** and/or **in** objects in the enclosed subpatch. Patchers loaded into a **pfft~** object can only be given signal inlets by **fftin~** objects within the patch. See **fftin~** and **in** for details.

bang — Patchers loaded into a **pfft~** object can only accept bang messages by **in** objects within the patch. The number of inputs is determined by the **in** objects in the enclosed subpatch. See **in** for details.

mute — The word mute, followed by a 1 or 0, will mute or unmute the **pfft~**, turning off signal processing within the enclosed subpatch.

open — The word open will open the subpatch loaded into the **pfft~** object.

wclose — Closes the enclosed subpatch if it is open.

Arguments

symbol — Obligatory. The first argument must be the name of a subpatch which will be loaded into the **pfft~** and assigned its own signal-processing chain. The signal processing chain connections for input and output are made using **fftin~** and **fftout~** objects in the subpatcher.

int — Optional. Specifies the FFT size, in samples, of the overlapped windows which are transformed to and from the spectral domain by the FFT/IFFT. The window size must be a power of 2, and defaults to 512. (Note: The size of the spectral “frames” processed by the **pfft~** object's subpatch will be half this size, as the 2nd half of the spectrum is a mirror of the first, and thus redundant.)

int — Optional. The third argument determines the overlap factor for FFT analysis and resynthesis windows. The hop size (number of samples between each successive FFT window) of Fast Fourier transforms performed is equal to the size of the Fast Fourier transform divided by the overlap factor (e.g. if the frame size is 512 and the overlap is set to 2 then the hop size is 256 samples). The value must be a power of 2 and defaults to 2.

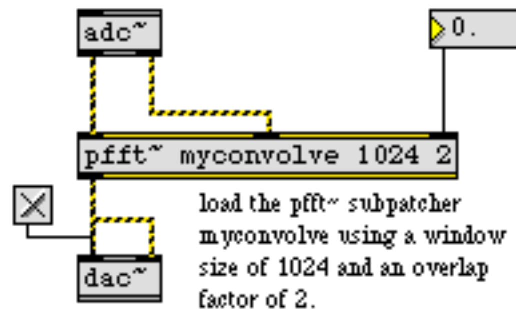
int — Optional. The fourth argument specifies the start onset in samples for the Fast Fourier transform. It must be a multiple of the current signal vector size and defaults to 0.

Output

signal: The output is the result of the FFT-based signal processing subpatch. As with the **fft~** and **ifft~** objects, **pfft~** introduces a slight delay from input to output (although it is less than half the delay than with an **fft~/ifft~** combination). The I/O delay is equal to the window size minus the hop size (e.g., for a 1024-sample FFT window with an overlap factor of 4, the hop size is equal to 256, and the overall delay from input to output is 768 samples). The number of outlets is determined by the number of **fftout~** and/or **out** objects in the loaded subpatcher. Patchers loaded into a **pfft~** object can be given outlets by **fftout~** or **out** objects within the patch. See **fftout~** and **out** for details.

message: Any messages received by an **out** object in a loaded patcher appear at the message outlet of the **pfft~** object which corresponds to the number argument of the **out** object. The message outlets of a **pfft~** object appear to the right of the rightmost signal outlet.

Examples



pfft~ *loads subpatchers specially designed for frequency domain processing*

See Also

cartopol	Cartesian to Polar coordinate conversion
cartopol~	Signal Cartesian to Polar coordinate conversion
fft~	Fast Fourier transform
fftin~	Input for a patcher loaded by **pfft~**
fftinfo~	Report information about a patcher loaded by **pfft~**
fftout~	Output for a patcher loaded by **pfft~**
frameaccum~	Compute "running phase" of successive phase deviation frames
framedelta~	Compute phase deviation between successive FFT frames
ifft~	Inverse Fast Fourier transform
in	Message input for a patcher loaded by **poly~** or **pfft~**
out	Message output for a patcher loaded by **poly~** or **pfft~**
poltocar	Polar to Cartesian coordinate conversion
poltocar~	Signal Polar to Cartesian coordinate conversion
vectral~	Vector-based envelope follower
Tutorial 25	Analysis: Using the FFT
Tutorial 26	Frequency Domain Signal Processing with **pfft~**

phaseshift~

Input

signal — The signal to be shifted in phase.

float — In middle inlet: Sets the frequency at which signals will be shifted by 180 degrees. Signals below this frequency will be shifted less; signals above will be shifted more, up to 360 degrees.

In right inlet: Sets the "Q" factor, or steepness with which the object's phase shift changes from zero to 360 degrees. Useful values for Q are generally in the range 1. to 10.

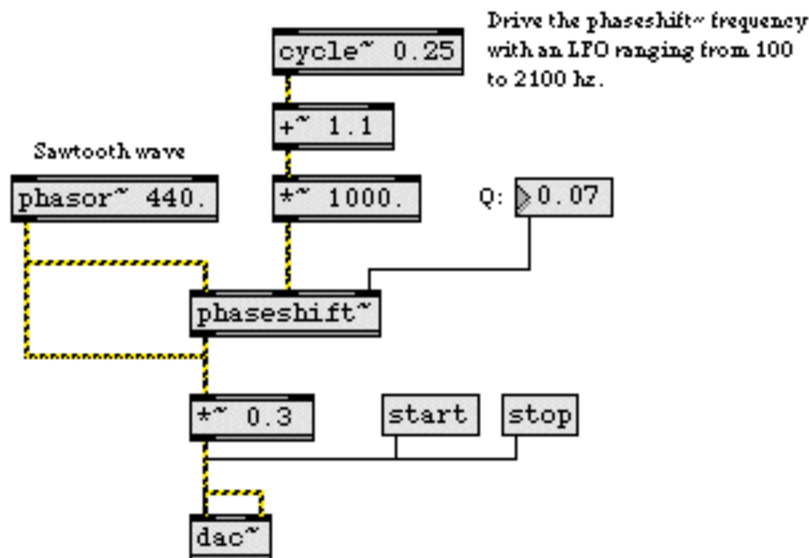
Arguments

float — Optional. If one argument is provided, it sets the **phaseshift~** object's frequency parameter. If two arguments are provided, the first sets the frequency parameter and the second sets the Q factor.

Output

signal — The input signal, its the frequency components or harmonics shifted in phase from zero to 360 degrees, dependent upon their frequency and the values of the object's frequency and Q parameters.

Examples



Simulate an analog phase shifter using **phaseshift~** *and an LFO*

See Also

allpass~	Allpass filter
comb~	Comb filter

phasewrap~

Wrap a signal between -π and π

Input

signal The signal to be wrapped. If the input signal value exceeds π (3.14159), the output signal value is "wrapped" to a range whose lower bound is -π (-3.14159)—thus, a signal of increasing value outputs sawtooth waveform with -π and π as lower and upper values.

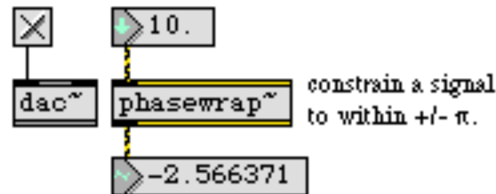
Arguments

None.

Output

signal The wrapped input signal value.

Examples



Use **phasewrap~** *to make sure that signals stay within normal radial values*

See Also

cartopol~	Signal Cartesian to Polar coordinate conversion
pfft~	Spectral-processing manager for Patchers
pong~	Variable range signal folding

phasor~

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, int and float messages are ignored.

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

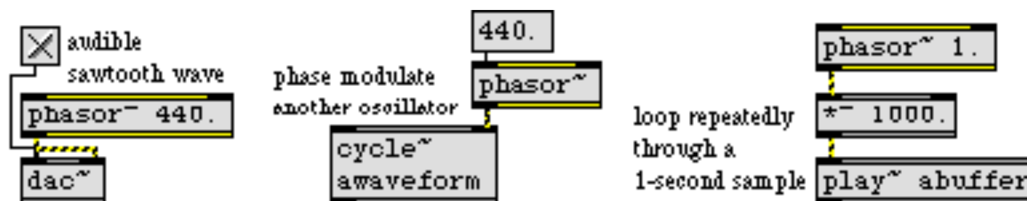
Arguments

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

2d.wave~	Two-dimensional wavetable
cycle~	Table lookup oscillator
line~	Linear ramp generator
trapezoid~	Trapezoidal wavetable
triangle~	Triangle/ramp wavetable
wave~	Variable-size wavetable
Tutorial 3	Analysis: Wavetable oscillator

pink~

Input

None.

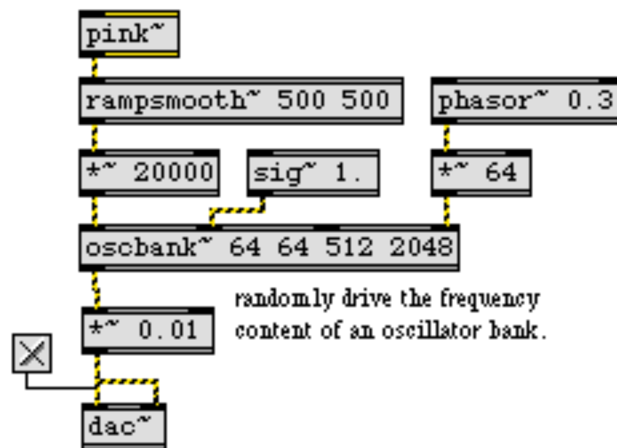
Arguments

None.

Output

signal The **pink~** object generates a signal consisting of random value in the range -1.0-1.0, with an even distribution of power per octave of frequency. Noise with this power distribution is known as “pink noise”. “White noise”, as generated by the object **noise~**, has an even distribution of power over all frequencies. Perceptually, white noise sounds bright and harsh, and pink noise sounds more even and “natural”.

Examples



pink~ *generates random numbers such that the frequency content is equal power per octave*

See Also

noise~ White noise generator

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

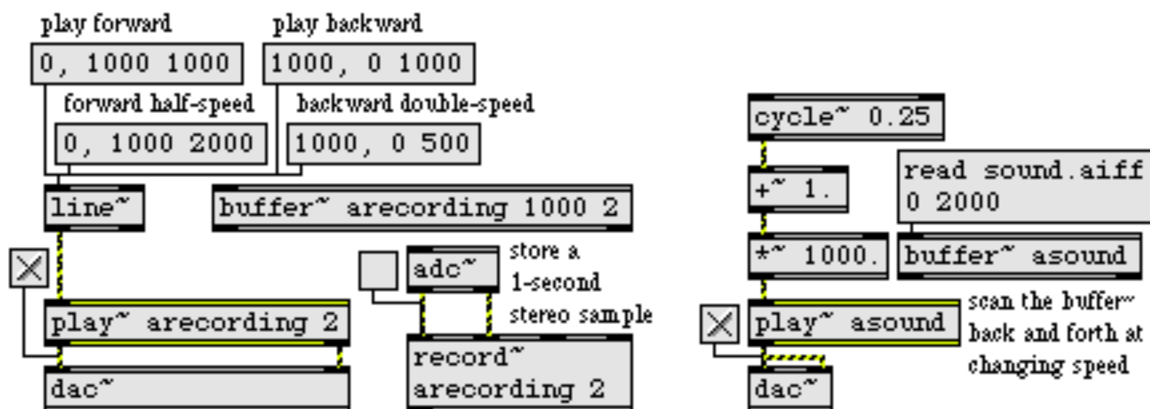
symbol — Obligatory. Names the **buffer~** object whose sample memory is used by **play~** for playback.

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

2d.wave~	Two-dimensional wavetable
buffer~	Store audio samples
buffir~	Buffer-based FIR filter
groove~	Variable-rate looping sample playback
record~	Record sound into a buffer
Tutorial 13	Sampling: Recording and playback

plugconfig

Configure the behavior of a plug-in

The **plugconfig** object lets you configure your plug-in's behavior using a script that will be familiar to users of the **env** and **menubar** objects. The script can be accessed by double-clicking on a **plugconfig** object. You should only have one **plugconfig** object per plug-in patcher; if you have more than one, the object that loads last will be used by the runtime plug-in environment. Since it's not easy to determine which object that will be, just use one.

When you double-click on **plugconfig**, you'll see a short script already in place. These are the default settings, which are in fact identical to those you'd get if your patch contained no **plugconfig** object at all.

plugconfig is pretty much a read-only object when used within the runtime plug-in environment. The environment reads the settings from the object's script and is configured accordingly. You can send the messages view and offset to the object to scroll the patcher to a new location, but most plug-ins will allow the user to do this using the View menu that appears above the plug-in interface.

Input

Use the capture and recall messages to build a set of interesting presets that are embedded within your plug-in.

capture — The word capture, followed by a program number (1-based) and optional symbol, stores the current settings of all **pp** and **plugmultiparam** objects in the patcher containing the **plugconfig** object as well as its subpatchers. The settings are stored using a setprogram message added to the **plugconfig** object's script. The parameter numbers of the **pp** and **plugmultiparam** objects determine the order of the values in the setprogram message. capture does not work within the runtime plug-in environment.

recall — The word recall, followed by a program number (1-based), sets all **pp** and **plugmultiparam** objects to the values stored within a setprogram message in the **plugconfig** object's script. The parameter numbers of the **pp** and **plugmultiparam** objects determine the values they are assigned from the contents of the setprogram message.

read — The word read, followed by an optional symbol, imports a file of effect programs saved in Cubase format and loads as many as possible into the **plugconfig** object for saving as setprogram messages. No checking is done to verify that the file contains effect programs for a plug-in with the same unique ID code as the one in the **plugconfig** object, nor is there any checking to ensure that the number of **plugconfig** parameters match. If the symbol is present, **plugconfig** looks for a file with that name. Otherwise, a standard open file dialog is displayed, allowing you select an effect program file.

view — The word view, followed by a symbol that is the name of a view defined in the **plugconfig** object's script, scrolls the patcher containing the **plugconfig** object to the coordinate offset assigned to the view.

offset — The word offset, followed by numbers for the X and Y coordinates, scrolls the patcher containing the **plugconfig** object to the specified coordinates.

Script Messages

Messages for View Configuration

A View is a particular configuration of the plug-in's edit window. **plugconfig** lets you control which views you'd like to see, and add views of the plug-in patcher at various pixel offsets that you can select with the menu. These might correspond to "pages" of controls you offer to the user.

usedefault — Arguments: none

If this message appears in a script, there is no plug-in edit window. Instead, the parameter editing features of the host environment are used. By default, usedefault is not present in a script, and the plug-in's editing window appears.

useviews — Arguments: 1/0 for showing views, as discussed below

useviews determines which plug-in edit window views are presented to the user. The views are specified in the following order: Parameters (the egg sliders), Interface (a Max patcher-based interface), Messages (a transcript of the Max window useful for plug-in development), and Plug-in Info (where you can brag about your plug-in). If the edit window is visible, the Pluggo Info view always appears.

For example, useviews 1 0 0 0 would place only the Parameters view in the plug-in edit window's View menu. The user would be unable to switch to another view.

defaultview — Arguments: name, x offset, y offset, 1/0 for initial view

defaultview renames the Interface item in the plug-in's View menu to the name argument, scrolling the patcher to the specified x and y offsets when the view is made visible. If the third argument (optional) to defaultview is non-zero, the view is made the initial view shown when the plug-in editing window is opened. This will be true anyway if there is no Parameters view (as specified by the useviews message).

addview — Arguments: name, x offset, y offset

addview adds an additional Interface view to the plug-in's View menu with a specified x and y offset. This allows you to scroll the patcher to a different location to expose a different part of the interface that might correspond to a "page" of parameter controls. If you send the view message to **plugconfig** with the name an added view as an argument, the patcher window will scroll to the view's x and y offset. This works in Max as well as in the run-time plug-in environment, allowing you to test interface configurations.

dragscroll — Arguments: allow (1), disallow (0)

This message is currently unimplemented.

meter — Arguments: 1 (meter the input, default), 2 (meter the output), 3 (off)

The meter message sets the initial mode of the level meter at the top of the plug-in edit window. There is currently no way to permanently disable the meter, but it is disabled if there isn't enough space to display it fully because you've defined an edit window that is too narrow.

Messages for Window Configuration

autosize — Arguments: none

autosize, which by default is enabled, sizes the plug-in edit window to be the height necessary to display all of the parameters, and the width of the parameter display.

setsize — Arguments: width, height

setsize sets the plug-in edit window to be a specific size in pixels. If you use the Parameters view, this size may be overridden if you've specified a window too narrow to display the egg sliders properly. Note that you should add approximately 30 pixels to the size of the patcher window in order to account for the height of the View menu and level meter panel.

windowsize — Arguments: none

windowsize sets the size of the plug-in edit window to the size of the patcher window.

Messages for Program Information

numprograms — Arguments: number of programs

numprograms sets the number of stored programs for the plug-in. Programs are collections of values (between 0 and 1) for each of the parameters you've defined using **pp** and **plugmultiparam** objects. The default number of programs is 64, the minimum is 1, and the maximum is 128. By default, all programs are set to 0 for each parameter, but you can override this with the setprogram message.

setprogram — Arguments: number, name, start index offset, list of values…

Normally, you won't be typing the setprogram message into a script yourself; you'll send capture messages to generate it automatically. You might end up editing it though—for example, to change the program's name—so it's useful to know a little about the message's format. setprogram lets you name a specific program and, optionally, set some initial values for it. Program numbers (for the first argument)

start at 1. The name is a symbol, so if there are spaces in the name, it must be contained in double quotes. The start index offset argument sets a number added to 1 that determines the starting parameter number of the parameter values listed in the message. After this argument, one or more parameter values follow. If you don't supply enough values to set all the defined parameters, the additional ones are set to 0. You don't need to set the values at all if you want them to be 0. However, when you re-open the **plugconfig** script, the additional zero values will have been added. The start index offset argument is used to handle stored programs containing more than 256 parameters. 256 is the maximum size of a Max message.

initialpgm Arguments: program number

The initialpgm message specifies the program that should be loaded when the plug-in is initially opened. The default is 0, which means no program will be loaded; instead in this case, you would use **loadbang** objects to set the initial values of plug-in parameters. This behavior, however, is not consistent with the majority of plug-ins that get set to the values in program 1 when they are loaded (since 1 is always the initial program, unless the plug-in is being restored as part of a document for the host application). Once you have a collection of settings that you like, consider storing them in the first program inside **plugconfig** and adding an initialpgm 1 message. This has the added benefit of doing away with loadbang objects used to initialize your parameters. Any other program number (up to the number of programs in the plug-in specified by the numprograms message) can also be loaded, but the current program number as shown in the host sequencer's window cannot be changed by the plug-in, so given that all host sequencers are initially set to program 1, you'll end up confusing the user if you load another program number initially.

Messages for DSP Settings

accurate Arguments: none

The accurate message tells the runtime plug-in environment to run the Max event (or control) scheduler at the same number-of-samples interval as the signal vector size. At 32 samples this is slightly less than 1 ms but running the scheduler this often can have some impact on the overall CPU intensiveness of the plug-in.

By default, accurate mode is not enabled and the scheduler runs at the same interval as the I/O vector size of the host environment, typically 512 or 1024 samples. The only thing accurate mode affects is parameter updating to a plug-in, so for example if you have a control-rate "LFO" you may want to use this mode. The use of accurate mode will also increase the frequency of parameter updating from control-rate scheduled **plugmod** processes.

sigvs default Arguments: signal vector size

This message is currently ignored by the runtime plug-in environment. 32 is currently the only possible signal vector size.

oversampling Arguments: code number

This message is currently ignored by the runtime plug-in environment.

preempt Arguments: 1/0 sets priority of control messages.

This message is currently ignored by the runtime plug-in environment.

Messages for Descriptive Information

When configuring the plug-in's informational view, you choose between using text with infotext, a picture with infopict, or not having an info view at all with noinfo.

infotext Arguments: text as separate words and numbers

infotext allows you to describe the effect and have the text appear in the Plug-in Info view. There is a limit of about 256 words. A special symbol <P> produces a carriage return. Note that all commas and semicolons in the text must be preceded by a backslash. If you do not do this, you could wipe out the rest of your script when you save it.

infopict Arguments: file name of a PICT file in the Max search path

infopict allows you to include a picture to display in the Plug-in Info view. If you use infopict, you need to include the picture (manually) to your plug-in's collective script. The runtime plug-in environment will be able to find the picture within the collective.

noinfo Arguments: none

This is the default behavior for plug-in information. If neither text nor picture has been provided as information about the effect, the Plug-in Info item does not appear in the View menu, even if you've enabled it with the useviews command above. If noinfo and either infopict or infotext appear together in a script, noinfo "loses" and the info view is displayed.

welcome Arguments: text as separate words and numbers

The text arguments to the welcome message are displayed at the bottom hint area when the user opens the plug-in editing window for the first time and looks at the Parameters view, as well as when the cursor is moved into the top part of the window when the Parameters view is being used. If the nohintarea message is present in the script, the lack of a hint area in the Parameters view will cause the welcome message not to be displayed.

nohintarea Arguments: none

If the nohintarea message appears in a script, the runtime plug-in environment does not provide additional space for a hint area at the bottom of the Parameters view. If however the number of egg sliders does not completely fill the edit window because its size was defined using windowsize or setsize, a hint area will be present.

swirl Arguments: none

The swirl message sets the hint area background to be drawn as a swirl inspired by the pluggo packaging (which was itself inspired by the publicity poster for the classic French film musical "Les Demoiselles de Rochefort"). The default appearance of the hint area is the pain, non-swirl background. To set the swirl colors, use hintfg and hintbg.

hintbg Arguments: red, green, and blue color components as 16-bit values

If you are offended by the yellow background color of the hint area, you can change it to something else. As an example, a medium gray would be specified with hintbg 40000 40000 40000, and a white background would be specified with hintbg 65535 65535 65535.

hintfg Arguments: red, green, and blue color components as 16-bit values

When using the swirl mode for the hint area, the hintfg message specifies the color of the dark part of the swirl. For best results, hintfg should be darker than hintbg.

uniqueid Arguments: id1 id2 id3 (between 0 and 255)

You'll find this message in your **plugconfig** script when you first open it. The arguments will be three randomly generated numbers between 0 and 255, something like three quarters of an IP address.

These numbers are used to build an ID code that will uniquely identify your plug-in. The code is used to identify a plug-in as a pluggo-based animal as well as to preserve **plugmod** connections between patchers.

You can either use the three randomly generated numbers or something intentional. There are about 16 million possibilities. 0 0 0 is reserved and cannot be used. 0 followed by two other numbers is reserved for use by Cycling '74 and its registered plug-in developers. You won't need to interact with this ID code, although you might want to know that part of it will be used as the basis for a floating-point "patcher code" output by the **plugmod** object. The floating-point value, however, will not in any way resemble the ID you choose.

plugconfig

Configure the behavior of a plug-in

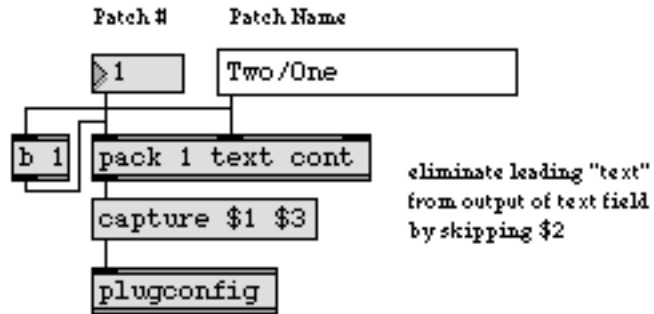
Arguments

None.

Output

None.

Examples



Send the capture message to **plugconfig** *to create presets*

See Also

plugmod	Modify plug-in parameter values
Pluggo Tutorial P2	Enhancing the plug-in interface
Pluggo Tutorial P3	A plug-in with a Max interface

plugin~

Define a plug-in's audio inputs

plugin~ and **plugout~** define the signal inputs and outputs to a plug-in. You can use them within Max as simple thru objects, feeding **plugin~** a test signal and routing the output of **plugout~** to a **dac~** object. When **plugin~** and **plugout~** are operating within the runtime environment however, they act differently. **plugin~** ignores its input and instead outputs the plug-in's signal inputs fed to it by the host mixer. **plugout~** does not output any type of signal out its outlets; instead it feeds its signal inputs to the plug-in's audio outputs to the host mixer.

Input

signal — In left and right inlets: When used in Max/MSP, the **plugin~** object echoes its input to its output. When used in the runtime plug-in environment, signals sent to its inputs are ignored, and instead the audio inputs to the plug-in are copied to the **plugin~** object's outlets.

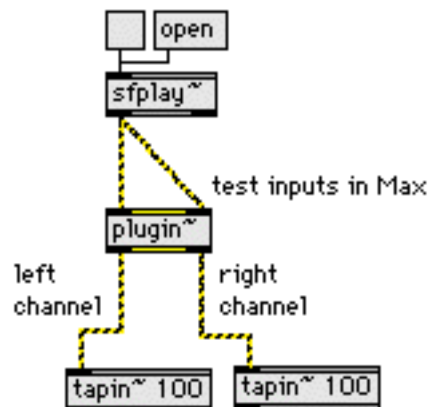
Arguments

None. **plugin~** always has two inlets and two outlets.

Output

signal — When used in Max/MSP, the signal output of the **plugin~** object is simply its signal input. When used in the runtime plug-in environment, the signal output will be the left and right channels of the audio input to the plug-in from the host. If the plug-in is inserted in a mono context, it's possible that only the left channel will contain the incoming audio signal and the right channel will be 0. The exact nature of the audio input to the plug-in is up to the host mixer.

Examples



See Also

plugout~	Define a plug-in's audio outputs

plugmidiin

Receive MIDI from a plug-in host

plugmidiin delivers any MIDI information targeted to the plug-in. It functions analogously to the Max **midiin** object, delivering raw MIDI as a sequential byte stream. You'll want to connect the **midiparse** object to its outlet. MIDI information is always delivered by **plugmidiin** at high-priority (interrupt) level. You may have more than one **plugmidiin** object in a patcher; each will output the same information.

Input

None.

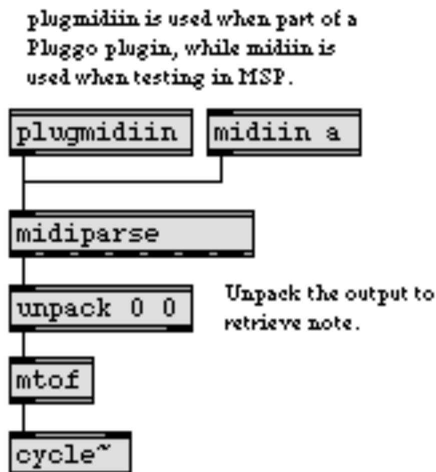
Arguments

None.

Output

int MIDI message bytes in sequential order. For instance, a note-on message on channel 1 for note number 60 with velocity of 64 would be output as 144 followed by 60 followed by 64.

Examples



MIDI message received from the host application are output by the **plugmidiin** *object*

See Also

midiparse	Interpret raw MIDI data
plugmidiout	Send MIDI to a plug-in host

plugmidiout

Send MIDI to a plug-in host

plugmidiout sends MIDI information to the host, where it is routed according to the host's current configuration. The plug-in has no control over the routing of its MIDI output. plugmidiout is analogous to **midiout**; it expects raw MIDI bytes in sequential order. You can use **midiformat** to transform numbers into MIDI messages appropriate for **plugmidiout**.

Input

int MIDI message bytes in sequential order. For instance, a note-on message on channel 1 for note number 60 with velocity of 64 would be sent to **plugmidiout** as 144 followed by 60 followed by 64.

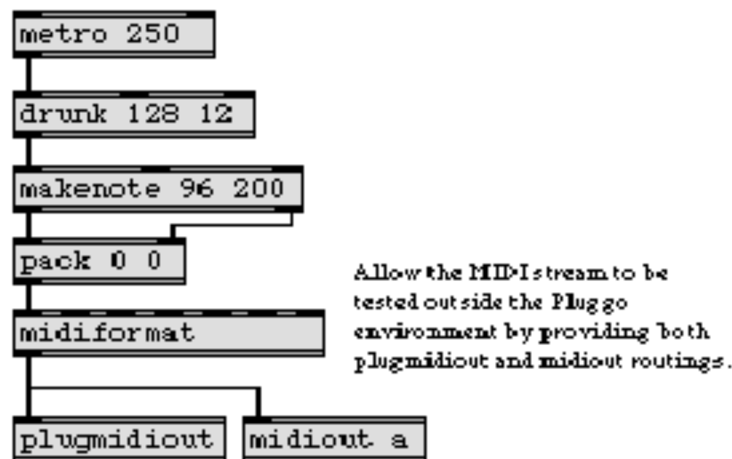
Arguments

None.

Output

None.

Examples



See Also

midiformat	Prepare data in the form of a MIDI message
plugmidiin	Receive MIDI from a plug-in host

plugmod

Modify plug-in parameter values

plugmod allows a plug-in to modify the parameter values of another plug-in. It generates a pop-up menu listing all the visible parameters of all currently loaded plug-ins. The output of this menu is fed back to the input of the object to tell it what parameter should be modified with the numeric input **plugmod** receives. Additional inlets and outlets interface with **pp** objects to save the object's connection to a particular plug-in and parameter in effect presets. This allows **plugmod** to reconnect to its target plug-in and parameter when a sequencer document is reloaded.

Input

anything — In left inlet: A plug-in name followed by a parameter index sets the parameter the **plugmod** object will modify with its numeric input. This plug-in and parameter are referred to as the object's *target*.

No Connection — In left inlet: When the word No Connection is received, the **plugmod** object breaks its connection (if any) with its current target and stops affecting the target parameter. The No Connection symbol is always the first item in the menu generated by the **plugmod** object's left outlet when plug-ins are inserted or deleted in the runtime environment.

int or float — In left inlet: The value received, which is constrained between 0 and 1, is assigned to the target plug-in and parameter.

In 2nd inlet: The value received is added to the base value of the parameter before **plugmod** began to modify it.

In 3rd inlet: The value received is multiplied by the base value of the parameter before **plugmod** began to modify it.

float — In 4th inlet: The value is interpreted as a code to assign a new plug-in as a target. The outlet of a **pp** object is normally connected to this inlet.

In right inlet: The value is interpreted as a code to assign a new parameter as a target. The outlet of a **pp** object is normally connected to this inlet.

Arguments

None.

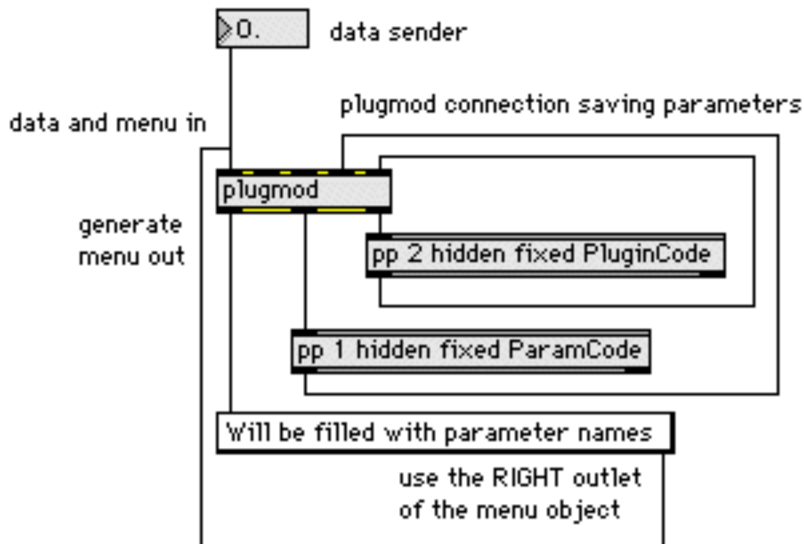
Output

anything — Out left outlet: Output from this outlet of the **plugmod** object occurs when a new plug-in is either inserted or deleted. The messages update an attached menu object with a new list of plug-ins and parameters that are potential targets for this object to modify.

float — Out 2nd outlet: The current plug-in code is output when the object's target changes via a message from the attached pop-up menu object sent to the object's left inlet, or when a new plug-in code is received in the 4th inlet.

Out right outlet: The current parameter code is output when the object's target changes via a message from the attached pop-up menu object sent to the object's left inlet, or when a new parameter code is received in the right inlet.

Examples



See Also

menu	Pop-up menu, to display and send commands
Pluggo Tutorial P5	A modulator plug-in

plugmorph

Generate parameter values from programs

plugmorph allows a plug-in to modify the parameter values of another plug-in by creating a weighted average of two or more of its effect programs. Such an average is often known as a "morph" since it can often (but not always) create a continuous perceptual space between one effect program and another. **plugmorph** generates a pop-up menu listing all currently loaded plug-ins. The output of this menu is fed back to the input of the object, allowing the user to specify which plug-in should be modified according to the input **plugmorph** receives. An additional inlet and outlet interface with a **pp** object saves the object's connection to a particular plug-in. This allows **plugmorph** to reconnect to its target plug-in when a sequencer document is reloaded.

Input

anything — In left inlet: A plug-in name sets what the **plugmorph** object will modify with its input. This plug-in is referred to as the object's *target*.

No Connection — In left inlet: When the word No Connection is received, the **plugmorph** object breaks its connection (if any) with its current target and will no longer change a plug-in's parameters. The No Connection symbol is always the first item in the menu generated by the **plugmorph** object's left outlet when plug-ins are inserted or deleted in the runtime environment.

list — In left inlet: Causes **plugmorph** to calculate new values for the connected plug-in's parameters. The format of the list is an effect program number followed by a weighting fraction. A maximum of 128 program numbers can be specified. If the fractions do not add up to 1, they are normalized to do so. As an example, the list 1 0.5 2 0.5 would set the target plug-in's parameters to values that were a simple average of effect programs 1 and 2. A list of 1 0.6 2 0.6 3 0.6 4 0.6 would perform a weighted averaging of the first four effect programs where the parameter values of each program were represented equally. In other words, each programs's parameter value contributes 25% to the morphed value. If the target plug-in's current effect program is among those being morphed, an attempt is made not to store the parameter values so the user can perform more than one morph. The generated parameter values can be stored later using the store message to **plugmorph**. However, some **multislider**-based plug-ins defer parameter changes in such a way that this storage prevention mechanism doesn't work, requiring that the user set the current effect program to a number that isn't involved in the morph.

morphfixed — In left inlet: The word morphfixed, followed by a number, determines whether parameters marked as fixed are included in the morph. If the number is 0, fixed parameters are not included and their values are left unchanged. If the number not zero, fixed parameters are included. The default behavior of **plugmorph** is to include fixed parameters.

morphhidden — In left inlet: The word morphhidden, followed by a number, determines whether parameters marked as hidden are included in the morph. If the number is 0, hidden parameters are not included and their values are left unchanged. If the number not zero, hidden parameters are included. The default behavior of **plugmorph** is to include hidden parameters.

store — In left inlet: The word store copies the current values of the target plug-in's parameters to its effect program.

float — In right inlet: The value is interpreted as a code to assign a new plug-in as a target. The outlet of a **pp** object is normally connected to this inlet.

Arguments

None.

Output

anything — Out left outlet: Output from this outlet of the **plugmorph** object occurs when a new plug-in is either inserted or deleted. The messages update an attached menu object with a new list of plug-ins that are potential targets.

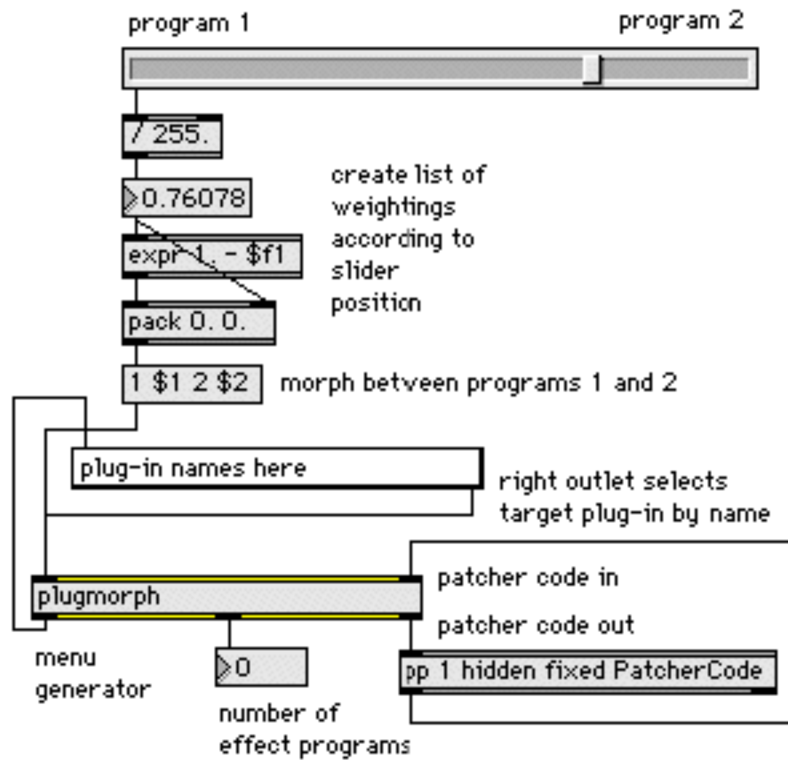
float — Out 2nd outlet: When a new plug-in is selected as a target, **plugmorph** outputs the number of effect programs it contains out this outlet.

Out right outlet: The current parameter code is output when the object's plug-in target changes via a message from the attached pop-up menu object sent to the object's left inlet, or when a new parameter code is received in the right inlet.

plugmorph

Generate parameter values from programs

Examples



See Also

umenu Pop-up menu, to display and send commands

plugmultiparam

Define multiple plug-in parameters

The **plugmultiparam** object lets you define three or more parameters that are displayed and changed by a single object. However, these parameters will be hidden from the Parameters view in the plug-in window; they can only be changed by creating a Max user interface. Primarily, **plug-multiparam** was designed to be used in conjunction with the multislider object; it can also work with the **plugstore** object, or simply a set of cleverly organized **pack** and **unpack** objects.

Input

int — The value at the specified parameter index is sent out the object's right outlet.

list — Interpreted as a set of values to be assigned to the object's parameters, starting at the lowest numbered parameter. If the list is longer than the number of parameters defined by the object, the extra elements are ignored. The values of the list are constrained to be within the minimum and maximum arguments of the object.

bang — Sends the currently stored values out the object's left outlet.

setmessage — The word setmessage, followed by a symbol, changes the message that sets individual values when they change (for example, because the stored program was changed). The default select message is useful in conjunction with the **multislider** object.

Arguments

int — Obligatory. Defines the starting parameter index to be covered by the object.

int — Obligatory. Defines the number of parameter indices to be covered by the object.

float or int — Optional. Sets the minimum value of the input and output for all parameters. The default value is 0.

float or int — Optional. Sets the maximum value of the input and output for all parameters. The default value is 1.

Example: 32 parameters whose value ranges between 1 and 99 are stored starting at parameter index 13 with the following arguments to **plugmultiparam**:

plugmultiparam 13 32 1 99

fixed — Optional. If the word fixed appears as an argument, the parameters will not be affected by the Randomize and Evolve commands in the parameter pop-up menu available in the plug-in edit window when the user holds down the command key and clicks in the interface. This is appropriate for gain parameters, where randomization usually produces irritating results.

Output

list — Out left outlet: The left outlet produces the current values as a list when the object receives a bang message.

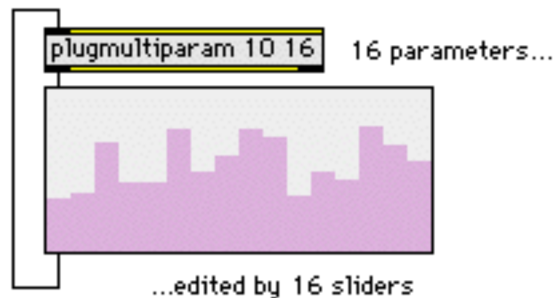
any message — Out left outlet: The **plugmultiparam** object also produces a message to set individual values in the collection using the following format

<message name> <index> value

By default, the message name is select—this is appropriate for setting one value in a **multislider** object. You can change the name to something else with the setmessage message described above. The index argument starts at 0 for the first parameter and goes up by 1 for each subsequent parameter—it is not affected by the starting parameter index argument to **plugmultiparam**. The index argument is followed by the current parameter value.

float — Out right outlet: When an int message is received, the value at the specified parameter index is output.

Examples



See Also

plugstore	Store multiple plug-in parameter values
pp	Define a plug-in parameter
Pluggo Tutorial P4	Using **multislider** and **plugmultiparam**

plugout~

Define a plug-in's audio outputs

plugin~ and **plugout~** define the signal inputs and outputs to a plug-in. You can use them within Max as simple thru objects, feeding **plugin~** a test signal and routing the output of **plugout~** to a **dac~** object. When **plugin~** and **plugout~** are operating within the runtime environment however, they act differently. **plugin~** ignores its input and instead outputs the plug-in's signal inputs fed to it by the host mixer. **plugout~** does not output any type of signal out its outlets; instead it feeds its signal inputs to the plug-in's audio outputs to the host mixer.

Input

signal In left and right inlets: When used in Max/MSP, the **plugout~** object echoes its input to its output. When used in the runtime plug-in environment, the input to **plugout~** is copied to the audio outputs of the plug-in.

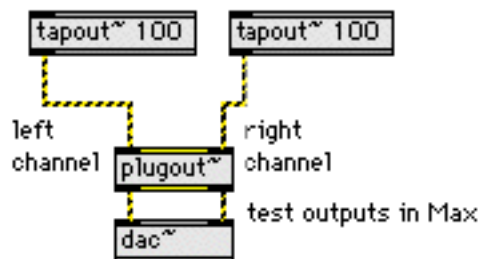
Arguments

int Optional. One or two int arguments, if present, specify the output channel destination (within the plug-in). If no arguments are present, **plugout~** has two outlets assigned to channels 1 and 2.

Output

signal When used in Max/MSP, the signal output of the **plugout~** object is simply its signal input. When used in the runtime plug-in environment, the signal output to the outlets is undefined, and the input is copied to the audio outputs of the plug-in.

Examples



See Also

plugin~ Define a plug-in's audio inputs

plugphasor~

Host-synchronized sawtooth wave

plugphasor~ outputs an audio-rate sawtooth wave that is sample-synchronized to the beat of the host sequencer. The waveform can be fed to other audio objects to lock audio processes to the audio of the host.

Input

None.

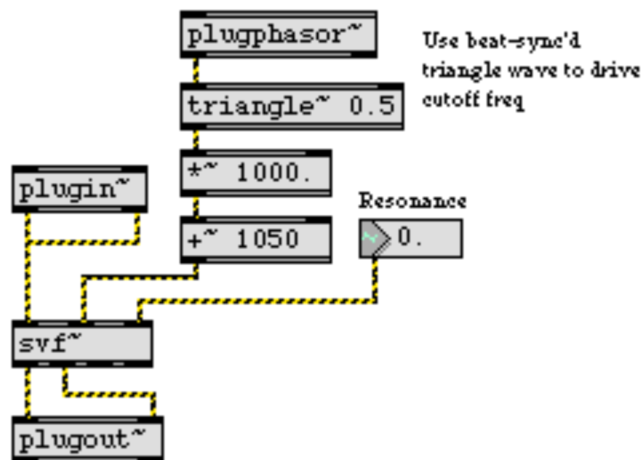
Arguments

None.

Output

signal The output of **plugphasor~** is analogous to **phasor~**: it ramps from 0 to 1.0 over the period of a beat. If the current host environment does not support synchronization or the host's transport is stopped, the output of **plugphasor~** is a zero signal.

Examples



Drive an oscillator with a beat-synced ramp wave

See Also

plugsync~ Report host synchronization information

plugreceive~

Receive audio from another plug-in

The **plugreceive~** and **plugsend~** objects are used to send audio signals from one plug-in to another. They are used in the implementation of the PluggoBus feature of many of the plug-ins included with pluggo.

Input

signal — The input to the **plugreceive~** object comes from a **plugsend~** object to which it is currently connected. Initially, this will be a **plugsend~** having the same name as the **plugreceive~** object's argument.

set — The word set, followed by a symbol naming a **plugsend~** object, connects the **plugreceive~** object to the specified **plugsend~** object(s), and the **plugreceive~** object's audio output becomes the input to the **plugsend~**. If the symbol doesn't name a **plugsend~** object, the audio output becomes zero.

Arguments

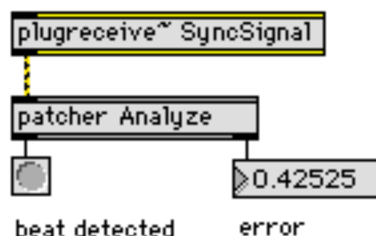
symbol — Obligatory. Gives the **plugreceive~** object a name used for connecting with one or more **plugsend~** objects.

Output

signal — The audio signal input to the **plugsend~** objects connected to this object. If no **plugsend~** objects are connected, the audio output is zero.

There may be a delay of one processing (I/O) vector size of the host mixer between the **plugreceive~** output and the inputs to the plug-in which the **plugreceive~** is located. This occurs when a **plugsend~** occurs later in the processing chain than the **plugreceive~** to which it is sending audio.

Examples



See Also

plugsend~	Send audio to another plug-in

plugsend~

Send audio to another plug-in

The **plugsend~** and **plugreceive~** objects are used to send audio signals from one plug-in to another. They are used in the implementation of the PluggoBus feature of many of the plug-ins included with pluggo.

Input

signal The input to the **plugsend~** object is mixed with other **plugsend~** objects, which can be in the same plug-in or a different plug-in, and is then sent out the signal outlets of any connected **plugreceive~** objects.

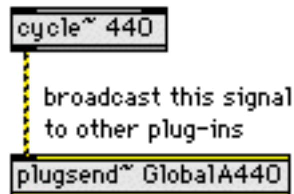
Arguments

symbol Obligatory. Gives the **plugsend~** object a name used for connecting with other **plugsend~** and **plugreceive~** objects.

Output

None.

Examples



See Also

plugreceive~	Receive audio from another plug-in

plugstore

Store multiple plug-in parameter values

The **plugstore** object works with **plugmultiparam** to allow you to get values into and out of **plugmultiparam** from multiple locations in a patcher.

Input

bang	Sends the stored list out the object's outlet.
list	Stores the elements of the list (up to the size of the object) and repeats them to the object's outlet.
select	The word select, followed by an index and value, stores the value at the specified index (starting at 1 for the first element) and sends the stored list out the object's outlet.
set	The word set, followed by an index and value, stores the value at the specified index (starting at 1 for the first element) but does not output the stored list.

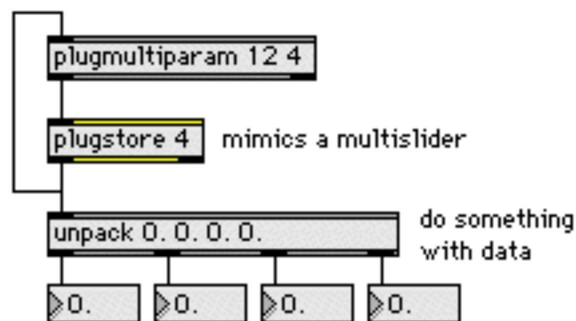
Arguments

int	Obligatory. Sets the number of elements stored in the **plugstore** object's list.

Output

list	The stored list is output whenever a list, bang, or select message is received.

Examples



See Also

plugmultiparam	Define multiple plug-in parameters

plugsync~

Report host synchronization information

The **plugsync~** object provides information about the current state of the host. Sample count information is available in any host; even Max. The validity of the other information output by the object is dependent upon what synchronization capabilities the host implements; the value from the flags (9th) outlet tells you what information is valid. Output from **plugsync~** is continuous when the scheduler is running.

Input

None.

Arguments

None.

Output

int — Out left outlet: 1 if the host's transport is currently running; 0 if it is stopped or paused.

int — Out 2nd outlet: The current bar count in the host sequence, starting at 1 for the first bar. If the host does not support synchronization, there is no output from this outlet.

int — Out 3rd outlet: The current beat count in the host sequence, starting at 1 for the first beat. If the host does not support synchronization, there is no output from this outlet.

float — Out 4nd outlet: The current beat fraction, from 0 to 1.0. If the host does not support synchronization, the output is 0. If the host does not support synchronization, there is no output from this outlet.

list — Out 5th outlet: The current time signature as a list containing numerator followed by denominator. For instance, 3/4 time would be output as the list 3 4. If the host does not support time signature information, there is no output from this outlet.

float — Out 6th outlet: The current tempo in samples per beat. This number can be converted to beats per minute using the following formula: (sampling-rate / samples-per-beat) * 60. If the host does not support synchronization, there is no output from this outlet.

float — Out 7th outlet: The current number of beats, expressed in 1 PPQ. This number will contain a fractional part between beats. If the host does not support synchronization, there is no output from this outlet.

float — Out 8th outlet: The current sample count, as defined by the host.

int Out 9th outlet: A number representing the validity of the other information coming from **plugsync~**. Mask with the following values to determine if the information from **plugsync~** will be valid.

Sample Count Valid	1 (always true)
Beats Valid	2 (2nd, 3rd, 4th, and 7th outlets valid)
Time Signature Valid	4 (5th outlet valid)
Tempo Valid	8 (6th outlet valid)
Transport Valid	16 (left outlet valid)

See Also

plugphasor~	Host-synchronized sawtooth wave

poke~

Write sample values by index

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~** at the 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

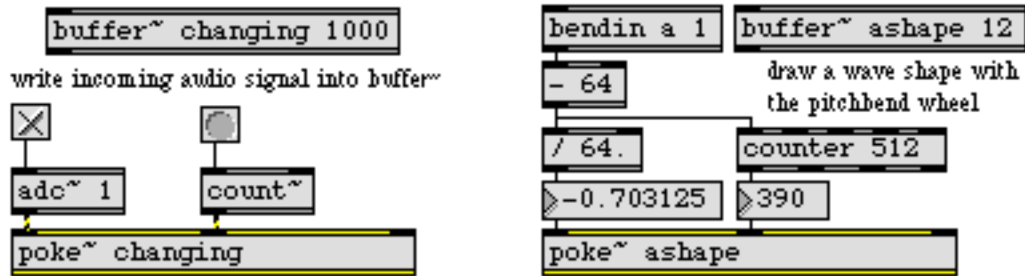
symbol — Obligatory. Names the **buffer~** where **poke~** will write its incoming samples.

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

Output

None.

Examples



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

See Also

buffer~	Store audio samples
buffir~	Buffer-based FIR filter
peek~	Read and write sample values

poltocar~

Input

signal In left inlet: The magnitude (amplitude) of the frequency bin to be converted into a cartesian (real/imaginary) signal pair.

In right inlet: The phase of the frequency bin to be converted into a cartesian (real/imaginary) signal pair.

Arguments

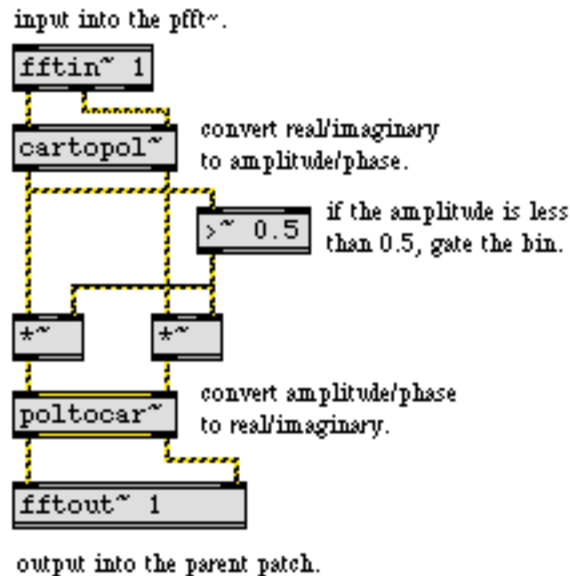
None.

Output

signal Out left outlet: The real part of a frequency domain signal suitable for input into an **ifft~** or **fftout~** object.

Out right outlet: The imaginary part of a frequency domain signal suitable for input into an **ifft~** or **fftout~** object.

Examples



poltocar~ *converts amplitude/phase pairs into the Cartesian pairs that* **fftout~** *uses*

See Also

cartopol	Cartesian to Polar coordinate conversion
cartopol~	Signal Cartesian to Polar coordinate conversion
fft~	Fast Fourier transform
fftin~	Input for a patcher loaded by pfft~
fftinfo~	Report information about a patcher loaded by pfft~
fftout~	Output for a patcher loaded by pfft~
frameaccum~	Compute "running phase" of successive phase deviation frames
framedelta~	Compute phase deviation between successive FFT frames
ifft~	Inverse Fast Fourier transform
pfft~	Spectral processing manager for patchers
poltocar	Polar to Cartesian coordinate conversion
vectral~	Vector-based envelope follower
Tutorial 26	Frequency Domain Signal Processing with **pfft~**

The **poly~** object is similar to the **patcher** object: it lets you encapsulate a patcher inside an object box. However, as the name suggests, where the **patcher** object only has one copy of the encapsulated patcher, the **poly~** object allows one or more instances (copies) of a patcher to be loaded. You specify the patcher filename and the number of instances you want as arguments to **poly~**. The maximum number of instances is 1023.

The **poly~** object directs signals and events (messages) received in its inlets to **in** and **in~** objects inside patcher instances. The patcher can also contain **out** and **out~** objects to send signals or events to the outlets of the **poly~** object. Messages to the **poly~** object control audio processing in its loaded patcher instances and let you control the routing of events.

Input

anything — The number of inlets and outlets for **poly~** is determined by the patcher that is loaded. The inlets for the patcher loaded by a **poly~** object accept both signal and event connections.

The signals are routed inside of the loaded patcher by using the **in~** objects for signals or the **in** object for events. The number of total inlets in a **poly~** object is determined by the highest number of an **in~** or **in** object in the loaded patcher (e.g., if there is an **in~** with argument 3 and an **in** with argument 4, the **poly~** object will have four inlets. All the inlets accept signal connections even though there may not be an **in~** object corresponding to each inlet.

Signal inputs are fed to all instances.

any message — In any inlet: Messages are sent to the **in** objects in the **poly~** object's current target patcher instance(s). Messages received in the left inlet of **poly~** are sent to **in 1** objects, messages in the second inlet are sent to **in 2** objects, and so on.

signal — In any inlet: Sends a signal to the corresponding **in~** object in all patcher instances. Signals connected to the left inlet of **poly~** are received by all **in~ 1** objects, signals connected to the second inlet of **poly~** are sent to all **in~ 2** objects, and so on.

list — In any inlet: If you want to send a message to a **poly~** instance that starts with one of the words used to control the **poly~** object itself, prepend the message with the word list. For example, the message list target 2 sent to the left inlet of **poly~** will output target 2 out the outlet of all **in 1** objects, rather than changing the current target instance to the second patcher.

busymap — In left inlet: The word busymap, followed by a number which specifies a message outlet number, will report voice busy states out the specified message outlet of the **poly~** object.

down — In left inlet: The word down, followed by a number which is a power of 2, specifies that upsampling by the designated power of two is to be done on the currently loaded patcher. The message down 2 specifies downsampling by a factor of 2 (e.g.,

22050 Hz at a sampling rate of 44100 Hz). The new sampling rate used by the patcher will be set on the next compilation of the DSP chain; the down message does not force a recompilation of the DSP chain.

midinote — In left inlet: The word midinote, followed by one or more numbers, will send the data to the first **in** object of the first instance of the loaded patcher that has received a note-on message without a corresponding note-off message. The first number after the word midinote is the note number, followed by the velocity. As an example, sending midinote 60 64 to a **poly~** with two instances will mark the first one busy. A subsequent midinote 67 64 will be directed to the second patcher instance. Once a midinote 60 0 is received by the **poly~** object, it is sent to the first instance (since **poly~** keeps track of which instance received the note-on message). Similarly, a midinote 67 0 is directed to the second instance.

mute — In left inlet: The word mute, followed by a number and a zero or one, will turn signal processing off for the specified instance of a patcher loaded by the **poly~** object and send a bang message to the **thispoly~** object for the specified instance. When the second number is a 1 processing in the patcher instance is turned *off* (muted). When the second number is a 0, the processing in the patcher instance is turned *on*. The message mute 0 1 mutes all instances, and mute 0 0 turns on signal processing for all instances of the patcher.

mutemap — In left inlet: The word mutemap, followed by a number which specifies a message outlet number, will report voice mutes out the specified message outlet of the **poly~** object.

note — In left inlet: The word note, followed by a message, will send the data to the first **in** object of the first instance of the patcher that has not marked itself "busy" by sending a 1 to a **thispoly~** object inside the patcher instance.

open — In left inlet: The word open, followed by a number, opens the specified instance of the patcher. You can view the activity of any instance of the patcher up to the number of voices (set by the voices message or by an argument to the **poly~** object). You can use this message to view an individual instance of the patcher at work. With no arguments, the open message opens the instance that is currently the target (see the target message below).

steal — In left inlet: The word steal, followed by a zero or one, toggles *voice stealing*. If voice stealing is set using the steal 1 message, the **poly~** object sends the data from **note** or **midinote** to instances that are still marked "busy" — this can result in clicks depending on how the instances handle the interruption. The default is 0 (voice stealing off).

target — In left inlet: The word target, followed by a number, specifies the **poly~** instance that will receive subsequent messages (other than messages specifically used by the **poly~** object itself) arriving at the **poly~** object's inlets. target 0 turns off input to all instances. target 1 routes messages to the first instance, etc.

voices In left inlet: The word voices, followed by a number, changes the number of instances (copies) of the loaded patcher. Instances of the patcher are loaded or deleted as needed. The maximum number of instances is 1023.

up In left inlet: The word up, followed by a number which is a power of 2, specifies that upsampling by the designated power of two is to be done on the currently loaded patcher. The message up 2 specifies upsampling by a factor of 2 (e.g., 88200 Hz at a sampling rate of 44100 Hz). The new sampling rate used by the patcher will be set on the next compilation of the DSP chain. The up message does not force a recompilation of the DSP chain.

wclose In left inlet: The word wclose, followed by a number, will close the window which contains the instance of the loaded patcher identified by the numbered index. It is the complement to the open message. When used without the number argument, wclose will close the patcher window with the highest numbered index.

vs In left inlet: The word vs, followed by a number which is a power of 2 in the range 2-2048, specifies the signal vector size for the **poly~** object's loaded patch. The signal vector size will be set on the next compilation of the DSP chain. The vs message does not force a recompilation of the DSP chain. vs 0 specifies no fixed vector size. The default is the current signal vector size.

Arguments

symbol Obligatory. The first argument must be the name of a patcher.

Note: Unlike the **patcher** object, a subpatch window is *not* automatically opened for editing when a patcher argument is supplied for the **poly~** object; the patcher containing the object must already exist and be found in the Max/MSP search path.

int Optional. After the patcher name argument, the number of instances of the loaded patcher (which correspond to the number of available "voices") is specified. The default value is 1, and the maximum number of instances is 1023. The number of available voices may be dynamically changed by using the voices message.

local Optional. The word local, followed by a zero or one, toggles *local scheduling* for the **poly~** object's loaded patcher. Local scheduling means that the **poly~** object maintains its own scheduler that runs during its audio processing rather than using the global Max scheduler. This allows finer resolution for events generated by multiple patcher instances. However, no scheduling occurs if audio processing is turned off, either globally or locally for the **poly~** object or one or more of its instances. The default is off (local 0). Local scheduling cannot be changed by send-

ing messages to the **poly~** object. Scheduler locality is permanent for any patcher which is loaded.

up Optional. The word up, followed by a number which is a power of 2, specifies that upsampling by the designated power of two is to be done on the currently loaded patcher. The message up 2 specifies upsampling by a factor of 2 (e.g., 88200 Hz at a sampling rate of 44100 Hz). Although both up and down are permissible arguments to the **poly~** object, the down message takes precedence over up.

down Optional. The word down, followed by a number which is a power of 2, specifies that downsampling by the designated power of two is to be done on the currently loaded patcher. The message down 2 specifies downsampling by a factor of 2 (e.g., 22050 Hz at a sampling rate of 44100 Hz). Although both up and down are permissible arguments to the **poly~** object, the down message takes precedence over up.

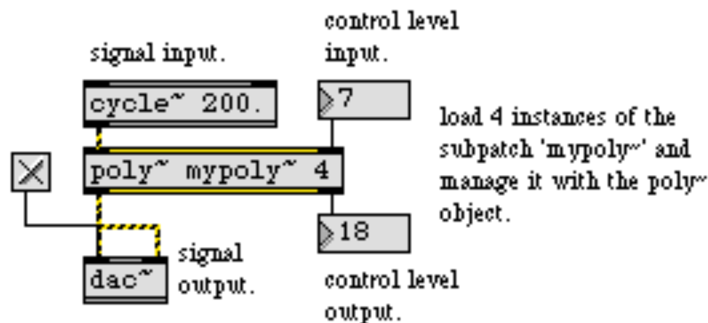
args Optional. The word args can be used to initialize any pound-sign arguments (e.g., #1) in the loaded patcher. If used, the args argument **must** be the last argument word used—everything which appears after the word args will be treated as an argument value.

Output

anything The number of outlets of a **poly~** object is determined by the sum of the highest argument numbers of the **out** and **out~** objects in the loaded patcher. For instance, if there is an **out 3** object and an **out~ 2** object, the **poly~** object will have five outlets. The signal outputs corresponding to the **out~** objects are leftmost in the **poly~** object, followed by the event outlets corresponding to the **out** objects.

Signals sent to the inlet of **out~** objects in each patcher instance are mixed if there is more than one instance and appear at the corresponding outlets of the **poly~** object.

Examples



The **poly~** *object manages multiple instances of a subpatch*

See Also

in	Message input for a patcher loaded by **poly~**
in~	Signal input for a patcher loaded by **poly~**
out	Message output for a patcher loaded by **poly~**
out~	Signal output for a patcher loaded by **poly~**
patcher	Create a subpatch within a patch
thispoly~	Control **poly~** voice allocation and muting
Tutorial 20	MIDI control: Sampler
Tutorial 21	MIDI control: Using the **poly~** object

Input

signal or float — In left inlet: All incoming signal or float values which exceed the high or low value ranges specified by arguments to the **pong~** object are either *folded* back into this range (i.e., values greater than one are reduced by one plus the amount that they exceed one, and negative values are handled similarly) or *wrapped* (i.e., values greater than one are reduced by two, and negative values are increased by two), according to the mode of the **pong~** object (see the mode message below).

In center or right inlet: The **pong~** objects accepts low and high range values for the range outside of which folding occurs. If the **pong~** object has either one or no arguments, **pong~** will have two inlets. The right inlet is used to set the high range value above which signal folding occurs, the low range value is assumed to be zero.

If the **pong~** object has two arguments, the object has three inlets. The center inlet specifies the low value range below which folding occurs, and the right inlet specifies the high range limit. The default object has no arguments, and the right inlet specifies the upper value.

If the current low range value is greater than the high range value, their behavior is swapped.

mode — The word mode, followed by a 0 or 1, sets the folding mode of the **pong~** object.

pong 0 sets the **pong~** object to *signal folding*. Values greater than one are reduced by one plus the amount that they exceed one, and negative values are handled similarly. This is the default mode of the object.

pong 1 sets the **pong~** object to *signal wrapping*. Values greater than one are reduced by two, and negative values are increased by two.

Arguments

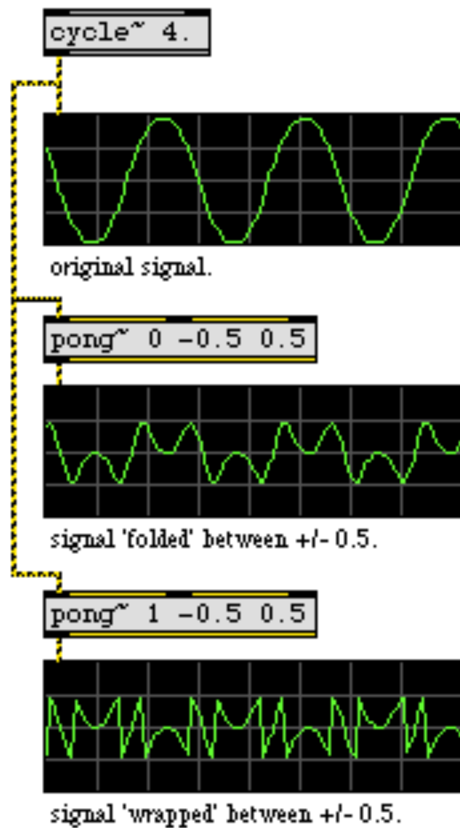
int — Optional. An optional argument is used to set the mode of the **pong~** A 0 sets signal folding (the default), and a 1 sets signal wrapping (see the mode message, above).

float — Optional. When used with the optional mode argument, the low and high range values for the **pong~** objects can be specified by two additional float arguments. If only one argument is given following the mode argument (e.g., pong~ 0 .1), it specifies the low range value of the **pong~** object above which folding occurs, and the high range value is set to 1.0 (the default). If two arguments are present, the first argument specifies the low range value and the second argument specifies the high range value.

Output

signal The folded signal or float value.

Examples



pong~ *distorts a signal by folding it or wrapping it around an upper and lower threshold level*

See Also

phasewrap~	Wrap a signal between -π and π

pow~

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.

Input

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.

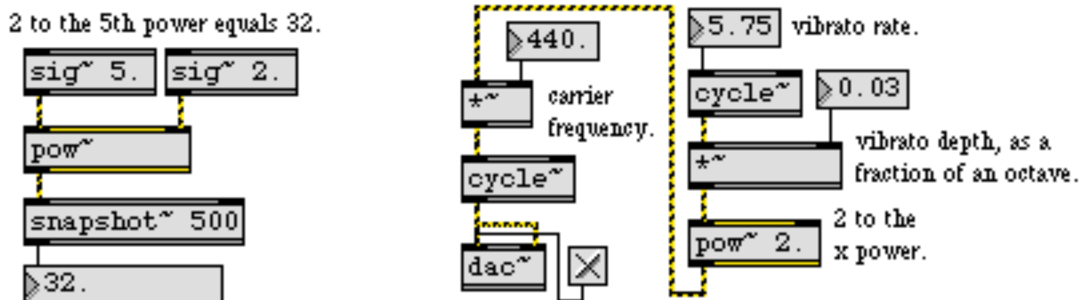
Arguments

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

The **pp** object (an abbreviation for plug-in parameter) defines plug-in parameters. It has a number of optional arguments that let you define the parameter minimum and maximum, hide the parameter from display, set the color of the egg slider associated with it, etc. You connect the output of the **pp** object to something you want to control with a stored parameter. If your plug-in will use a Max patcher interface, you need to connect the interface element that will change the parameter's value to the inlet of the **pp** object. The **pp** object will send new parameter values out its outlet at various times: when you move an egg slider, when the user switches to a new effect program, and when the host mixer is automating the parameter changes of your plug-in.

Internally, the **pp** object and the runtime plug-in environment store values between 0 and 1.0. By giving the **pp** object optional arguments for minimum and maximum, you can store and receive any range of values and the object will convert between the range you want and the internal representation. If for some reason you want to know the internal 0-1.0 representation, you can get it from the object's right outlet. If you want to send a value that is based on the internal 0-1.0 representation, use the rawfloat message.

Input

bang — Sends the current value of the parameter out the object's right outlet in its internal (unscaled) form between 0 and 1.0, then out the object's left outlet scaled by the object's minimum and maximum.

float or int — In left inlet: Sets the current value of the parameter and then sends the new value out the right and left outlets as described above for the bang message. The incoming number is constrained between the minimum and maximum values of the object.

float or int — In right inlet: Sets the current value of the parameter without any output. The incoming number is constrained between the minimum and maximum values of the object.

open — Same as choosing **Get Info…** from the Object menu.

text — The word text, followed by a single symbol, allows you to set the text displayed in the Parameters view of the plug-in edit window when the user moves the mouse over the egg slider corresponding to the parameter.

rawfloat — The word rawfloat, followed by a number between 0 and 1.0 sets the current parameter value to the number without scaling it by the object's minimum and maximum. The value is then send out the right and left outlets of the object as described above for the bang message.

(Get Info...) — Choosing **Get Info…** from the Object menu opens an Inspector for editing a description of the parameter displayed in the Parameters view of the plug-in edit window when the user moves the cursor over the egg slider corresponding to the parameter.

Inspector

The behavior of a **pp** object is displayed and can be edited using its Inspector. If you have enabled the floating inspector by choosing **Show Floating Inspector** from the Windows menu, selecting any **pp** object displays the **pp** Inspector in the floating window. Selecting an object and choosing **Get Info…** from the Object menu also displays the Inspector.

Typing in the *Describe Parameter* text area specifies the parameter description.

The *Revert* button undoes all changes you've made to an object's settings since you opened the Inspector. You can also revert to the state of an object before you opened the Inspector window by choosing **Undo Inspector Changes** from the Edit menu while the Inspector is open.

Arguments

The **pp** object takes a number of arguments. They are listed in the order that they need to appear.

int — Obligatory. The first argument sets the parameter number. The first parameter is 1. Parameter numbers should be consecutive (but they need not be), and two **pp** objects should not have the same parameter number. An error will be reported in the Messages view of the runtime plug-in environment if duplicate parameter numbers are encountered.

hidden — Optional. If the word hidden appears as an argument, the parameter will not be given an egg slider in the plug-in edit window and will not appear in the pop-up menu generated by the **plugmod** object.

fixed — Optional. If the word fixed appears as an argument, the parameter will not be affected by the Randomize and Evolve commands in the parameter pop-up menu available in the plug-in edit window when the user holds down the command key and clicks in the interface. This is appropriate for gain parameters, where randomization usually produces irritating results.

c2-c5 — Optional. If c2, c3, c4,or c5 appears as argument, the color of the egg slider is set to something other than the usual purple. Currently c2 is Wild Cherry, c3 is Turquoise, c4 is Harvest Gold, and c5 is Peaceful Orange.

symbol — Optional. The next symbol after any of the optional keywords names the parameter. This name appears in the Name column of the Parameters view and in the pop-up menu generated by the **plugmod** object.

float or int — Optional. After the parameter name, a number sets the minimum value of the parameter. The minimum and maximum values determine the range of values that are sent into and out of the **pp** object's outlets, as well as the displayed value in the Parameters view. The type of the minimum value determines the type of the

parameter values the object accepts and outputs. If the minimum value is an integer, the parameters will interpreted and output as integers. If the minimum value is a float, the parameters will be interpreted and output as floats.

float or int — Optional. After the minimum value, a number sets the maximum value of the parameter. The minimum and maximum values determine the range of values that are sent into and out of the **pp** object's outlets, as well as the displayed value in the Parameters view.

symbol — Optional. After the minimum and maximum values, a symbol sets the label used to display the units of the parameter. Examples include Hz for frequency, dB for amplitude, and ms for milliseconds.

choices — Optional. If the word choices appears after the minimum and maximum values, subsequent symbol arguments are taken as a list of discrete settings for the object and are displayed as such in the Parameters view. As an example pp 1 Mode 0 3 choices Thin Medium Fat would divide the parameter space into three values. 0 (anything less than 0.33) would correspond to Thin, 0.5 (and anything between 0.33 and 0.67) would correspond to Medium, and 1 (and anything between 0.67 and 1.0) would correspond to Fat. Only the name of the choice, rather than the actual value of the parameter, is displayed in the Parameters view.

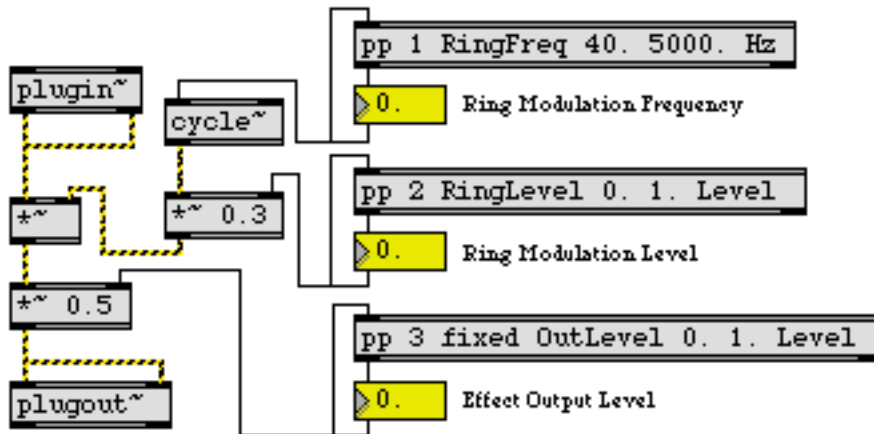
dB — Optional. If the word choices does not appear as argument, the word dB can be used to specify that the value of the parameter be displayed in decibel notation, where 1.0 is 0 dB and 0.0 is negative infinity dB.

Output

int or float — Out left outlet: The scaled value of the parameter is output when it is changed within the runtime environment or when a bang, int, float, or rawfloat message is received in the object's inlet. The parameter value can be changed in the runtime environment in the following ways: the user moves an egg slider, the parameter is being automated by the host mixer, or the user has selected a new effect program for the plug-in within the host mixer.

float — Out right outlet: The unscaled value of the parameter is output when it is changed by the runtime environment or when a bang, int, float, or rawfloat message is received in the object's inlet. You might use this value if you want to use a different value in your plug-in's computation than you display to the user.

Examples



See Also

plugmultiparam	Define multiple plug-in parameters
plugstore	Store multiple plug-in parameter values

Input

bang — Sends the current value of the mode parameter (0 to 3) out the object's right outlet and then sends the current value of the tempo parameter out the object's left outlet.

int — In left inlet: Sets the current value of the tempo parameter and then sends the new value out left outlet. The incoming number is constrained between the minimum and maximum values of the object.

In right inlet: Sets the current value of the mode parameter and then sends the new value out the right outlet. The number is constrained between 0 and 3. Mode values are as follows:

Value	*Description*
0	*Free Mode.* If there is an egg slider display associated with this parameter, it is disabled. It's assumed that another parameter will set the "tempo" in units of milliseconds or Hertz.
1	*Host Mode.* If there is an egg slider display associated with this parameter, it is enabled but the user cannot change it. Instead the tempo is set by the host and merely displayed by the slider. The patch should enable synchronizing to the host in some way (probably by using the **plugsync~** or **plugphasor~** objects).
2	*PluggoSync Mode.* This mode functions similarly to Host mode in that the egg slider is enabled but cannot be changed by the user. Instead the tempo is set by the host and merely displayed by the slider. The patch should enable synchronizing to PluggoSync in some way.
3	*User-Defined Tempo (UDT) Mode.* In this mode, there is no synchronization and the user can change the tempo slider to any desired value. The patch should use this value to calculate some sort of time-based behavior.

set — In right inlet: The word set, followed by a number, sets the sync mode parameter to the number but does not output the sync mode and the tempo.

rawfloat — In left inlet: The word rawfloat, followed by a number between 0 and 1, sets the tempo to a value scaled between the minimum and maximum values scaled by the number. For example, if the minimum tempo were 100 and the maximum were 200, the message rawfloat 0.5 would set the tempo to 150.

In right inlet: The word rawfloat, followed by a number between 0 and 1, sets the sync mode parameter to a value based on multiplying the number by 3 and truncating. Numbers below 0.33 set the sync mode to 0 (Free), numbers between 0.33 and 0.66 set it to Host, numbers at or above 0.67 and less than 1 set it to PluggoSync, and numbers equal to 1 set it to User-Defined Tempo.

rawlist — The word rawlist, followed by two numbers, is equivalent to sending the rawfloat message with the first number to the left inlet and the rawfloat message with the second number to the right inlet.

(Get Info...) — Choosing **Get Info…** from the Object menu opens an Inspector for editing a description of the parameter displayed in the Parameters view of the plug-in edit window when the user moves the cursor over the egg slider corresponding to the parameter.

Inspector

A parameter description can be assigned to a **pptempo** object and can be edited using its Inspector. If you have enabled the floating inspector by choosing **Show Floating Inspector** from the Windows menu, selecting any **pptempo** object displays the **pptempo** Inspector in the floating window. Selecting an object and choosing **Get Info…** from the Object menu also displays the Inspector.

Typing in the *Describe Parameter* text area specifies the parameter description.

The *Revert* button undoes all changes you've made to an object's settings since you opened the Inspector. You can also revert to the state of an object before you opened the Inspector window by choosing **Undo Inspector Changes** from the Edit menu while the Inspector is open.

Arguments

int — Obligatory. A number greater than or equal to 1 sets the parameter index of the tempo parameter.

int — Obligatory. A number greater than or equal to 1 sets the parameter index of the sync mode parameter.

hidden — Optional. If the word hidden appears as an argument, the parameter will not be given an egg slider in the plug-in edit window and will not appear in the pop-up menu generated by the **plugmod** object.

fixed — Optional. If the word fixed appears as an argument, the parameter will not be affected by the Randomize and Evolve commands in the parameter pop-up menu available in the plug-in edit window when the user holds down the command key and clicks in the interface.

c2-c5 — Optional. If c2, c3, c4, or c5 appears as argument, the color of the egg slider is set to something other than the usual purple. Currently c2 is Wild Cherry, c3 is Turquoise, c4 is Harvest Gold, and c5 is Peaceful Orange.

symbol — Optional. The next symbol after any of the optional keywords names the tempo parameter. This name appears in the Name column of the Parameters view and in

the pop-up menu generated by the **plugmod** object. The name of the sync mode parameter will be the name of the tempo parameter followed by the word mode. The default parameter name is *ParamN*, where *N* is the index assigned to the tempo parameter by the first argument to **pptempo**.

float or int — Optional. After the parameter name, a number sets the minimum value of the parameter. The minimum and maximum values determine the range of values that are sent into and out of the **pptempo** object's left inlet and outlet, as well as the displayed value in the Parameters view of the plug-in edit window. The type of the minimum value determines the type of the parameter values the object accepts and outputs. If the minimum value is an integer, the parameters will interpreted and output as integers. If the minimum value is a float, the parameters will be interpreted and output as floats.

float or int — Optional. After the minimum value, a number sets the maximum value of the parameter. The minimum and maximum values determine the range of values that are sent into and out of the **pptempo** object's left inlet and outlet, as well as the displayed value in the Parameters view of the plug-in edit window.

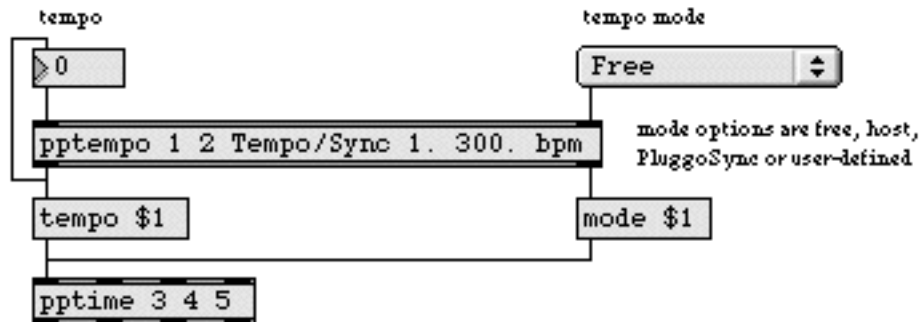
(Get Info...) — Optional. Choosing **Get Info…** from the Object menu opens an Inspector for editing a description of the parameter that is displayed in the Parameters view of the plug-in edit window when the user moves the cursor over the egg slider corresponding to the parameter.

Output

float or int — Out left outlet: The scaled value of the tempo parameter is output when it is changed within the runtime environment or when a bang, int, float, or rawfloat message is received in the object's inlets. The parameter value can be changed in the runtime environment in the following ways: the user moves an egg slider, the parameter is being automated by the host mixer, or the user has selected a new effect program for the plug-in within the host mixer.

Out right outlet: The value of the sync mode parameter, between 0 and 3, when the parameter is changed within the runtime environment, an int, float, or rawfloat message is received in the object's right inlet, or a bang message is received in the object's inlets. The modes are described above in the Input section.

Examples



pptempo *provides tempo and synchronization information to* **pptime**

See Also

pp	Define a plug-in parameter
pptime	Define a time-based plug-in parameter

pptime

Define a time-based plug-in parameter

The **pptime** object defines time-based plug-in parameters for use in plug-ins which provide synchronization with a host sequencing application. Like the **pp** object, **pptime** has a number of optional arguments that let you define the parameter and control the appearance when using the generic plug-in interface.

The **pptime** object supports the four modes of host synchronization. The functionality of the object varies according to its mode of operation. In *Free mode*, **pptime** works like **pp** for the ms/Hz parameter using the leftmost inlet and outlet. In *Host sync mode* and *Pluggo Sync mode, t*he egg-slider display changes to a smaller slider plus a unit value pop-up menu. When a change to either the slider or menu is made, the beat value output (rightmost) produces a value you can feed to a **rate~** object. The *User-Defined Tempo mode* expects a tempo value to be fed to **pptime** via the tempo message (you can use **pptempo** for this). **pptime** then calculates the ms/Hz value based on the current tempo, unit multiplier, and unit value and outputs the value out the leftmost outlet.

Input

float or int — In left inlet: Sets the parameter indices for the ms/Hz value.

In second inlet: Sets the unit multiplier value. Values are in the range 0.0-15.0.

In third inlet: Sets the unit index. The unit index is expressed in terms of float or int values between 0 and 18, with each number representing a unit of musical subdivision. The unit indices are defined as follows:

unit index	*note value*
0	1
1	1/2
2	1/2. (dotted half)
3	1/2t (1/2 triplet)
4	1/4
5	1/4. (dotted 1/4)
6	1/4t (1/4 triplet)
7	1/8
8	1/8. (dotted 1/8)
9	1/8t (1/8 triplet)
10	1/16
11	1/16. (dotted 1/16)
12	1/16t (1/16 triplet)
13	1/32
14	1/32. (dotted 1/32)
15	1/32 (1/32 triplet)
16	1/64
17	1/64. (dotted 1/64)
18	1/64 (1/64 triplet)

In fourth inlet: Sets the unit value input.

bang — Sends the current value of the parameter out the object's left outlet.

mode	In left inlet: The word mode, followed a number in the range 0-3, specifies the host sync mode. Host sync modes are defined as follows: 0=Free, 1=Host Sync, 2=Pluggo Sync, 3=User-Defined Tempo. The default is 1 (Free mode).
open	Same as choosing **Get Info…** from the Object menu.
rawfloat	The word rawfloat, followed by a number between 0 and 1.0 sets the current parameter value to the number without scaling it by the object's minimum and maximum. The value is then send out the right and left outlets of the object as described above for the bang message.
timesig	In left inlet: The word timesig, followed by two numbers, are used to specify the time signature. The time signature (composed of a numerator and denominator) is used to calculate the beat value in sync modes and the ms/Hz value in User-Defined Tempo mode. This list can be fed from the output of the plugsync~ object. The default is 4/4 (timesig 4 4).
tempo	In left inlet: If the pptime object is in User-determined Tempo mode, the word tempo, followed a number, specifies the current tempo, and send the ms/Hz value associated with that tempo out the left outlet.
(Get Info...)	Choosing **Get Info…** from the Object menu opens an Inspector window for editing a description of the parameter that is displayed in the Parameters view of the plug-in edit window when the user moves the cursor over the egg slider corresponding to the parameter. This command is not available in the runtime plug-in environment.

Inspector

A parameter description can be assigned to a **pptime** object and can be edited using its Inspector. If you have enabled the floating inspector by choosing **Show Floating Inspector** from the Windows menu, selecting any **pptime** object displays the **pptime** Inspector in the floating window. Selecting an object and choosing **Get Info…** from the Object menu also displays the Inspector.

Typing in the *Describe Parameter* text area specifies the parameter description.

The *Revert* button undoes all changes you've made to an object's settings since you opened the Inspector. You can also revert to the state of an object before you opened the Inspector window by choosing **Undo Inspector Changes** from the Edit menu while the Inspector is open.

Arguments

The **pptime** object takes three required arguments plus numerous optional ones. They are listed in the order that they need to appear.

float Obligatory. The three required float arguments are the parameter indices for the ms/Hz value, the multiplier value, and the unit index.

hidden Optional. If the word hidden appears as an argument, the parameter will not be given an egg slider in the plug-in edit window and will not appear in the pop-up menu generated by the **plugmod** object.

fixed Optional. If the word fixed appears as an argument, the parameter will not be affected by the Randomize and Evolve commands in the parameter pop-up menu available in the plug-in edit window when the user holds down the command key and clicks in the interface. This is appropriate for gain parameters, where randomization usually produces irritating results.

c2-c4 Optional. If c2, c3, or c4 appears as argument, the color of the egg slider is set to something other than the usual purple. Currently c2 is Wild Cherry, c3 is Turquoise, and c4 is Harvest Gold.

symbol Optional. The next symbol after any of the optional keywords names the parameter. This name appears in the Name column of the Parameters view and in the pop-up menu generated by the **plugmod** object.

float or int Optional. After the parameter name, a number sets the minimum value of the parameter. The minimum and maximum values determine the range of values that are sent into and out of the **pptime** object's outlets, as well as the displayed value in the Parameters view. The type of the minimum value determines the type of the parameter values the object accepts and outputs. If the minimum value is an integer, the parameters will interpreted and output as integers. If the minimum value is a float, the parameters will be interpreted and output as floats.

float or int Optional. After the minimum value, a number sets the maximum value of the parameter. The minimum and maximum values determine the range of values that are sent into and out of the **pptime** object's outlets, as well as the displayed value in the Parameters view.

symbol Optional. After the minimum and maximum values, a symbol sets the label used to display the units of the parameter. Examples include Hz for frequency, dB for amplitude, and ms for milliseconds.

Output

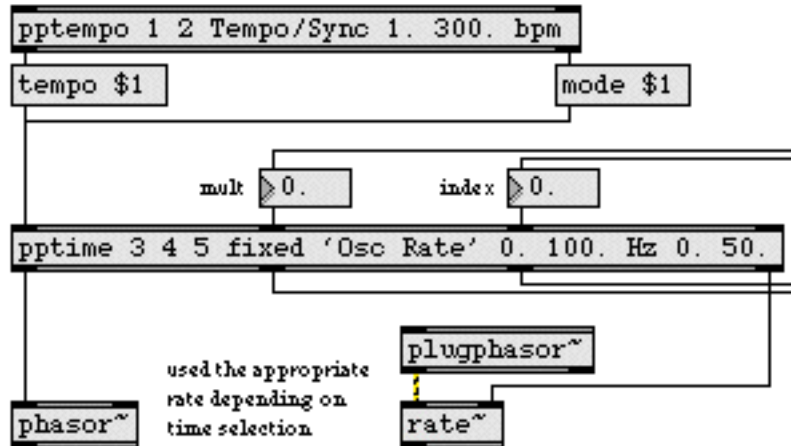
int or float Out left outlet: The scaled value of the parameter is output when it is changed within the runtime environment or when a bang, int, float, or rawfloat message is received in the object's inlet. The parameter value can be changed in the runtime environment in the following ways: the user moves an egg slider, the parameter is being automated by the host mixer, or the user has selected a new effect program for the plug-in within the host mixer.

Out second outlet: The unit multiplier value. Values are in the range 0.0-15.0.

Out third outlet: The unit index. The unit index is expressed in terms of float or int values between 0 and 18

Out fourth Outlet: The beat value output.

Examples



Use **pptime** *to control beat- and/or time-syncronized parameters*

See Also

pp	Define a plug-in parameter
pptempo	Define plug-in tempo and sync parameters

rampsmooth~

Smooth an incoming signal

Input

signal or float — A signal or value to be smoothed. Each time an incoming value changes, the **rampsmooth~** object begins a linear ramp over a specified number of samples to reach the new value.

ramp — In left inlet: The word ramp, followed by a number, specifies the number of samples over which an signal will be smoothed. Each time an incoming value changes, the **rampsmooth~** object begins a linear ramp of the specified number of samples to reach the new value. The default value is 0.

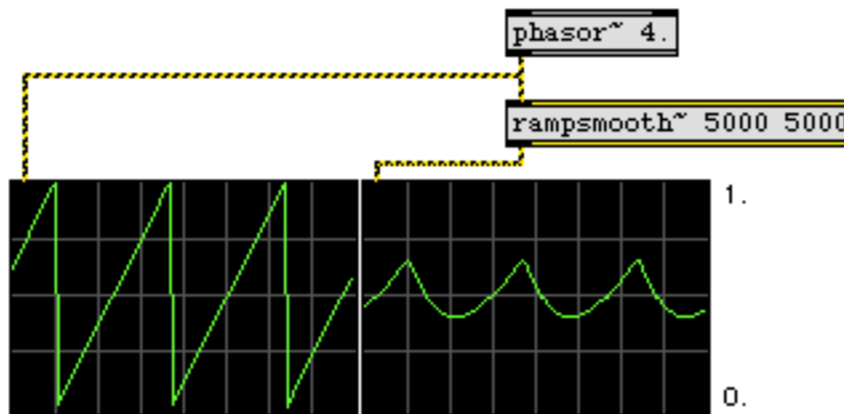
rampdown — In left inlet: The word rampdown, followed by a number, specifies the number of samples over which an signal will be smoothed when an incoming value less than the current value arrives.

rampup — In left inlet: The word rampup, followed by a number, specifies the number of samples over which an signal will be smoothed when an incoming value greater than the current value arrives.

Arguments

int — Optional. The number of samples across which to generate a ramp up or ramp down can be specified by a pair of numbers.

Examples



rampsmooth~ *performs linear smoothing on an input signal*

See Also

slide~	Filter a signal logarithmically

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.

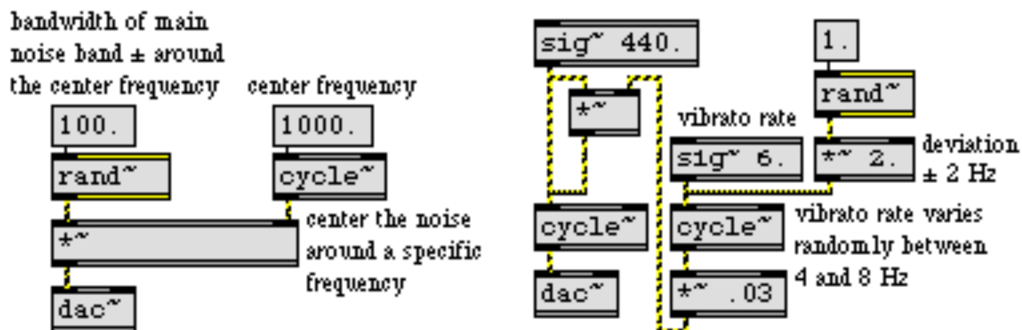
Arguments

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

line~	Linear ramp generator
noise~	White noise generator
pink~	Pink noise generator

Input

signal — In left inlet: An input signal from a **phasor~** object. The **rate~** object time scales the input signal from a **phasor~** by a *multiplier value*. The multiplier value can be specified as an argument or received as a float to the **rate~** object's right inlet.

float — In left inlet: Sets the phase value for the **rate~** object's signal output.

In right inlet: The signal multiplier value used to scale the **phasor~** signal input. Float values less than 1.0 create several ramps per phase cycle. Numbers greater than 1.0 create fewer ramps. This can be useful for synchronizing multiple processes to a single reference **phasor~** object, preserving their ratio relationships.

goto — In left inlet: The word goto, followed by a float, causes the **rate~** object to jump immediately to the specified value. An optional second argument may be used to specify the time at which to jump to the value (e.g., goto 1.0 .5 will output a value of 1.0 at the halfway point of the **phasor~** object's input signal ramp).

reset — In left inlet: The word reset will lock the output to the input on its next reset. It is equivalent to the message goto 0. 0.

sync — In left inlet: The word sync, followed by a number between 0 and 2 or the words cycle, lock, or off, sets the sync mode of the **rate~** object. The sync mode determines whether or not the **rate~** "in" will stay in phase with the input signal, and the method used for synchronization. When the output of the **rate~** object is "in phase," the input and output signals align precisely at the least common multiple of their periods (i.e., they pass through zero and begin a new cycle at precisely the same time). If the signals are in phase, and a new multiplier value is received, the **rate~** object changes the frequency of its output ramp accordingly. However, the change in multiplier values means that the two signals may be out of phase. The **rate~** object handles this situation in one of three different ways, depending on the sync mode: The sync modes are described below:

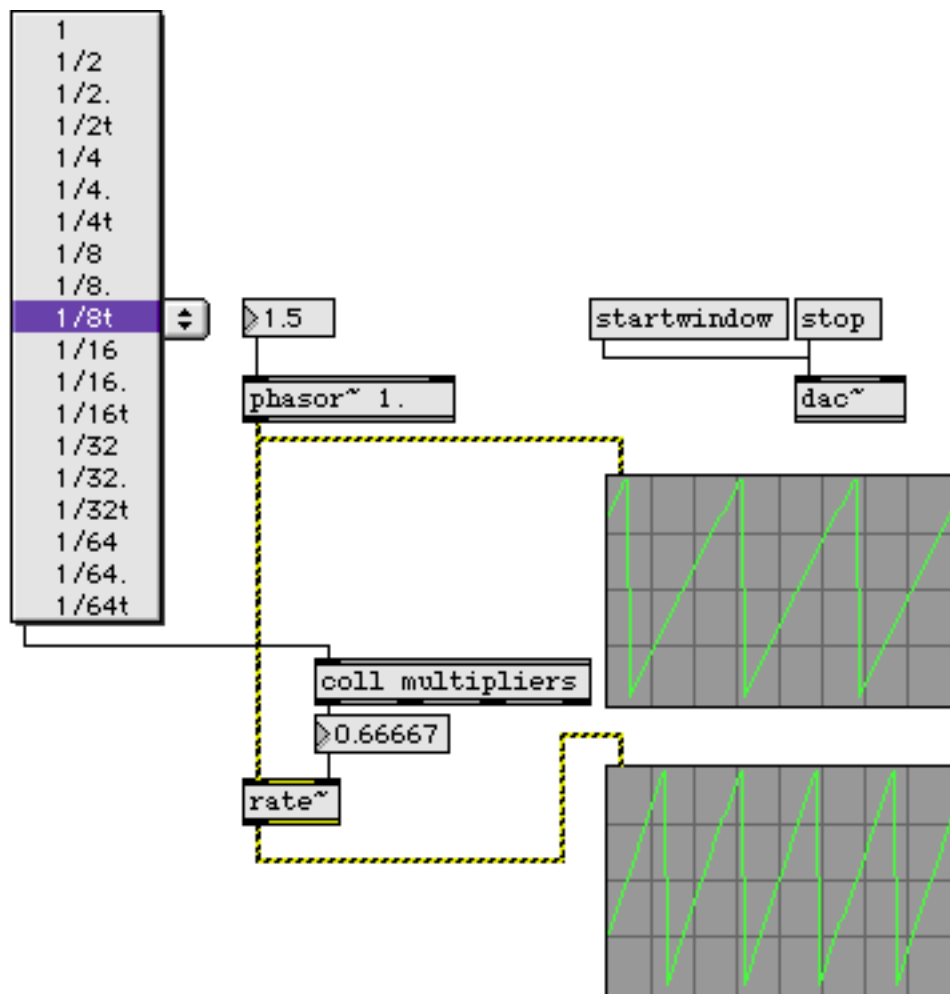
mode	*description*
cycle	The messages sync 0 or sync cycle set the *cycle* mode of the **rate~** object (the default mode). In cycle mode, the **rate~** object does not change the phase of its output until the end of the current cycle. When the input ramp reaches its peak and starts over from zero, the rate~ object immediately restarts the output ramp, causing a discontinuity in the output signal, and immediate phase synchronization.
lock	The messages sync 1 or sync lock set the *lock* mode of the **rate~** object. In sync lock mode, the **rate~** object performs synchronization whenever a new multiplier is received. The **rate~** object immediately calculates the proper ramp position which corresponds to being "in phase" with the new multiplier value, and jumps to that position.

off The messages sync 2 or sync off disables the sync mode of the **rate~** object. In this mode **rate~** never responds to phase differences; when a new multiplier is received, the **rate~** object adjusts the speed of its output ramps and they continue without interruption. Since this mode never introduces a discontinuous jump in the ramp signal, it may be useful if phase is unimportant.

Arguments

float Optional. The multiplier value used to scale the output signal.

Examples



Use **rate~** *to generate synchronized waveforms or control sources*

See Also

phasor~ Sawtooth waveform 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.

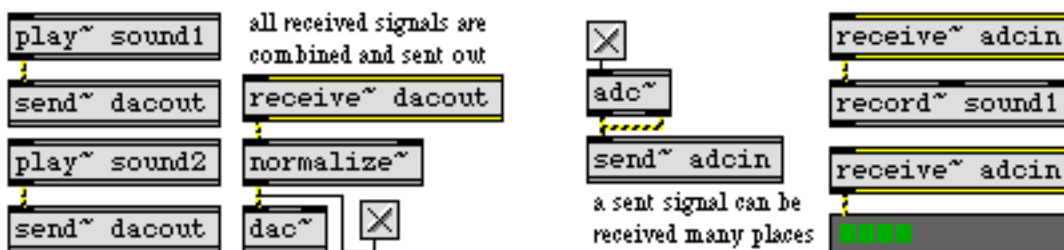
Arguments

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

record~

Record sound into a buffer

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~** object'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. The **record** object also takes into account any changes in the **buffer~** object's sampling rate if the **buffer~** object's length is modified for the purpose of establishing loop points.

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

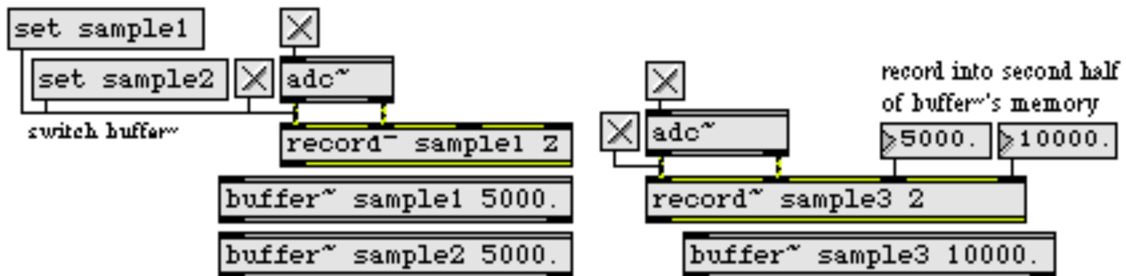
symbol — Obligatory. Names the **buffer~** where **record~** will write the recorded samples.

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

2d.wave~	Two-dimensional wavetable
buffer~	Store audio samples
groove~	Variable-rate looping sample playback
play~	Position-based sample playback
Tutorial 13	Sampling: Recording and playback

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 by the center frequency divided by the filter bandwidth. 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.

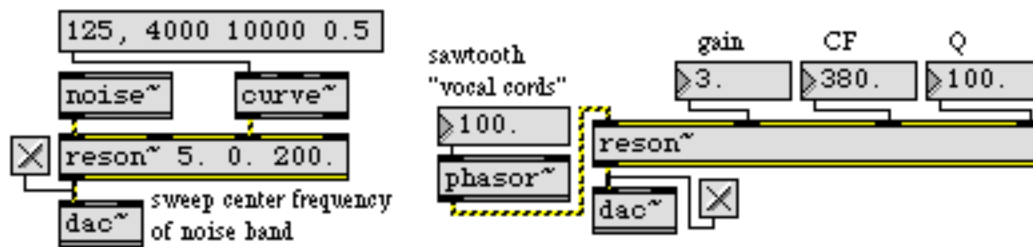
Output

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

rewire~

The ReWire system connects audio applications together. It allows a program that generates audio (a *client*) to feed it into a program that plays audio (a *mixer*).

The **rewire~** object requires a properly installed ReWire client to be installed and available. The **rewire~** object allows MSP to be a ReWire mixer; there can only be one mixer active at any one time.

You can use several **rewire~** objects. Each object is associated with one ReWire client.

rewire~ is intended to be used with other ReWire-compatible software synthesizers. For a list of compatible applications, visit the Propellerheads web site at *http://www.propellerheads.se.*

ReWire is a trademark of Propellerhead Software AS.

Input

bang — In left inlet: If a ReWire device has been loaded, bang causes a list of its output channel names to be sent out the second-from-right outlet.

int — In left inlet: 1 starts the ReWire transport, 0 stops it. No sound can occur without the transport being started.

play — In left inlet: Starts the ReWire transport.

stop — In left inlet: Stops the ReWire transport.

openpanel — In left inlet: If the current device has a user interface panel, the word openpanel will open it.

closepanel — In left inlet: Closes the current device's user interface panel if it is open.

device — In left inlet: The word device, followed by a number, switches to the ReWire device associated with the number index. The index is obtained as the order in which device names appear in a pop-up menu object connected to the second-to-right outlet.

any symbol — In left inlet: The symbol is interpreted as the name of a ReWire device. If the name is valid, **rewire~** attempts to switch to the device.

tempo — In left inlet: The word tempo, followed by a number, sets the tempo to that number in beats per minute. ReWire only handles integer tempos, and tempo is updated on the next call to the client to return audio samples.

position — In left inlet: The word position, followed by a number, sets the current play position (in samples).

loop — In left inlet: The word loop, followed by three numbers, sets the current loop position and mode. The first number sets the loop start position in samples. The sec-

ond number sets the loop end position in samples. If the third number is 1, looping is turned on. If the third number is 0, looping is turned off. However, note that ReWire clients may ignore looping if they do not produce transport- or time-based output. For example, a software synthesizer that only responds to MIDI note commands would probably not be affected by looping.

midi In left inlet: The word midi, followed by four or five numbers, sends a MIDI event to a ReWire device. The first number is a time stamp value and is currently ignored (in other words, the event is sent out immediately). The second number is the MIDI bus index. ReWire 2 has 256 MIDI busses, indexed from 0 to 255. The third number is the MIDI message status byte, and the fourth and fifth numbers are the MIDI message data bytes.

map The word map, followed by two numbers, maps a ReWire device's output channel to an outlet of the **rewire~** object. ReWire channels start at 1 with a maximum of 256. **rewire~** object outlets are specified starting at 1 for the left outlet, or 0 to turn the ReWire channel off. For example, map 3 2 causes the ReWire device's audio output channel 3 to be mapped to the second-from-left outlet of the **rewire~** object. You can find out the names of the ReWire audio output channels with the bang message after the **rewire~** object has a connection to a ReWire device. By default, audio outlets map to the first channels of the ReWire device; in other words, the leftmost signal outlet outputs the first channel of the device.

Arguments

symbol Optional. If present, a ReWire device name can be specified. **rewire~** will attempt to open the device when the object is initialized.

int Optional. Specifies the number of audio outputs the **rewire~** object will have. If no argument is present, one audio outlet is created. The maximum number of outlets is 256.

Output

signal Out audio outlets (starting at left): The audio signal output from the ReWire device is sent out the **rewire~** object's outlets. By default, the leftmost outlet outputs the first channel of the device, but this mapping can be changed with the map message.

symbol Out fourth-from-right outlet: Messages indicating the transport state of the ReWire device. The position message with an int argument reports the transport position in 15360 PPQ. The play and stop messages report when the transport is started and stopped.

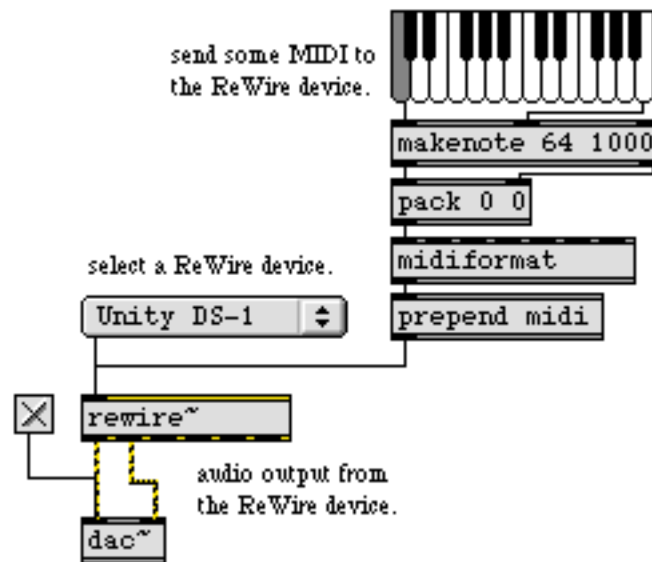
MIDI Out third-from-right outlet: MIDI events received from the ReWire device are sent out this outlet preceded by the word midi. The first argument is always 0 (it is the time stamp), the second argument is the ReWire MIDI bus index, the third

argument is the MIDI status byte, and the fourth and (optional) fifth arguments are the MIDI data bytes.

symbol Out second-from-right outlet: A list of the currently available ReWire devices in response to the bang message.

symbol Out right outlet: A list of the currently available device output names (in channel order) for the currently used ReWire device.

Examples



rewire~ *allows MIDI communication to and signal output from ReWire compatible devices*

See Also

vst~ Host VST plug-ins

round~

Input

signal In left inlet: A signal whose values will be rounded.

In right inlet: A signal whose value is used for rounding. Signal values received in the left inlet will be rounded to either the absolute nearest integer multiple or the nearest integer multiple between the value received in this inlet or 0 (See the nearest message for more information).

nearest In left inlet: The word nearest, followed by a non-zero value, will cause the **round~** object to round its input to the nearest absolute integer multiple of the value received in the right inlet. The default is on. nearest 0 will cause the **round~** object to round the input signal to the nearest integer multiple between the value received in the right inlet and zero (for positive numbers this will round down).

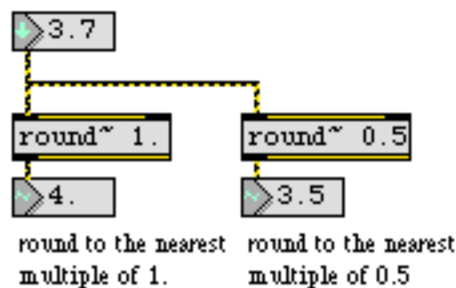
Arguments

int or float Optional. Sets the value the input signal will be rounded to.

Output

signal The rounded input signal.

Examples



round~ *takes floating-point signals and rounds them to a specific increment*

See Also

rampsmooth~	Smooth an incoming signal
slide~	Filter a signal logarithmically
trunc~	Truncate fractional signal values

sah~

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.

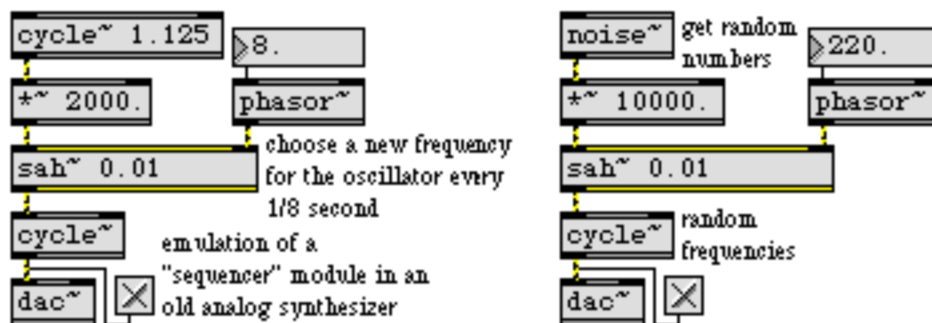
Arguments

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

sampstoms~

Convert samples to milliseconds

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

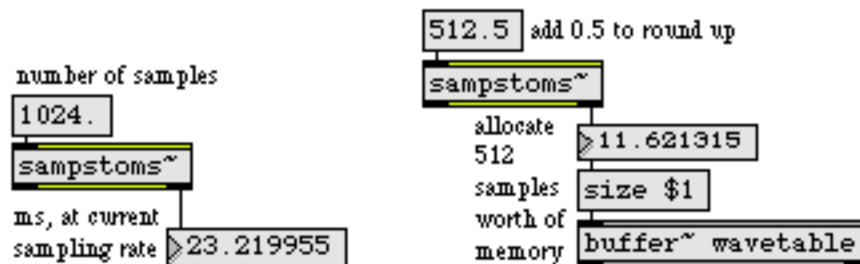
None.

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

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 signal objects 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.

brgb — The word brgb, followed by three numbers between 0 and 255, sets the RGB values for the background color of the **scope~** object's display. The default value is set by brgb 135 135 135.

bufsize — The word bufsize, followed by a number, changes the number of samples stored in the buffer used by the **scope~** object.

drawstyle — The word drawstyle, followed by a non-zero number, toggles an alternate drawing style for the **scope~** object which may make some waveforms more easily visible. The default is off (drawstyle 0).

frgb — The word frgb, followed by three numbers between 0 and 255, sets the RGB values for the color of the **scope~** object's waveform display. The default value is set by frgb 102 255 51.

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.

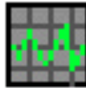
(mouse) — When you click on a **scope~**, its display freezes for as long as you hold the mouse button down.

Inspector

The behavior of a **scope~** object is displayed and can be edited using its Inspector. If you have enabled the floating inspector by choosing **Show Floating Inspector** from the Windows menu, selecting any **scope~** object displays the **scope~** Inspector in the floating window. Selecting an object and choosing **Get Info…** from the Object menu also displays the Inspector.

The **scope~** Inspector lets you specify the following attributes:

Buffers per Pixel sets the number of buffers per pixel which the **scope~** object displays. The default is 25. *Buffer Size* specifies the number of samples stored in the buffer used by the **scope~** object. The default is 128. The *Range* number boxes set the minimum and maximum values for the **scope~** display. The default *Min.* value is -1.0, and the default *Max.* value is 1.0. The *Delay* value sets the number of milliseconds of delay before **scope~** begins collecting values. The *Trigger Mode* checkboxes let you specify *Line Up* (default) or *Line Down* modes (see the trigger message, above). *Trigger Level* sets the trigger level used by modes 1 and two of the **scope~** display (see the triglevel message in Input for more information) The default trigger level is 0.



The *Colors* pull-down menu lets you use a swatch color picker or RGB values to specify the colors used for phosphor and background of the **scope~** display. display by the **scope~** object. *Phosphor* sets the color the **scope~** object uses for its display. The default phosphor color is 102 255 51. *Background* sets the **scope~** object's background color. The default value is 135 135 135.

The *Revert* button undoes all changes you've made to an object's settings since you opened the Inspector. You can also revert to the state of an object before you opened the Inspector window by choosing **Undo Inspector Changes** from the Edit menu while the Inspector is open.

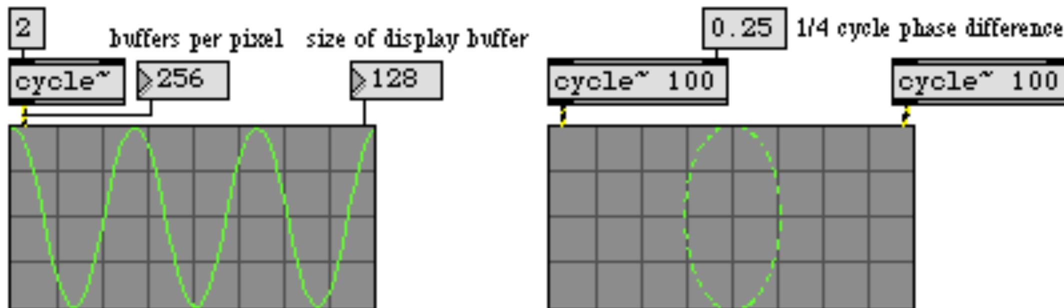
Arguments

None.

Output

None.

Examples



Display a signal, or plot two signals in X-Y mode

See Also

meter~	Visual peak level indicator
Tutorial 24	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 left-most 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.

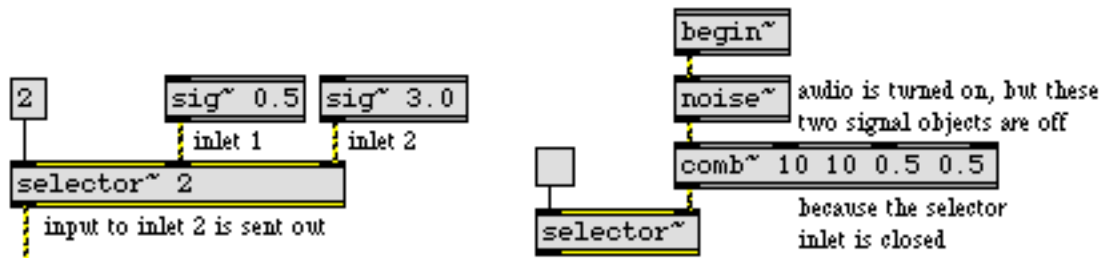
Arguments

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

send~

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).

clear — The clear message clears all of the **receive~** buffers associated with the **send~** object. This message is only used with patchers which are being muted inside a subpatch loaded by the **poly~** object.

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.)

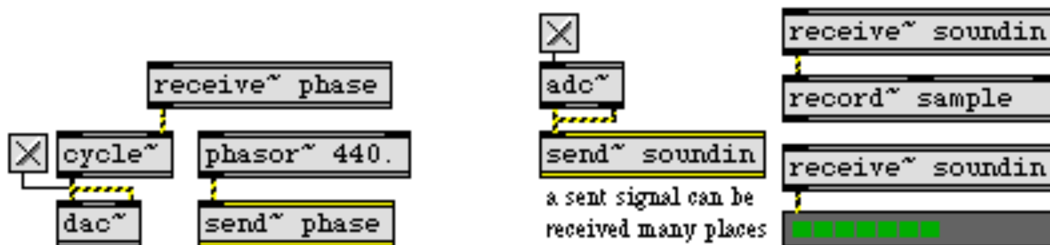
Arguments

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

signal — An input signal whose output is between 0. and 1.0 (usually the output of a **phasor~**) is used to drive the event sequencer.

any message — The **seq~** object is used to record and play back messages. All events received in the inlet are stored according to the current value of the input signal. Any message can be sequenced except for commands to the **seq~** object itself. The example shows a simple way to work around this limitation.

Note: **seq~** can be used to sequence MIDI data if the MIDI input stream is converted into lists of MIDI events. This conversion is necessary to avoid outputting a corrupted MIDI stream which would occur if only the raw int messages of a MIDI stream were sequenced individually and the **seq~** object were not doing a simple forward linear playback.

bang — Causes information about the **seq~** object's current sequence number, mode of operation (record, overdub, play) and total number of current events to be printed in the Max window.

add — The word add, followed by an int, a float and a message, inserts a Max event specified by the message at the time specified by the float for the sequence number specified by the int. (e.g., add 2 0.5 honk will insert the message honk to be played at the halfway point of sequence 2.)

dump — Causes the contents of all stored event sequences to be sent out the right outlet. The word dump, followed by a number, outputs only the sequence designated by the number.

erase — Erases all current sequences.

overdub — The word overdub, followed by 1, causes **seq~** to begin Max event recording of the current sequence (set by the seqnum message) in "overdub" mode. Recording begins at the current point of the loop and *wraps around* at the point where the input signal reaches 1, continuing to record as the signal passes its original value. The message overdub 0 turns off overdub mode.

play — The word play, followed by 1, causes **seq~** to begin Max event playback of the current sequence (set by the seqnum message) at the point of the loop specified by the current value of the signal input. play 0 turns off playback. By default, playback is off.

read — Reads a text file containing Max event sequences created using the **seq~** object's write message into the memory of the **seq~** object. If no symbol argument appears after the word read, a standard open file dialog is opened showing available text files. The word read, followed by a symbol, reads the file whose filename corresponds to the symbol into the **seq~** object's memory without opening the dialog box.

record — The word record, followed by 1, causes **seq~** to begin recording events into the current sequence (set by the seqnum message) at the point of the loop specified by the current value of the signal input. record 0 turns off playback. By default, recording is off.

seqnum — The word seqnum, followed by a number or symbol, sets the current Max event sequence being recorded or played back.

write — Saves the contents of all current Max event sequences 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 **seq~** object. If the patch has not yet been saved, the **seq~** file is saved in the same folder as the Max application.

Arguments

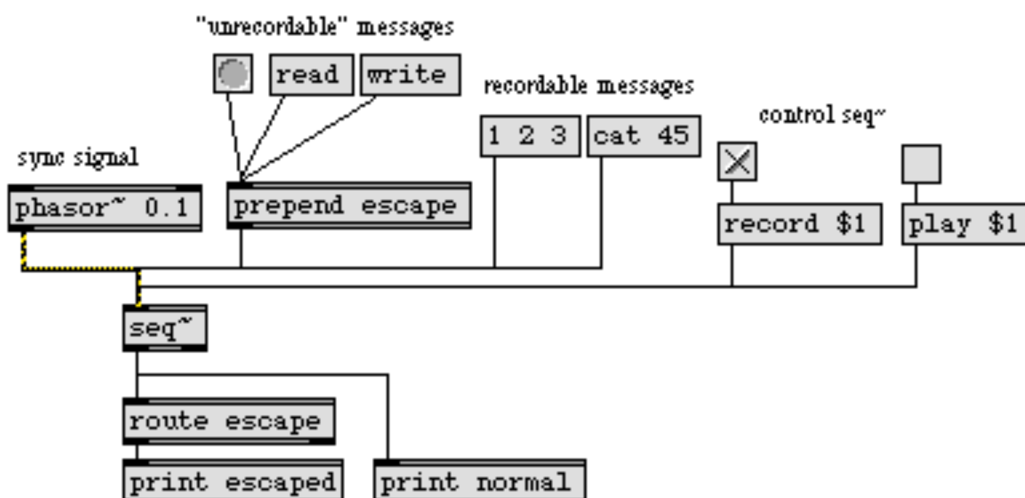
None.

Output

any message — Out left outlet: When playback is enabled with the play 1 message, the **seq~** object outputs all events recorded at the time specified by the input signal.

list — Out right outlet: The dump message will cause the **seq~** object to output the contents of a specified sequence to be output in the form of a list consisting of an int which specifies the sequence number, a float which specifies the signal value associated with that point in time, and the int, float, symbol or list to be output at that time.

Examples



See Also

phasor~	Sawtooth wave generator

sfinfo~

Report audio file information

Input

open — The word open, followed by a name of an audio 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, file length in milliseconds, sample type, and filename 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.

getnamed — In left inlet: The word getnamed, followed by a symbol which specifies the name of an **sfplay~** object, interrogates the named **sfplay~** object and reports the number of channels, sample size, sample rate, file length in milliseconds, sample type, and filename out its 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 audio file.

Out 2nd outlet: The audio file's sample size in bits (typically 16).

float — Out 3rd outlet: The audio file's sampling rate.

Out 4th outlet: The duration of the audio file in milliseconds.

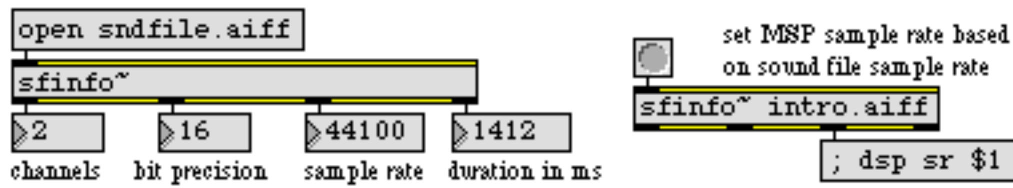
symbol — Out 5th outlet: the sample type of the audio file.

The following types of sample data are supported:

int8	8-bit integer
int16	16-bit integer
int24	24-bit integer
int32	32-bit integer
float32	32-bit floating-point
float64	64-bit floating-point
mulaw	8-bit µ-law encoding
alaw	8-bit a-law encoding

Out 6th outlet: The filename of the audio file

Examples



Report information about a specific audio file

See Also

info~	Report information about a sample
sflist~	Store audio file cues
sfplay~	Play audio file from disk
Tutorial 16	Sampling: Record and play audio files

sflist~

Input

open — The word open, followed by the name of an AIFF, WAV, NeXT/Sun or Sound Designer II (Macintosh only) audio file, opens the file if it is located 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.

clear — The word clear with no arguments clears all defined cues. After a clear message is received, only the number 1 will play anything (assuming there's an open file). The word clear followed by one or more cue numbers removes them from the **sflist~** object's cue list.

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.

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.

openraw — The openraw message functions exactly like open, but allows you to open any type of file for playback and make it the current file. The openraw message assumes that the file being opened is a 16-bit stereo file sampled at a rate of 44100 Hz, and assumes that there is no header information to ignore (i.e., an offset of 0). The file types can be explicitly specified using the samptype, offset, srate, and srchans messages.

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.

After the start and/or end time arguments, a optional directional buffer flag is used to enable reverse playback of stored cues. Setting this flag to 1 enables reverse cue playback. The default setting is 0 (bidirectional buffering off).

A final optional argument is used to set the playback speed. A float value sets the playback speed for an **sfplay~** object relative to the object's global playback speed—set by the speed message. The default value is 1.

Each cue that is defined requires approximately 40K of memory per **sfplay~** channel at the default buffer size (40320), with bidirectional buffering turned off. With bidirectional buffering turned on, the amount of memory per cue is doubled.

print Prints a list of all the currently defined cues.

samptype The word samptype, followed by a symbol, specifies the sample type to use when interpreting the audio file's sample data (thus overriding the audio file's actual sample type). This is sometimes called "header munging." When reading files in response to the openraw message, the assumed sample type is 16-bit integer. Modifications using samptype make no changes to the file on disk.

The following types of sample data are supported:

int8	8-bit integer
int16	16-bit integer
int24	24-bit integer
int32	32-bit integer
float32	32-bit floating-point
float64	64-bit floating-point
mulaw	8-bit µ-law encoding
alaw	8-bit a-law encoding

srcchans The word srcchans, followed by a number, specifies the number of channels in which to interpret the audio file's sample data (thus overriding the audio file's actual number of channels). This is sometimes called "header munging." When reading files in response to the openraw message, the assumed number of channels is 2. Modifications using srcchans make no changes to the file on disk.

Arguments

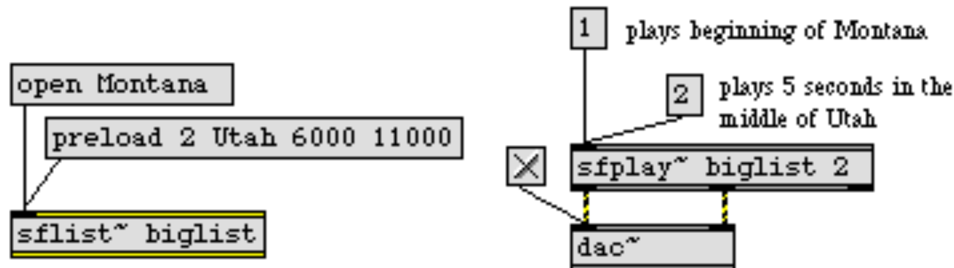
symbol Obligatory. Names the **sflist~**. **sfplay~** objects use this name to refer to cues stored inside the object.

int Optional. Sets the buffer size used to preload audio files. The default and minimum is 16384. Preloaded buffers are 4 times the buffer size per channel of the audio file.

Output

None.

Examples



Store a global list of cues that can be used by one or more **sfplay~** *objects.*

See Also

buffer~	Store audio samples
groove~	Variable-rate looping sample playback
play~	Position-based sample playback
sfinfo~	Report audio file information
sfplay~	Play audio file from disk
sfrecord~	Record to audio file on disk
Tutorial 16	Sampling: Record and play audio files

Input

float In right inlet: Defines the playback rate of an audio file. A value of 1.0 plays the audio file at normal speed. A playback rate of -1 plays the audio file backwards at normal speed. A playback rate of 2 plays the audio file at twice the normal speed. A playback rate of .5 plays the audio file at half the normal speed.

signal In left inlet: An input signal may be used for the sample-accurate triggering of prestored cues. When a signal value is received in the left inlet, the integer portion of the signal value is monitored. When the integer portion of the input signal changes to a value equal to the index of a prestored cue, that cue is triggered. Negative values are ignored.

In right inlet: The playback rate of an audio file can also be defined by a signal, allowing for playback speed change over time for vibrato or other types of speed effects. The same conventions with respect to number value and sign and playback rate apply as for float values.

int In left inlet: 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.

anything In left inlet: 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.

clear In left inlet: The word clear with no arguments clears all defined cues. After a clear message is received, only the number 1 will play anything (assuming there's an open file). The word clear followed by one or more cue numbers removes them from the **sfplay~** object's cue list.

embed In left inlet: 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.

fclose — In left inlet: 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.

list — In left inlet: Gives a set of cues for **sfplay~** to play, one after the other. The maximum number of cues is in a list is 128. Cue numbers (set using the preload message) can be any integer greater than or equal to 2.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.

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

modout — In left inlet: The word modout, followed by 1, turns on *modulo output*. If the number of channels in a audio file is less than the number of outputs for the sfplay~ object, the **sfplay~** object will reduplicate the audio file's channels across all of **sfplay~** object's outputs (rather than outputting zero) if modulo output is enabled. For example, a mono audio file loaded into an **sfplay~** object with two outputs will be played with the mono channel sent out both outputs of the object if modulo output is enabled. Similarly a stereo audio file will be played on an **sfplay~** object with four outlets with the left channel played on outputs 1 and 3, while the right will be played on outputs 2 and 4. The message modout 0 disables this feature.

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

offset — In left inlet: The word offset, followed by a number, specifies the sample start offset in bytes. The default value is 0. This value useful for aligning samples and avoiding playback of header information.

open — In left inlet: followed by the name of an AIFF, WAV, NeXT/Sun, raw format, or Sound Designer II (Macintosh only) audio file or CD-audio track, opens the 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.

openraw In left inlet: The openraw message functions exactly like open, but allows you to open any type of file for playback and make it the current file. The openraw message assumes that the file being opened is a 16-bit stereo file sampled at a rate of 44100 Hz, and assumes that there is no header information to ignore (i.e., an offset of 0). The file types can be explicitly specified using the samptype, offset, srate, and srchans messages.

pause In left inlet: The pause message causes the audio file playback to pause at its current playback position. Playback can be restarted with the resume message.

preload In left inlet: 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.

After the start and/or end time arguments, a optional directional buffer flag is used to enable reverse playback of stored cues. Setting this flag to 1 enables reverse cue playback. The default setting is 0 (bidirectional buffering off).

A final optional argument is used to set the playback speed. A float value sets the **sfplay~** object's playback speed relative to the object's global playback speed—set set by either the speed message or the **sfplay~** object's right inlet. The default value is 1.

Each cue that is defined requires approximately 40K of memory per **sfplay~** channel at the default buffer size (40320), with bidirectional buffering turned off. With bidirectional buffering turned on, the amount of memory per cue is doubled.

The preload message is always deferred to low priority. The pause, resume, and int messages are not. If you have problems with these messages arriving before you want them to in overdrive mode(i.e., before you've preloaded the most recent cue), use the **defer** object.

print	In left inlet: Prints information about the state of the object, plus a list of all the currently defined cues.
resume	In left inlet: If playback was paused, playback resumes from the paused point in the file.
samptype	In left inlet: The word samptype, followed by a symbol, specifies the sample type to use when interpreting the audio file's sample data (thus overriding the audio file's actual sample type). This is sometimes called "header munging." When reading files in response to the openraw message, the assumed sample type is 16-bit integer. Modifications using samptype make no changes to the file on disk.

The following types of sample data are supported:

int8	8-bit integer
int16	16-bit integer
int24	24-bit integer
int32	32-bit integer
float32	32-bit floating-point
float64	64-bit floating-point
mulaw	8-bit µ-law encoding
alaw	8-bit a-law encoding

seek — In left inlet: 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.

NOTE: The seek message is always deferred to low priority. If you have problems with these messages arriving before you want them to in overdrive mode(i.e. before you've finished seeking to a new location), then use the **defer** object.

set — In left inlet: 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.

speed — In left inlet: The word speed, followed by a number, sets an overall multiplier on the playback rate of all cues played by the object. A value of 1.0 (the default) plays all cues at normal speed. A playback rate of -1 plays all cues backward at normal speed. A playback rate of 2 plays the cues at twice their defined speed. A playback rate of 0.5 plays cues at half their defined speed. For example, if a cue has a playback rate of 2, and the speed is set to 3, the cue will play back at 6 times the normal speed.

srate — In left inlet: The word srate, followed by a number, specifies the sampling rate (Hertz) at which to interpret the audio file's sample data (thus overriding the audio file's actual sampling rate). This is sometimes called “header munging.” When reading files in response to the openraw message, the assumed sampling rate is 44,100 Hz. Modifications using srate make no changes to the file on disk.

srcchans — In left inlet: The word srcchans, followed by a number, specifies the number of channels in which to interpret the audio file's sample data (thus overriding the audio file's actual number of channels). This is sometimes called “header munging.” When reading files in response to the openraw message, the assumed number of channels is 2. Modifications using srcchans make no changes to the file on 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 signal outlets that the **sfplay~** object will have. The maximum number of channels is 28. The default is 1. If the audio file being played has more output channels than the **sfplay~** object, higher-numbered channels will not be played. If the audio file has fewer channels, the signals coming from the extra outlets of **sfplay~** will be 0.

An additional optional argument can be used to specify the disk buffer size in samples. If this argument has a value of 0, the default disk buffer size will be used.

An additional optional argument can be used to create outlets to the sfplay~ object which display positioning information. Specifying a final argument of 1 creates a single outlet to the left of the rightmost "bang on finish or halt" outlet which outputs a signal value which corresponds to the current playback position in milliseconds.

Like all MSP audio signals, this playback position is a 32-bit single precision floating-point signal. If greater precision is desired, specifying a final argument of 2 creates a second outlet which outputs a second 32-bit single precision floating-point signal containing the single precision roundoff error. Together these signals provide near double precision floating-point accuracy. (Note: after several minutes a single precision floating-point value is no longer sample accurate) Using the two signals together with objects such as the unsupported Max/ MSP high resolution signal processing objects like **hr.+~**, one may perform sample-accurate calculations based on file position

symbol — Optional. If the last argument is a symbol, it specifies a name by which other objects can refer to the **sfplay~** object to access its contents.

Output

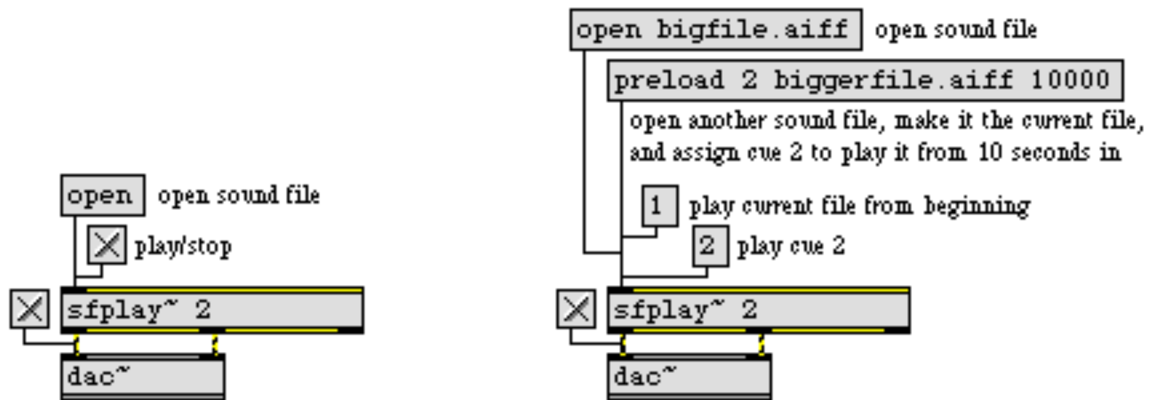
signal — There is one signal outlet for each of the **sfplay~** object's specified output channels (set by or as an argument to the **sfplay~** object) that sends out the audio data of the corresponding channel of the audio file when a cue number is received in the inlet. (The left outlet plays channel 1, and so on.)

If the optional output position argument is specified, there will be one or two signal outputs following the channel outputs whose signal outputs display positioning information. If the argument is 1, a single outlet to the left of the rightmost "bang on finish or halt" outputs a signal containing the current playback position in milliseconds. Specifying a final argument of 2 creates a second outlet which outputs a signal containing the playback position single precision roundoff error

in milliseconds (see Arguments for a more detailed description of the **sfplay~** object's position outlets).

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



Audio files can be played from the hard disk, without loading the whole file into memory

See Also

buffer~	Store audio samples
groove~	Variable-rate looping sample playback
play~	Position-based sample playback
sfinfo~	Report audio file information
sflist~	Store audio file cues
sfrecord~	Record to audio file on disk
Tutorial 16	Sampling: Record and play audio files

sfrecord~

Input

open — In left inlet: Opens a file for recording. By default, the file type is AIFF, but sfrecord~ also supports NeXT/Sun, WAV, and Sound Designer II (Macintosh only) formats. The word open without a filename argument brings up a standard Save As dialog allowing you to choose a filename. The optional symbols aiff, au, raw, wave, or sd2 (Macintosh only) specify the file format (which can also be set in the Save As dialog with a Format pop-up menu). If open is followed by another symbol, it creates a file in the current default volume. An existing file with the same name will be overwritten. The format symbol (e.g., aiff) can follow the optional filename argument.

int — In left inlet: If a file has been opened with the open message, a non-zero value begins recording, and 0 stops recording and closes the file. **sfrecord~** requires another open 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.

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

nchans — The word nchans, followed by a number in the range 1-28, sets the number of channels for the audio file to be recorded. The default is 1.

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.

resample — The word resample, followed by a float, will upsample or downsample the file. Sample rates are expressed as floating-point values—1.0 is the current sampling rate, 0.5 is half the current. 2.0 is twice the current sample rate, etc.

samptype — In left inlet: The word samptype, followed by a symbol, specifies the sample type to use when recording the audio file (thus overriding the audio file's actual sample type). This is sometimes called “header munging.” When reading files in response to the openraw message, the assumed sample type is 16-bit integer.

The following types of sample data are supported:

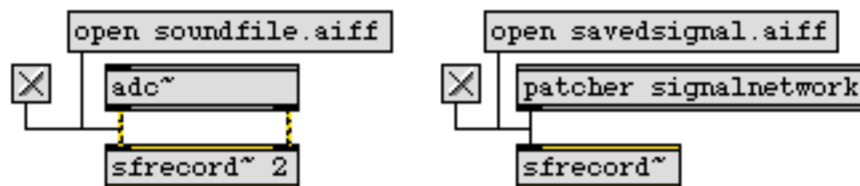
int8	8-bit integer
int16	16-bit integer
int24	24-bit integer
int32	32-bit integer
float32	32-bit floating-point
float64	64-bit floating-point
mulaw	8-bit µ-law encoding
alaw	8-bit a-law encoding

signal — Each inlet of **sfrecord~** accepts a signal which is recorded to a channel of an audio file when recording is turned on.

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 28, and the default is 1. The audio 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 an audio file containing "real world" sound and/or sound created in MSP

See Also

sfplay~	Play audio file from disk
Tutorial 16	Sampling: Record and play audio files

sig~

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.

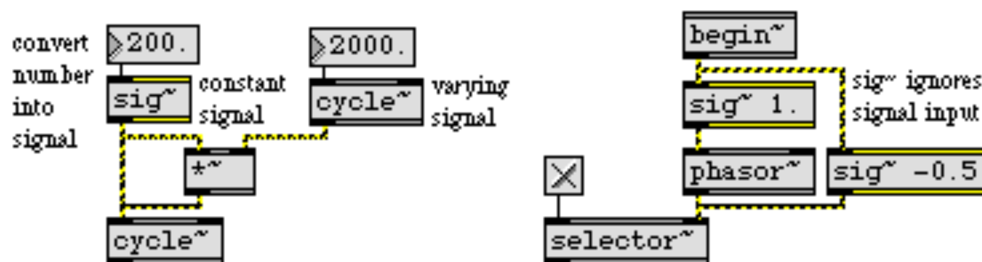
Arguments

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~	Linear ramp generator
Tutorial 4	Fundamentals: Routing signals

sinh~

Signal hyperbolic sine function

Input

signal Input to a hyperbolic sine function.

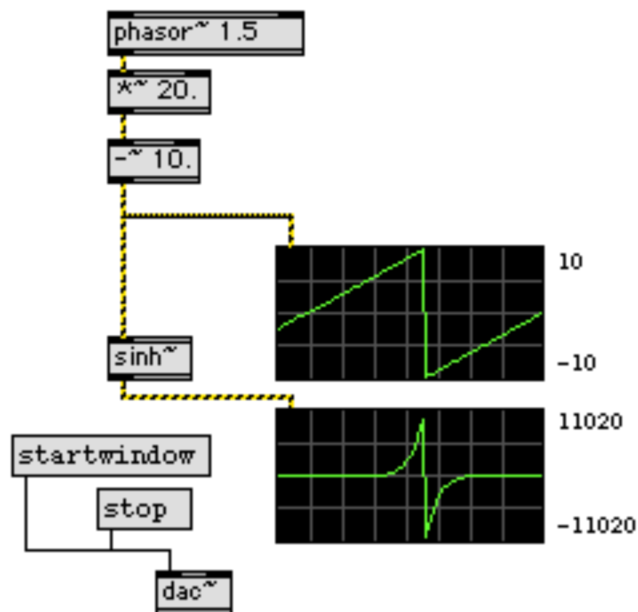
Arguments

None.

Output

signal The hyperbolic sine of the input.

Examples



sinh~ *can generate interesting oscillator-synced audio control signals*

sinh~

See Also

acos~	Signal arc-cosine function
acosh~	Signal hyperbolic arc-cosine function
asin~	Signal arc-sine function
asinh~	Signal hyperbolic arc-sine function
atan~	Signal arc-tangent function
atanh~	Signal hyperbolic arc-tangent function
atan2~	Signal arc-tangent function (two variables)
cos~	Signal cosine function (0-1 range)
cosh~	Signal hyperbolic cosine function
cosx~	Signal cosine function
sinx~	Signal sine function
tanh~	Signal hyperbolic tangent function
tanx~	Signal tangent function

sinx~

Input

signal Input to a sine function.

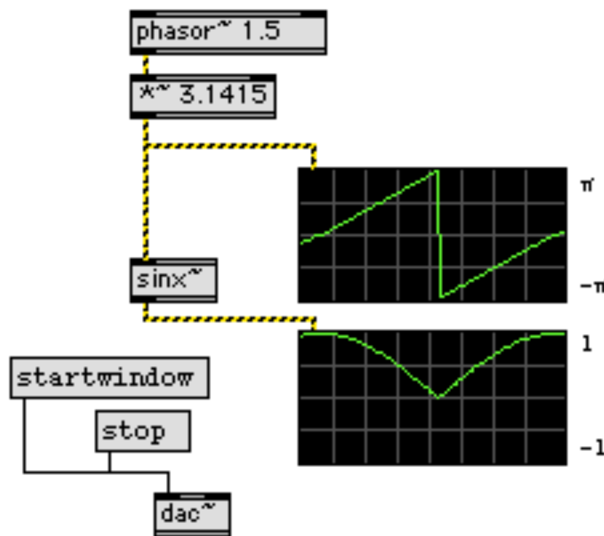
Arguments

None.

Output

signal The sine of the input.

Examples



sinx~ *can generate cycloids for audio control signals*

See Also

acos~	Signal arc-cosine function
acosh~	Signal hyperbolic arc-cosine function
asin~	Signal arc-sine function
asinh~	Signal hyperbolic arc-sine function
atan~	Signal arc-tangent function
atanh~	Signal hyperbolic arc-tangent function
atan2~	Signal arc-tangent function (two variables)
cos~	Signal cosine function (0-1 range)
cosh~	Signal hyperbolic cosine function
cosx~	Signal cosine function
sinh~	Signal hyperbolic sine function
tanh~	Signal hyperbolic tangent function
tanx~	Signal tangent function

Input

signal — A signal to be filtered. Whenever a new value is received, **slide~** filters the input signal logarithmically between changes in signal value. using the formula

$$y(n) = y(n-1) + ((x(n) - y(n-1))/slide).$$

A given sample output from **slide~** is equal to the last sample's value plus the difference between the last sample's value and the input divided by the slide value. Given a slide value of 1, the output will therefore always equal the input. Given a slide value of 10, the output will only change 1/10th as quickly as the input. This can be particularly useful for lowpass filtering or envelope following.

float — In middle inlet: Specifies the *slide up* value to be used when an incoming value is greater than the current value.

In right inlet: Specifies the *slide down* value to be used when an incoming value is less than the current value.

Arguments

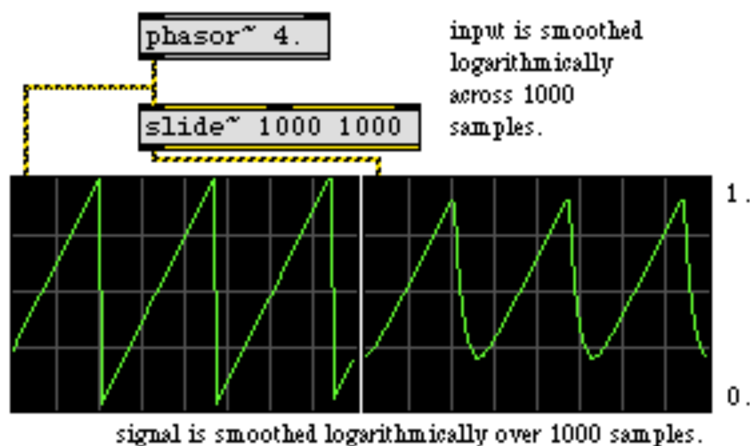
float — Optional. Specifies the *slide up* value. The default is 1.

float — Optional. A second argument specifies the *slide down* value. The default is 1.

Output

signal — The filtered signal.

Examples



slide~ *performs logarithmic smoothing of an input signal*

See Also

rampsmooth~	Smooth an incoming signal

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.

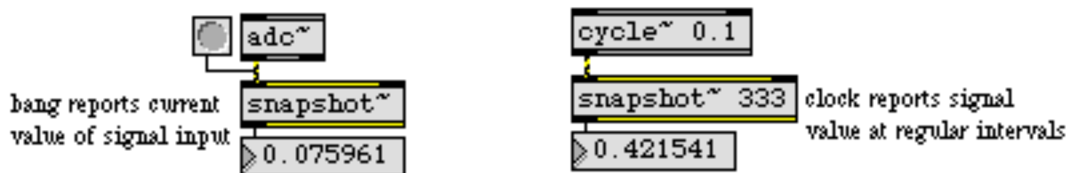
Arguments

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 23	Analysis: Viewing signal data

Input

signal In left inlet: A signal to be analyzed. The **spike~** object analyzes an incoming signal and reports the interval, in milliseconds, between transitions between zero and non-zero signal values. You can specify a *refractory period*, which defines how soon after detecting a transition the **spike~** object will report the next instance.

int or float In right inlet: Sets the refractory period, in milliseconds. When a signal transition is detected, this value sets the time, in milliseconds, during which no transitions are reported. After the refractory period has elapsed, the **spike~** object reports the next zero to non-zero signal transition. The default is 0.

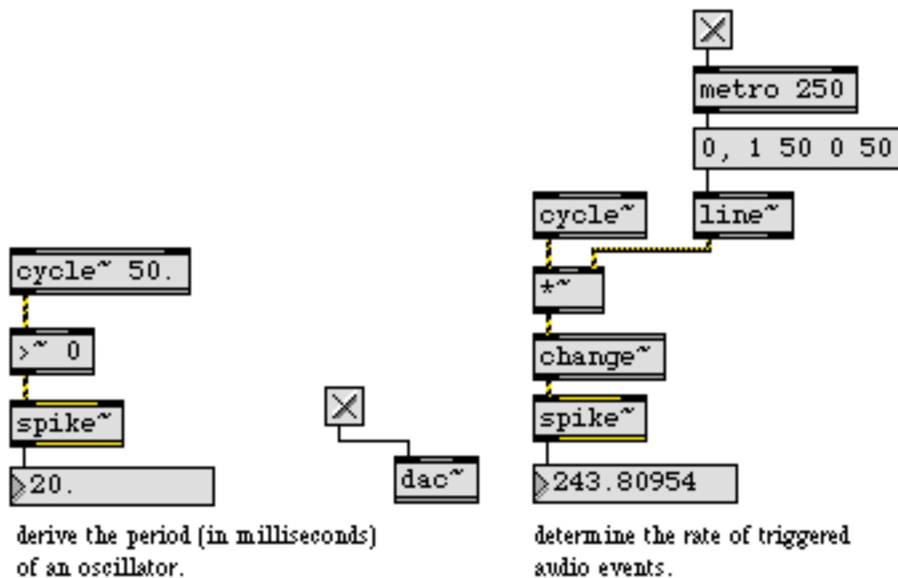
Arguments

int or float Optional. Sets the refractory period (see above).

Output

float The interval, in milliseconds, since the last zero to non-zero signal transition has occurred (which includes the refractory period, if one is set).

Examples



spike~ *reports how often a zero to non-zero transition occurs in its input signal*

See Also

change~	Report signal direction
edge~	Detect logical signal transitions
zerox~	Detect zero crossings

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.

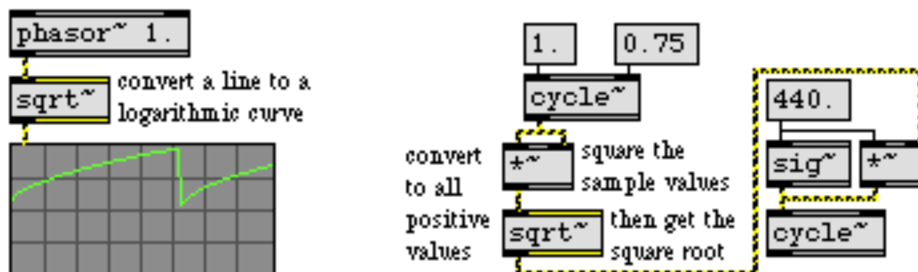
Arguments

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

Input

signal — In left inlet: Signals coming into the left inlet are stored in a record buffer, where they can be copied into a playback buffer and used as a playback source.

In middle inlet: Accepts a trigger signal, which can be specified to be positive or negative. When the signal changes polarity in the correct direction, samples recorded from the left inlet are copied to the playback buffer.

In right and successive inlets: A phase signal input in the range of 0-1 for each inlet controls the output speed of the playback buffer for that inlet. The number of phase inlets in a **stutter~** object is set using the fifth argument; the default is a single inlet. Specifying multiple phase inlets allows you to specify multiple playback points in the sampled buffer.

bang — In left inlet: A bang causes the last buffer of recorded samples to be copied to the playback buffer. You can use a bang instead of or in conjunction with the middle inlet trigger signal.

ampvar — The word ampvar, followed by a float, specifies a random amplitude variation in the output signal(s). The default is 0 (no variation).

dropout — The word dropout, followed by a float, determines the percentage chance of a playback signal dropping out (i.e. "gapping' or not playing). The default is 0 (no gapping).

int — In left inlet: Specifies the size (in samples) of the playback buffer. This can be any number up to the maximum memory determined by the first argument to **stutter~**.

maxsize — The word maxsize, followed by a number, sets the maximum buffer size, in samples.

polarity — The word polarity, followed by a 0 or 1, changes the trigger polarity of **stutter~** to negative or positive, respectively.

repeat — The word repeat, followed by a float, determines the percentage change of the record buffer not being copied to the playback buffer so that the previous playback buffer is repeated. The default is 0 (no repeat).

setbuf — The word setbuf, followed by arguments for a buffer name, a sample offset, and a channel, copies the specified samples to the named **buffer~** object. Note: **stutter~** always uses its internal buffer as the playback buffer; the copied samples can be sent to a named **buffer~** object for use in some other way, if desired. The time required to move the specified amount of memory to the buffer is *n/m*, there *n* is the number of samples being copied and *m* is the fourth argument to the **stutter~** object.

Arguments

int — Obligatory. The maximum buffer length, in samples. This determines the memory size of the record buffer. Parts of the record buffer are copied to the playback buffer when the object is triggered.

int — Obligatory. The initial buffer size, in samples, to copy from the record to the playback buffer upon receiving a trigger.

int — Obligatory. The polarity to use for accepting a trigger signal in the middle inlet. If the argument is greater than 0, **stutter~** accepts a positive trigger; otherwise **stutter~** accepts a negative trigger.

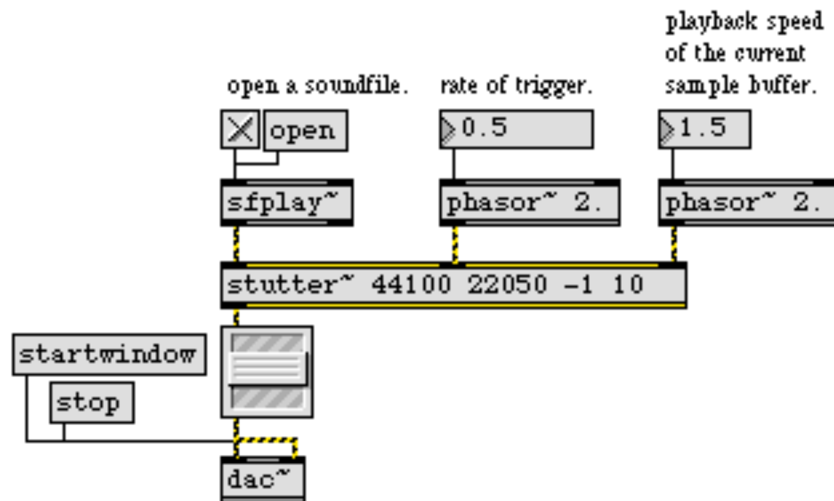
int — Obligatory. The number of samples which are copied from the record buffer to the playback buffer each iteration of the perform loop (the signal vector size). A larger value will decrease the **stutter~** object's memory requirements and increase the CPU requirements.

int — Optional. An optional fifth argument allows you to specify multiple independent signal outputs the **stutter~** object will use when playing back from the playback buffer. The default is 1, and the maximum is 30. The number of phase signal inputs to the **stutter~** object is also determined by this argument.

Output

signal — All outlets: The **stutter~** object's outlets produce a signal from the playback buffer, the location and speed of which is determined by the phase input for that playback outlet. The number of outlets is determined by the fifth argument to the **stutter~** object.

Examples



stutter~ *captures a new slice of incoming sound into an oscillating buffer whenever it receives a trigger*

See Also

buffer~	Store audio samples
phasor~	Sawtooth wave generator
record~	Record sound into a buffer

svf~

State-variable filter with simultaneous outputs

The **svf~** object is an implementation of a state-variable filter algorithm described in Hal Chamberlin's book, "*Musical Applications of Microprocessors*." A unique feature of this filter object is that it produces lowpass, highpass, bandpass, and bandreject (notch) output simultaneously—all four are available as outlets.

Input

signal — In left inlet: Signal to be filtered.

In middle inlet: Sets the filter center frequency in Hz.

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.

float — In middle and right inlets: A float can be sent in the two right inlets to change the center frequency and Q of the filter. By default, the center frequency is expressed in Hz, where the allowable range is from 0 to one fourth of the current sampling rate. For convenience, **svf~** has two additional input modes that use the more conventional input range, 0 - 1. (see the linear and radians messages). If a signal is connected to one of the inlets, a number received in that inlet is ignored. The values are sampled once every signal vector.

Hz — In either inlet: Sets the frequency input mode to Hz (the default).

linear — In any inlet: Sets the frequency input mode to linear (0 - 1). Linear mode is simply a scaled version of the standard Hz mode, except that values in the 0-1 range traverse the full frequency range.

radians — In any inlet: Sets the frequency input mode to radians (0 - 1). Radians mode lets you set the center frequency directly—while the input has the same range (0-1), the output has a curved frequency response that is closer to the exponential pitch scale of the human ear.

Arguments

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.

Hz — Optional. Sets the frequency input mode to Hz (the default mode - hence this is the same as providing no mode argument).

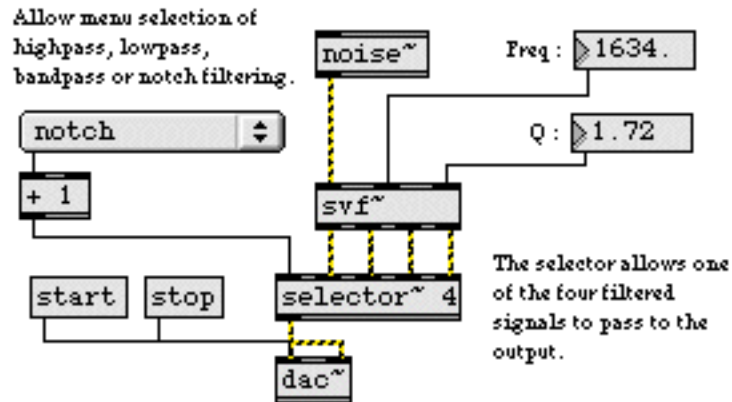
linear — Optional. Sets the frequency input mode to linear (0 - 1).

radians — Optional. Sets the frequency input mode to radians (0 - 1).

Output

signal The filtered input signal.

Examples



Four filter outputs are simultaneously available from the **svf~** *object*

See Also

biquad~	Two-pole, two-zero filter
onepole~	Single-pole lowpass filter

tanh~

Signal hyperbolic tangent function

Input

signal Input to a hyperbolic tangent function.

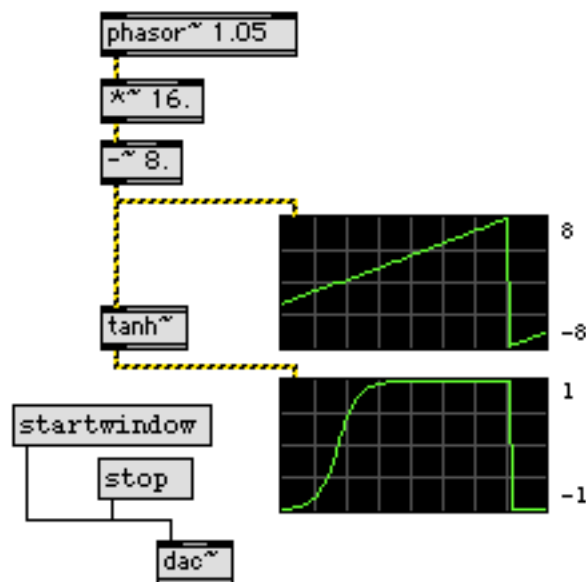
Arguments

None.

Output

signal The hyperbolic tangent of the input.

Examples



Use **tanh~** *to generate periodic control signals*

See Also

acos~	Signal arc-cosine function
acosh~	Signal hyperbolic arc-cosine function
asin~	Signal arc-sine function
asinh~	Signal hyperbolic arc-sine function
atan~	Signal arc-tangent function
atanh~	Signal hyperbolic arc-tangent function
atan2~	Signal arc-tangent function (two variables)
cos~	Signal cosine function (0-1 range)
cosx~	Signal cosine function
sinh~	Signal hyperbolic sine function
sinx~	Signal sine function
tanx~	Signal tangent function

tanx~

Input

signal — Input to a tangent function.

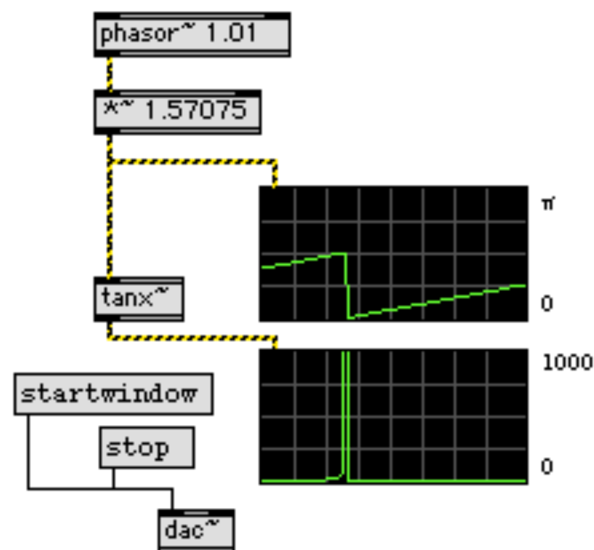
Arguments

None.

Output

signal — The tangent of the input.

Examples



Generate spikes (tangents increase exponentially as the input approaches π/2) using **tanx~**

See Also

acos~	Signal arc-cosine function
acosh~	Signal hyperbolic arc-cosine function
asin~	Signal arc-sine function
asinh~	Signal hyperbolic arc-sine function
atan~	Signal arc-tangent function
atanh~	Signal hyperbolic arc-tangent function
atan2~	Signal arc-tangent function (two variables)
cos~	Signal cosine function (0-1 range)
cosh~	Signal hyperbolic cosine function
cosx~	Signal cosine function
sinh~	Signal hyperbolic sine function
sinx~	Signal sine function
tanh~	Signal hyperbolic tangent 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.

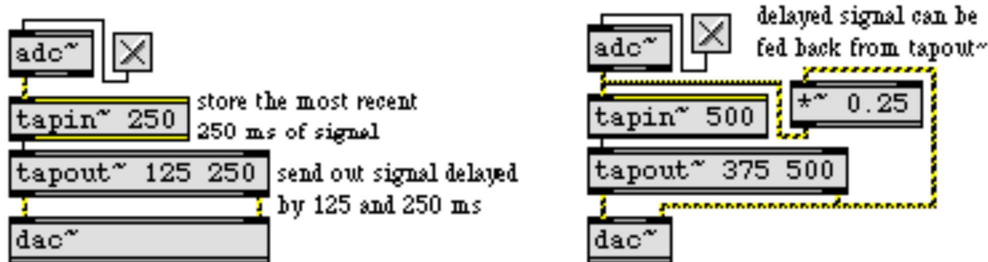
Arguments

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 27	Processing: Delay lines

tapout~

Output from a delay line

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.

Input

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.

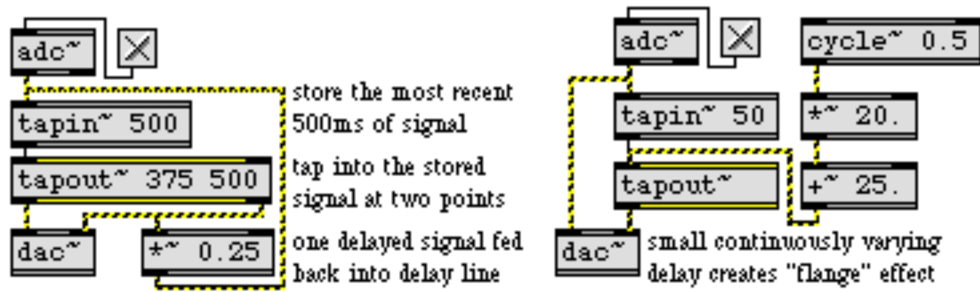
Arguments

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.

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 27	Processing: Delay lines

teeth~

Comb filter with feedforward and feedback delay control

Input

signal — In left inlet: Signal to be filtered. The **teeth~** object is a variant of **comb~**—a comb filter that mixes the current input sample with earlier input and/or output samples to accentuate and attenuate the input signal at regularly spaced frequency intervals. Unlike the **comb~** object, **teeth~** adds feedforward and feedback, which adds to the extremity of the effect.

In 2nd inlet: Feedforward—the delay, in milliseconds, before past samples of the *input* are added to the current input.

In 3rd inlet: Feedback—The delay, in milliseconds, before past samples of the *output* are added to the current input.

In 4th inlet: Gain coefficient for scaling the amount of the input sample to be sent to the output.

In 5th inlet: Gain coefficient for scaling the amount of feedforward to be sent to the output.

In right inlet: Gain coefficient for scaling the amount of feedback to be sent to the output.

float or int — The filter parameters 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 six parameters can be provided as a list in the left inlet. The first number in the list is the feedforward delay, the next number is the feedback delay, the third number is the Gain coefficient for the input sample, the fourth number is the feedforward gain coefficient, and the fifth number is the feedback gain coefficient. If a signal is connected to a given inlet, the coefficient supplied in the list for that inlet is ignored.

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

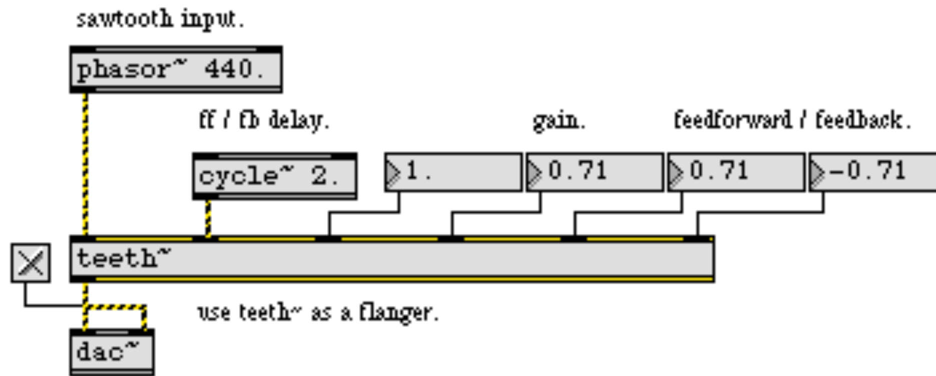
Arguments

float — Optional. Up to six numbers, to set the feedforward and feedback delays, the gain coefficient, and the feedforward and feedback gain coefficients. 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



teeth~ *does comb filtering on an input signal with variable feedforward and feedback delays*

See Also

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

thispoly~

The **thispoly~** object is placed *inside* a patcher loaded by the **poly~** object. It sends and receives messages from the **poly~** object that loads it.

Input

bang Reports the instance number of the patch. The first instance is reported as 1.

signal A signal input can be used to set the "busy" state of the patcher instance. When an incoming signal is non-zero, the busy state for the patcher instance is set to 1. When no signal is present, the busy state is set to 0.

int A value of 0 or 1 toggles the "busy" state off or on for the patcher instance. When "busy" (i.e., set to 1) the object will not receive messages generated by a note or midinote message to the left inlet of the parent **poly~** object.

mute The mute message toggles the DSP for the loaded instance of the patcher on (0) and off (1). This message can be combined with an int message which toggles the "busy" state of the patcher to create voices in a patcher which are only on while they play a "note".

Arguments

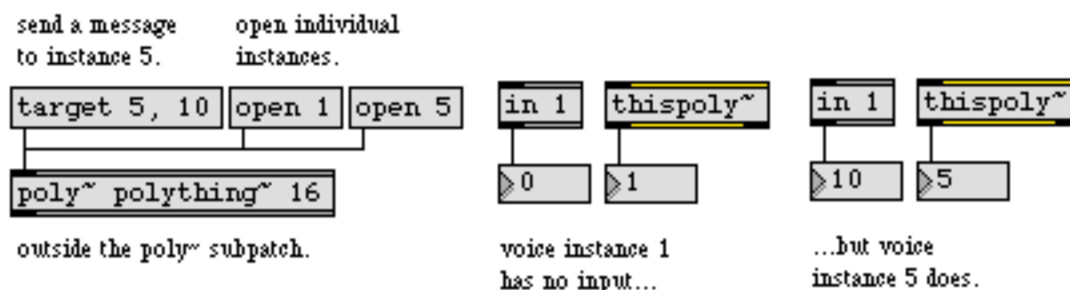
None.

Output

int Out left outlet: The instance number, starting at 1, reported when **thispoly~** receives the bang message. If the patcher containing **thispoly~** was not loaded within a **poly~** object, 0 is output.

int Out right outlet: If the loaded instance of the patcher is muted, a 1 is output. If the instance is not muted, a 0 is output.

Examples



thispoly~ *reports the instance number of its* **poly~** *subpatcher*

See Also

in	Message input for a patcher loaded by **poly~**
in~	Signal input for a patcher loaded by **poly~**
out	Message output for a patcher loaded by **poly~**
out~	Signal output for a patcher loaded by **poly~**
poly~	Polyphony/DSP manager for patchers
Tutorial 21	MIDI control: Using the **poly~** object

thresh~

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 signal objects to control period, pulse width, and phase. If a signal is also connected to the inlet, float and int messages 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 signal objects are connected to any of the **train~** object's inlets, the corresponding initial argument value is ignored.

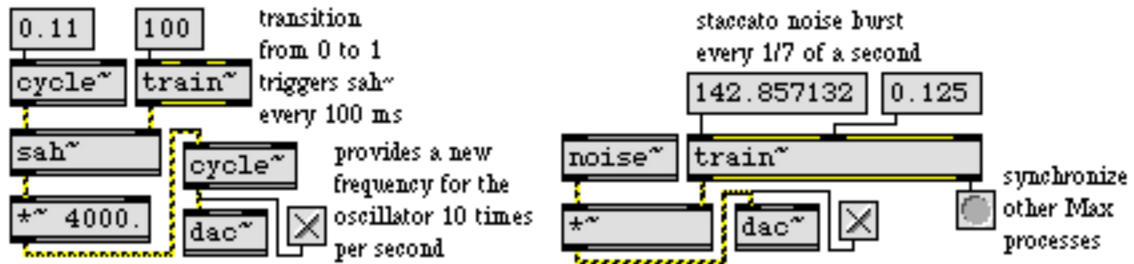
Output

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.

train~

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 wave generator

trapezoid~

Trapezoidal wavetable

Input

signal or float — In left inlet: Any float or signal or an input signal progressing from 0 to 1 is used to scan the **trapezoid~** object's wavetable. The output of a **phasor~** or some other audio signal can be used to control **trapezoid~** as an oscillator, treating the contents of the wavetable as a repeating waveform.

In middle inlet: The ramp up portion of the trapezoidal waveform, specified as a fraction of a cycle between 0 and 1.0. The default is .1.

In right inlet: The ramp up portion of the trapezoidal waveform, specified as a fraction of a cycle between 0 and 1.0. The default is .9.

lo — In left inlet: The word lo, followed by an optional number, sets the minimum value of **trapezoid~** for signal output. The default value is 0.

hi — In left inlet: The word hi, followed by an optional number, sets the maximum value of **trapezoid~** for signal output. The default value is 1.0.

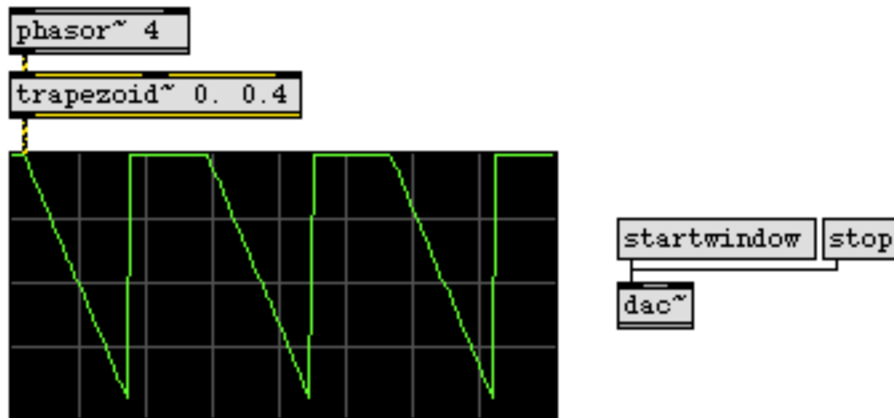
Arguments

float — Optional. Two floating-point values can be used to specify the ramp up and ramp down values. The arguments 0. 0. produce a ramp waveform, and .5 .5 produces a triangle waveform.

Output

signal — A signal which corresponds to the value referenced by the **trapezoid~** object's input signal. If the output of a **phasor~** or some other audio signal is used to scan the **trapezoid~** object, the output will be a periodic waveform.

Examples



trapezoid~ *generates a trapezoidal waveform that lets you specify the phase points at which it changes direction*

See Also

buffer~	Store audio samples
cos~	Cosine function
phasor~	Sawtooth wave generator
wave~	Variable-size wavetable
Tutorial 2	Fundamentals: Adjustable oscillator
Tutorial 3	Fundamentals: Wavetable oscillator

triangle~

Triangle/ramp wavetable

Input

signal or float — In left inlet: Any signal, float, or an input signal progressing from 0 to 1 is used to scan the **triangle~** object's wavetable. The output of a **phasor~** or some other audio signal can be used to control **triangle~** as an oscillator, treating the contents of the wavetable as a repeating waveform.

In right inlet: Peak value phase offset, expressed as a fraction of a cycle, from 0 to 1.0. The default is .5. Scanning through the **triangle~** object's wavetable using output of a **phasor~** with a phase offset value of 0 produces a ramp waveform, and a phase offset of 1.0 produces a sawtooth waveform.

lo — In left inlet: The word lo, followed by an optional number, sets the minimum value of **triangle~** for signal output. The default value is -1.0.

hi — In left inlet: The word hi, followed by an optional number, sets the maximum value of **triangle~** for signal output. The default value is 1.0.

Arguments

float — Optional. In right inlet: Peak value phase offset, expressed as a fraction of a cycle, from 0 to 1.0. The default is .5. A value of 0 produces a ramp waveform when the triangle~ object is driven by a **phasor~**, and a value of 1.0 produces a sawtooth waveform.

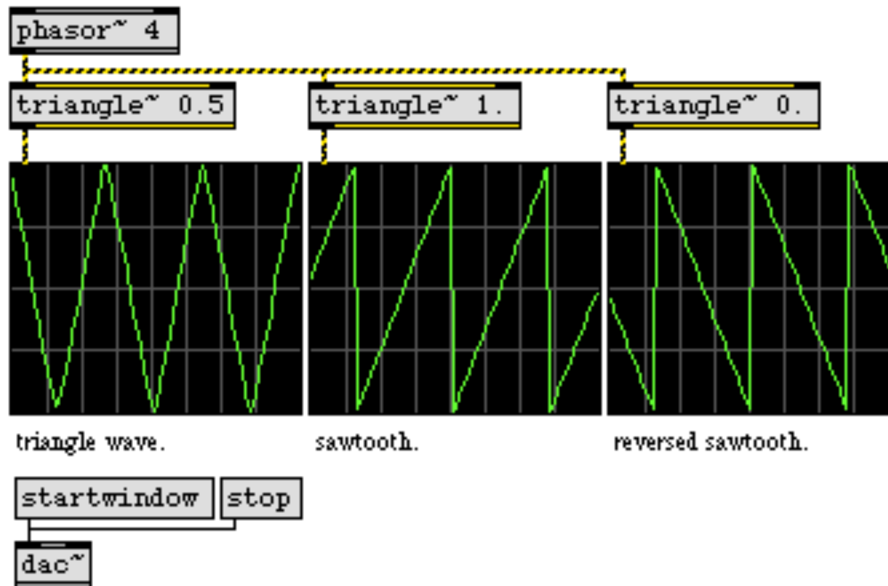
Output

signal — A signal which corresponds to the value referenced by the **triangle~** object's input signal. If the output of a **phasor~** or some other audio signal is used to scan the **triangle~** object, the output will be a periodic waveform.

triangle~

Triangle/ramp wavetable

Examples



triangle~ *lets you generate ramping waveforms with different reversal points*

See Also

buffer~	Store audio samples
cos~	Cosine function
phasor~	Sawtooth wave generator
trapezoid~	Trapezoidal wavetable
wave~	Variable-size wavetable
Tutorial 2	Fundamentals: Adjustable oscillator
Tutorial 3	Fundamentals: Wavetable oscillator

trunc~

Input

signal A signal whose values will be truncated. The **trunc~** object converts signals with fractional values to the nearest lower integer value (e.g., a value of 1.75 is truncated to 1.0, and -1.75 is truncated to -1.0). This object is simple but computationally expensive.

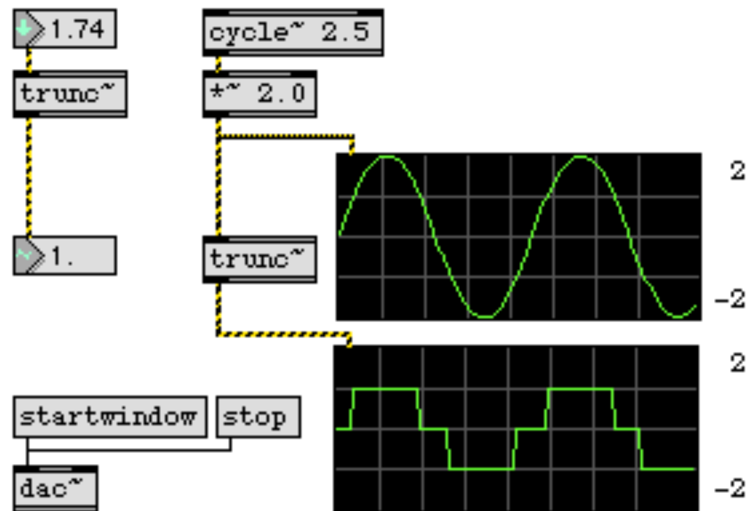
Arguments

None.

Output

signal The truncated input signal.

Examples



trunc~ *takes floating-point signals and truncated the fractional part*

clip~	Limit signal amplitude
round~	Round an input signal value

vectral~

Vector-based envelope follower

Input

signal — In left inlet: Accepts a sync signal for the output index of the vector. This is typically in the range of 0 to *n*-1 where *n* is the size of the vector.

In middle inlet: A sync signal received in the middle inlet is used to synchronize the input index of the vector being processed. The sync signal will typically be in the range 0 to n-1 where *n* is the size of the vector. If the range of the sync signal is different than the output index, the incoming vector will be "bin-shifted" by the difference between the two signals.

In right inlet: Signal data to be filtered. This will usually be frequency-domain information such as the output of an **fft~** or **fftin~** object.

rampsmooth — In left inlet: The word rampsmooth, followed by two ints, causes the vector to be smoothed in a linear fashion across successive frames. The arguments specify the number of frames to use to interpolate values in both directions. This is equivalent to the time-domain filtering done by the **rampsmooth~** object.

size — In left inlet: The word size, followed by a number, sets the vector size for the operation. The default is 512.

slide — In left inlet: The word slide, followed by two floats, causes **vectral~** to do logarithmic interpolation of successive vectors in a manner equivalent to the time-domain **slide~** object. The two arguments determine the denominator coefficient for the amount of the slide.

deltaclip — In left inlet: The word deltaclip, followed by two floats, limits the change in bins of successive vectors to the values given. This is equivalent to the time-domain **deltaclip~** object.

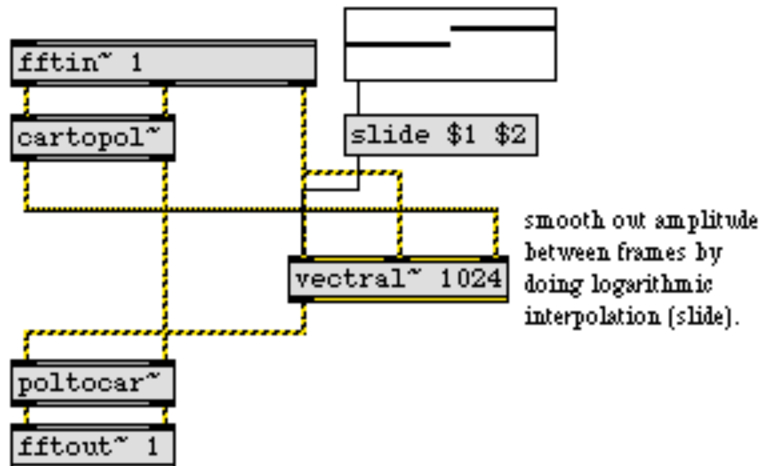
Arguments

int — Optional. The argument is the vector size for the operation. It defaults to 512, but should be set appropriately for the size of the vectors you feed into the **vectral~** object.

Output

signal — A smoothed version of the signal input into the right inlet, according to the parameters given to the **vectral~** object.

Examples



vectral~ *performs different types of smoothing between frames of vectored data (e.g., FFT signals)*

See Also

cartopol	Cartesian to Polar coordinate conversion
cartopol~	Signal Cartesian to Polar coordinate conversion
deltaclip~	Limit changes in signal amplitude
fft~	Fast Fourier transform
fftin~	Input for a patcher loaded by **pfft~**
fftinfo~	Report information about a patcher loaded by **pfft~**
fftout~	Output for a patcher loaded by **pfft~**
frameaccum~	Compute "running phase" of successive phase deviation frames
framedelta~	Compute phase deviation between successive FFT frames
ifft~	Inverse Fast Fourier transform
pfft~	Spectral processing manager for patchers
poltocar	Polar to Cartesian coordinate conversion
poltocar~	Signal Polar to Cartesian coordinate conversion
rampsmooth~	Smooth an incoming signal
slide~	Filter a signal logarithmically
Tutorial 26	Frequency Domain Signal Processing with **pfft~**

Note: The **vst~** object does not work with VST plug-ins created in Max/MSP.

Input

signal	Input to be processed by the plug-in. If the plug-in is an instrument plug-in, the input will be ignored.
int	In left inlet: Changes the effect program of the currently loaded plug-in. The first program is number 1.
float	Converted to int.
list	In left inlet: Changes a parameter value in the currently loaded plug-in. The first list element is the parameter number (starting at 1) and the second element is the parameter value. The second number should be a float between 0 and 1, where 0 is the minimum value of the parameter and 1 is the maximum.
any symbol	A symbol that names a plug-in parameter followed by a float between 0 and 1 set the value of the parameter.
bypass	The word disable, followed by a non-zero argument, stops any further processing by the currently loaded plug-in and copies the object's signal input to its signal output. bypass 0 enables processing for the plug-in.
disable	The word disable, followed by a non-zero argument, stops any further processing by the currently loaded plug-in and outputs a zero signal. disable 0 enables processing for the plug-in.
get	The word get, followed by a number argument, reports plug-in information out the plug-in's third outlet. If the number argument is between 1 and the number of parameter's of the currently loaded plug-in, the get message outputs the value of the numbered parameter (a number between 0 and 1). If the argument is 0 or negative, the get message produces the following information out the fourth outlet:

get -1	The plug-in's number of inputs
get -2	The plug-in's number of outputs
get -3	The plug-in's number of programs
get -4	The plug-in's number of parameters
get -5	Whether the plug-in's canMono flag is set. This indicates that the plug-in can be used in either a stereo or mono context
get -6	1 if the plug-in has its own edit window, 0 if it doesn't
get -7	1 if the plug-in is a synth plug-in, 0 if it isn't

midievent	The word midievent, followed by two to four numbers, sends a MIDI event to the plug-in. The first three number arguments are the bytes of the MIDI message.

The fourth, optional, argument is a detune parameter used for MIDI note messages. The value ranges from -63 to 64 cents, with 0 being the default.

mix — In left inlet: mix 1 turns mix mode on, in which the plug-in's output is added to the input. mix 0 turns mix mode off. When mix mode is off, the plug-in's output is not added to the input. Only the plug-in's output is sent to the vst~ object's signal outlets.

open — Opens the plug-in's edit window.

params — The word params causes a list of the plug-in's parameters to be sent out the fourth-from-right outlet.

pgmnames — The word pgmnames causes a list of the plug-in's current program names to be sent out the right outlet.

plug — In left inlet: The word plug with no arguments opens a standard open file dialog allowing you to choose a new VST plug-in to host. The word plug followed by a symbol argument searches for VST plug-in with the specified name in the Max search path as well as a folder called VstPlugIns inside the Max application folder. If a new plug-in is opened and found, the old plug-in (If any) is discarded and the new one loaded.

read — With no arguments, read opens a standard open file dialog prompting for a file of effect programs, either in bank or individual program format. read accepts an optional symbol argument where it looks for a VST plug-in bank or effect program file in the Max search path.

set — In left inlet: The word set, followed by a symbol, changes the name of the effect current program to the symbol.

settitle — In left inlet: The word settitle, followed by a symbol, changes the title displayed for the name of the plug-in's edit window.

wclose — Closes the plug-in's edit window.

write — With no arguments, write opens a standard Save As dialog box prompting you to choose the name and type of the effect program file (single program or bank). write accepts an optional symbol argument that specifies a full or partial destination pathname. An individual program file is written in this case.

writebank — With no arguments, writebank opens a standard Save As dialog box prompting you to choose the name of the effect program bank file. writebank accepts an optional symbol argument that specifies a full or partial destination pathname.

writepgm — With no arguments, writepgm opens a standard Save As dialog box prompting you to choose the name of the individual effect program file. writepgm accepts an optional symbol argument that specifies a full or partial destination pathname.

Arguments

int — Optional. If the first or first and second arguments are numbers, they set the number of audio inputs and outputs. If there is only one number, it sets the number of outlets. If there are two numbers, the first one sets the number of inlets and the second sets the number of outlets.

symbol — Optional. Sets the name of a VST plug-in file to load when the object is created. You can load a plug-in after the object is created (or replace the one currently in use) with the plug message.

symbol — Optional. After the plug-in name, a name containing preset effects for the plug-in can be specified. If found, it will be loaded after the plug-in has been loaded.

Output

signal — Out left outlet and other signal outlets as defined by the number of outputs argument: Audio output from the plug-in. The left outlet is the left channel (or channel 1).

symbol — Out fourth-from-right outlet: The plug-in's parameters are sent out as a series of symbols in response to the params message.

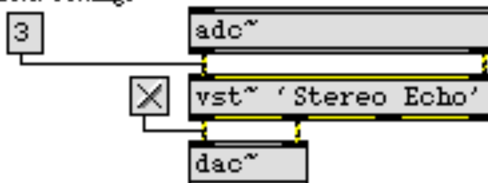
Note: Some plug-ins, especially those with their own editors, fail to name the parameters.

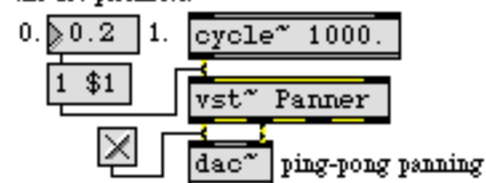
int or float — Out third-from-right outlet: Parameter values or plug-in informational values in response to the get message.

int — Out second-from-right outlet: Raw MIDI bytes received by the plug-in (but not any MIDI messages received using the midievent message).

symbol — Out right outlet: A series of symbols are sent out in response to the pgmnames message. If there are no program names, the message pgmnames: Default is output.

Examples





Process an audio signal with a VST plug-in

See Also

rewire~	Host ReWire devices

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 signal objects to control the start and end points of the waveform, provided a signal is not connected to the inlet that receives the number. The **wave~** object uses the **buffer~** sampling rate to determine loop points.

enable — In left inlet: The message enable 0 disables the object, causing it to ignore subsequent signal input(s). The word enable followed by any non-zero number enables the object once again.

interp — The word interp, followed by a number in the range 0-2, sets the wavetable interpolation mode. The interpolation modes are:

value	*description*
0	No interpolation. Wavetable interpolation is disabled using the interp 0 message.
1	High-quality linear interpolation (default)
2	Low-quality linear interpolation. This mode uses the interpolation method found in MSP *1.x* versions of the **wave~** object. While this mode is faster than mode 1, it cannot play **buffer~** objects of arbitrary length and produces more interpolation artifacts.

set — In left inlet: 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 signal objects 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~**.

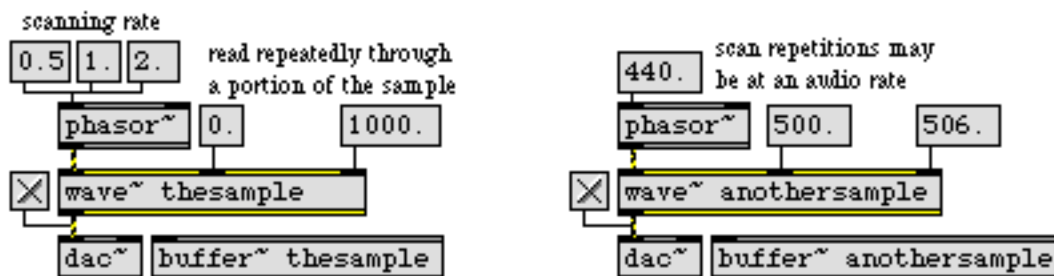
float or int Optional. After the **buffer~** name argument, you can type in values for the start and end points of the waveform, as millisecond offsets from the beginning of a **buffer~** object's sample memory. By default the start point is 0 and the end point is the end of the sample. If you want to 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. The **wave~** object uses the **buffer~** sampling rate to determine loop points. 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. An additional optional integer can used to specify the number of channels in the **buffer~** file.

int Optional. Sets the number of output channels, which determines the number of outlets that the **wave~** object will have. The maximum number of signal outputs is 4. If the **buffer~** object being played by **wave~** has more channels than the number of outputs of **wave~**, the extra channels are not played. If the **buffer~** object has fewer channels, the extra **wave~** signal outputs are 0.

Output

signal The portion of the **buffer~** specified by the **wave~** object's start and end points is scanned by signal values ranging from 0 to 1 in the **wave~** object's inlet, and the corresponding sample value from the **buffer~** is sent out the **wave~** object'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

2d.wave~	Two-dimensional wavetable
buffer~	Store audio samples
buffir~	Buffer-based FIR filter
groove~	Variable-rate looping sample playback
phasor~	Sawtooth wave generator
play~	Position-based sample playback
Tutorial 15	Sampling: Variable-length wavetable

Input

float — In left inlet: Sets the display start time in milliseconds. Changing this value will offset and/or zoom the view, so that the requested time in the **buffer~** sample data is aligned to the left edge of the display. The default is 0 (display starts at the beginning of the target **buffer~**).

In 2nd inlet: Sets the display length in milliseconds. The default is the length of the **buffer~**.

In 3rd inlet: Sets the start time of the selection range in milliseconds.

In 4th inlet: Sets the end time of the selection range in milliseconds.

list — In 5th inlet: The 5th inlet provides a link input, which allows any number of **waveform~** objects to share their start, length, select start, and select end values. Whenever any of these values changes, **waveform~** sends them all as a list out its right outlet. If this outlet is connected to the link input of another **waveform~** object, it will be updated as it receives the lists.

To complete the circuit, the second **waveform~** object's list output can be connected to the link input of the first. Then, changes in either one (via mouse clicks, etc.) will be reflected in the other. This is mainly useful when the **waveform~** objects are viewing different channels of the same **buffer~**. Any number of **waveform~** objects can be linked in this fashion, forming one long, circular chain of links. In this case **waveform~** will prevent feedback from occurring.

bpm — The word bpm, followed by one or two numbers, sets the reference tempo and number of beats per bar used by the **waveform~** display. The first argument sets the tempo in beats per minute. The default is 120. The second argument is optional, and specifies the number of beats per bar. The default is 4. The bpm message automatically changes the display time unit to bpm, as if you had sent the message unit bpm. Time values are shown in bars and beats, with subdivisions of the beat displayed in floating-point. The offset message can be useful to align the metric information with the contents of the target **buffer~**. **waveform~** can calculate a tempo based on the current selection with the setbpm message.

brgb — The word brgb, followed by three numbers in RGB format, sets the background color used to paint the entire object rectangle before the rest of the display components are drawn on top.

clipdraw — The word clipdraw, followed by a 1, will cause values being edited in draw mode to be clipped to the range of the display (as determined by the vzoom message). clipdraw 0 disables clipping, allowing values to be scaled freely beyond the range of the window. The default is 0, no clipping.

constrain — The constrain message should be followed by an int argument to toggle alternate behavior of the **waveform~** interface. The effect varies according to the current

mode (determined by the mode message), but it generally produces the same behavior that would be expected from holding down the shift key during mouse activity. For example, clicks in select mode are interpreted as incremental selections (setting only the nearest endpoint of the selection range); **buffer~** navigation in move mode is restricted to horizontal panning, with no zoom; selection length in loop mode is maintained regardless of vertical mouse movement. Obviously, this message is intended to implement this behavior where appropriate. Any non-zero int argument enables constrained interface activity. A zero, or no argument at all, disables constraint and returns to the default behavior.

crop — The crop message will trim the audio data in the target **buffer~** to the current selection. It resizes the **buffer~** to the selection length, copies the selected samples into it, and displays the result at default settings. The **buffer~** is erased, except for the selected range. This is a "destructive edit," and cannot be undone.

frgb — The word frgb, followed by three numbers in RGB format, sets the foreground color used to draw the **buffer~** data as a waveform graph.

grid — The word grid, followed by an int or float, specifies the spacing of the vertical grid lines, relative to the current time measurement unit. For example, when waveform~ is using milliseconds to display time values, the message, grid 1000 will cause grid lines to be drawn 1000 milliseconds apart in the waveform~ display. If labels are enabled, they will be drawn at the top of these grid lines. If tick marks are enabled, they will be drawn between these grid lines. An argument of 0 or no argument disables the grid lines.

labels — The word labels, followed by an int, enables (1) or disables (0) the numerical labels of time measurement across the top of the display. Any non-zero int causes the labels to be drawn. An argument of 0, or no argument, disables them.

mode — The word mode, followed by a symbol argument, determines how the **waveform~** object responds to mouse activity. Valid symbol arguments are none, select, loop, move, and draw.

none — Causes **waveform~** to enter a "display only" mode, in which clicking and dragging have no effect. For convenience, and to add custom interface behavior, mouse activity is still sent according to the mouseoutput mode. A mode message with no argument has the same effect as mode none.

select — Sets the default display mode of the **waveform~** object. In select mode, the cursor appears as an I-beam within the **waveform~** display area. You can click and drag with the mouse to select a range of values. Mouse activity will cause **waveform~** to generate update messages, according to the mouseoutput setting.

	loop	Sets an alternative loop selection style that uses vertical mouse movement to grow and shrink the selection length, while horizontal movement is mapped to position. This works well to control a groove~ object, as demonstrated in the **waveform~**.help file. When loop mode is selected, moving your cursor inside the display area changes its appearance to a double I-beam.
	move	Sets the move display mode of the **waveform~** object. This mode allows you to navigate the **waveform~** view. Vertical mouse movement lets you zoom in and out, while horizontal movement scrolls through the time range of the x-axis. Clicking on a point in the graph makes it the center reference point for the rest of the mouse event (until the mouse button is released). This lets you "grab" a spot and zoom in on it without having to constantly re-center the display.
	draw	Sets the draw display mode of the **waveform~** object. This mode allows you to edit the values of the target **buffer~**, using a pencil tool. Clicking and dragging in draw mode directly changes the **buffer~** samples, and can not be undone. Sample values are interpolated linearly as you drag, resulting in a continuous change, even if you are zoomed out too far to see the individual samples.

mouseoutput — The word mouseoutput, followed by a symbol argument, determines when selection start and end values are sent in response to mouse activity. Only the selection start and end (outlets 3 and 4) are affected. Mouse information is always sent from outlet 5, regardless of the mouseoutput mode. Valid symbol arguments are, none, down, up, downup, and continuous.

none	Selection start and end values are not sent in response to mouse activity. Sending the mouseoutput message with no argument has the same effect as the symbol (none).
down	Causes the current selection start and end values to be sent (from outlets 3 and 4) only when you click inside the **waveform~**.
up	Causes selection start and end to be sent only when you release the mouse button, after clicking inside the **waveform~**.
downup	Causes selection start and end to be sent both when you click inside the **waveform~**, and when the mouse button is released.
continuous	Causes selection start and end to be sent on click, release, and throughout the drag operation, whenever the values change.

normalize — The word normalize, followed by a float, will scale the sample values in the target **buffer~** so that the highest peak matches the value given by the argument. This can cause either amplification or attenuation of the audio, but in either case, every **buffer~** value is scaled, and this activity cannot be undone.

norulerclick — The word norulerclick, followed by an int, disables (1) or enables (0) clicking and dragging in the ruler area of the **waveform~** display. The default is enabled.

offset — The word offset, followed by a float, causes all labels and time measurement markings to be shifted by the specified number of milliseconds. Snap behavior is shifted as well. offset can be removed by sending the message offset 0., or the offset message with no argument.

rgb2 — The rgb2, followed by three numbers in RGB format, is applied to the selection rectangle, which identifies the selection range.

rgb3 — The word rgb3, followed by three numbers in RGB format, sets the frame color, used to draw the single-pixel frame around the object rectangle and the label area.

rgb4 — The word rgb4, followed by three numbers in RGB format, sets the label text color.

rgb5 — The word rgb5, followed by three numbers in RGB format, sets the label background color.

rgb6 — The word rgb6, followed by three numbers in RGB format, applies the color to tickmarks and measurement lines (if enabled).

rgb7 — The word rgb7, followed by three numbers in RGB format, sets the selection rectangle "OpColor". The selection rectangle is painted using rgb2 as a foreground color, as specified above. However, the transfer mode during this operation is set to "blend," with rgb7 as an OpColor. Experiment with different combinations of rgb2 and rgb7 to see how they affect color and opacity differently. Shades of gray can be useful here.

set — The word set, followed by a symbol or int which is the name of a **buffer~** object, links **waveform~** to the target **buffer~**, which is drawn with default display values. An optional int argument sets the channel offset, for viewing multi-channel **buffer~** objects. The name of the linked **buffer~** is not saved with the Max patch, so should be stored externally if necessary.

setbpm — The word setbpm, with no arguments, causes **waveform~** to calculate a tempo based on the current selection range. It automatically changes the display time unit to bpm, as if you had sent the message unit bpm. A tempo is selected such that the selection range constitutes a logical multiple or subdivision of the bar, preserving the current beats per bar value, and attempting to find the closest value to the current tempo that satisfies its criteria. When a suitable tempo is selected, the

offset parameter is adjusted so that the start time of the selection range falls exactly on a bar line.

The result is that the selection area will be framed precisely by a compatible tempo. One use of this technique is to quickly establish time labels and tick marks for a section of audio. After selecting a bar as accurately as possible, sending the setbpm message and turning on snap to label allows immediate quantization of the selection range to metric values.

If the target **buffer~** contains an audio segment that is already cropped to a logical number of beats or bars, the best technique is to select the entire range of the **buffer~** (with messages to the select start and end inlets), followed by the setbpm message. If the **buffer~** is cropped precisely, the resulting tempo overlay should be quite accurate, and immediately reveal the tempo along with metric information.

When a new tempo is calculated, it is sent from the rightmost outlet (the link outlet), to update any linked **waveform~** objects, and to be used in whatever manner required by the surrounding patch.

snap — The word snap, followed by a symbol argument, Sets the *snap mode* of the **waveform~** selection range. snap causes the start and end points of the selection to automatically move to specific points in the **buffer~**, defined by the snap mode. Possible arguments are none, grid, and zero.

none	Disables snap to allow free selection. This is the default. The snap message with no argument has the same effect.
grid	Specifies that the selection start and end points should snap to the vertical grid lines, as set by the grid message. Since the spacing of the grid lines is affected by the current time measurement unit, and by the offset value (if an offset has been specified), snap to grid will be affected by these parameters as well.
tick	Causes the selection start and end to snap to the tick divisions specified by the ticks message.
zero	Instead of snapping the selection to a uniform grid, this mode searches for zero-crossings of the **buffer~** data. These are defined as the points where a positive sample follows a negative sample, or vice-versa. This can be useful to find loop and edit points.

ticks — The word ticks, followed by a number, specifies the number of ticks that should be drawn between each grid line. The default is eight. An argument of 0, or no argument, disables the tick marks.

undo — This mode works for **waveform~** selection only. It causes the selection start and end points to revert to their immediately previous values. This is helpful when

you are making fine editing adjustments with the mouse and accidentally click in the wrong place, or otherwise cause the selection to change unintentionally. Repeated undo commands will toggle between the last two selection states.

unit — The word unit, followed by a symbol argument, sets the unit of time measurement used by the display. Valid symbol arguments are ms, samples, phase, and bpm.

ms — Sets the display unit to milliseconds. This is the default.

samples — Causes time values to be shown as sample positions in the target **buffer~**. The first sample is numbered 0, unless the display has been shifted by the offset message.

phase — Causes time to be displayed according to phase within the **buffer~**, normalized so that the 0 refers to the first sample, and 1 refers to the last. This type of measurement unit is especially relevant when working with objects that use 0-1 signal sync, such as **phasor~** and **wave~**.

bpm — Specifies beats per minute as the time reference unit, relative to a master tempo and number of beats per bar, both of which you can set with the bpm message. **waveform~** can also calculate a tempo that fits your current selection, via the setbpm message.

vlabels — The word vlabels, followed by an int, enables or disables the vertical axis labels along the rightmost edge of the **waveform~** display. Any non-zero number causes the labels to be drawn. An argument of 0, or no argument, disables them.

voffset — The word voffset, followed by a float, sets the vertical offset of the **waveform~** display. A value of 0. places the x-axis in the middle, which is the default.

vticks — The word vticks, followed by an int, enables or disables the vertical axis tick marks along the left and right edges of the **waveform~** display. Any non-zero int causes the tick marks to be drawn. An argument of 0, or no argument, disables them.

vzoom — The word vzoom, followed by a float, sets the vertical scaling of the **waveform~** display.

Inspector

The behavior of a **waveform~** object is displayed and can be edited using its Inspector. If you have enabled the floating inspector by choosing **Show Floating Inspector** from the Windows menu, selecting any **waveform~** object displays the **waveform~** Inspector in the floating window. Selecting an object and choosing **Get Info…** from the Object menu also displays the Inspector.

The **waveform~** Inspector lets you set the following attributes:

The *Snap* pull-down menu sets the snap mode of the **waveform~** selection range. snap causes the start and end points of the selection to automatically move to specific points in the **buffer~**, defined by the snap mode. Possible arguments are none (the default), grid, and zero. This corresponds to the snap message, above.

The *Grid* section of the Inspector is used to set an *offset*, in milliseconds. All labels and time measurement markings are shifted by the specified number of milliseconds (default 0). The *grid* option is used to specify the spacing of the vertical grid lines (default 1000.) relative to the current time measurement unit. A value of 0 disables the grid lines.

The *Tempo* section of the Inspector is used to set a tempo value for the display in BPM (beats per Minute). The default value is 120.n *offset*, in milliseconds. All labels and time measurement markings are shifted by the specified number of milliseconds (default 0). The *grid* option is used to specify the spacing of the vertical grid lines (default 1000.) relative to the current time measurement unit. A value of 0 disables the grid lines.

The *setbpm* button is used to automatically set the tempo for BPM display. this is similar to setting the PBM, except that **waveform~** object determines the new tempo. It finds the nearest tempo that "fits" the current selection - meaning that the selection length will be exactly one beat, one bar, or multiple (powers of 2) bars.

The *Ticks* section of the Inspector is used to display timing labels and markers (ticks) in the **waveform~** object display. Checking the *labels* checkbox turns on the numerical time display (default is on). Checking the *vlabels* checkbox turns on the vertical tick mark labels (default is off). Checking the *ticks* checkbox turns on the tick mark display beneath the time labels (default is on). Checking the *vticks* checkbox turns on the vertical tick marks (default is on).

The *Edit Mode* pull-down menu is used to set the display modes of the **waveform~** object used when selecting and editing. The default is select mode (see the mode message above).

Mouse Output pull-down menu determines when mouse activity triggers the display and selects output (see the output message above). The default mode is continuous.

The *Edit Mode* pull-down menu is used to set the display modes of the **waveform~** object. The default is select mode (see the mode message above).

The *Color* pull-down menu lets you use a swatch color picker or RGB values to specify the colors used for display by the **waveform~** object.

The *Revert* button undoes all changes you've made to an object's settings since you opened the Inspector. You can also revert to the state of an object before you opened the Inspector window by choosing **Undo Inspector Changes** from the Edit menu while the Inspector is open.

Arguments

None.

Output

float Out 1st outlet: The display start time of the waveform in milliseconds.

Out 2nd outlet: The display length in milliseconds.

Out 3rd outlet: The start time of the selection range in milliseconds.

Out 4th outlet: The end time of the selection range in milliseconds.

list Out 5th outlet: This is the mouse outlet, which sends information about mouse click/drag/release cycles that are initiated by clicking within the **waveform~** object. The list contains three numbers.

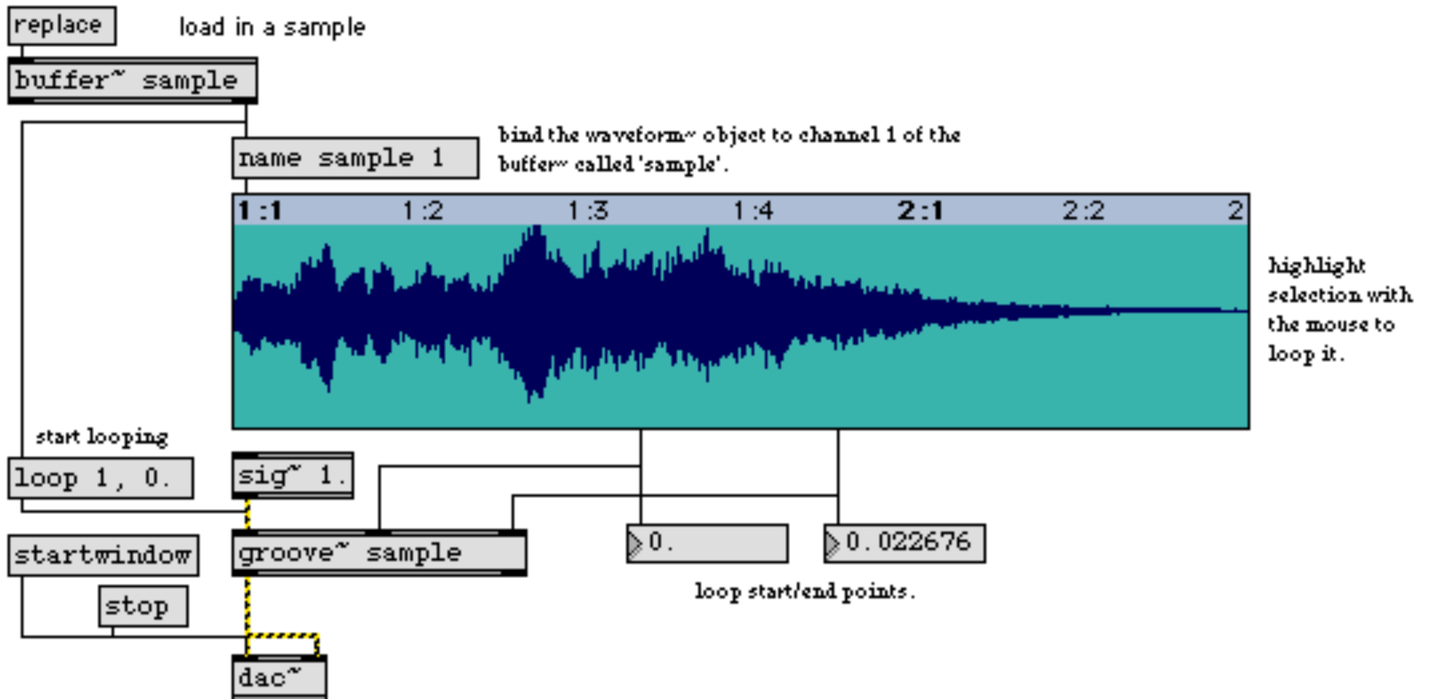
The first number is a float specifying the horizontal (x) position of the mouse, in 0-1 scale units relative to the **waveform~** object. x is always 0 at the left edge of the **waveform~**, and 1. at the right edge.

The second number in the list is the floating-point y value of the mouse, scaled to match the **buffer~** values. With the default vzoom = 1. and voffset = 0., the top of the **waveform~** gives a y value of 1, and the bottom is -1.

Finally, the third number in the list is an int that indicates which portion of the mouse event is currently taking place. On mouse down, or click, this value is 1. During the drag, it is 2, and on mouse up it is 3. These can be helpful when creating custom responses to mouse clicks. Note that a drag (2) message is sent immediately after the mouse down (1) message, whether the mouse has moved or not, to indicate that the drag segment has begun.

Out 6th outlet: **waveform~** outputs a list containing its display start time, display length, selection start time, and selection end time, whenever one of these values changes (by mouse activity, float input, etc.). See the link input information above for more information.

Examples



waveform~ *lets you view, select, and edit sample data from a* **buffer~** *object*

See Also

buffer~	Store audio samples
groove~	Variable-rate looping sample playback

zerox~

Input

signal In left inlet: A signal to be analyzed.

set In left inlet: The word set, followed by a floating-point number in the range 0.0-1.0, sets the volume of the click (impulse) sent out the right outlet. The default value is 1.0.

Arguments

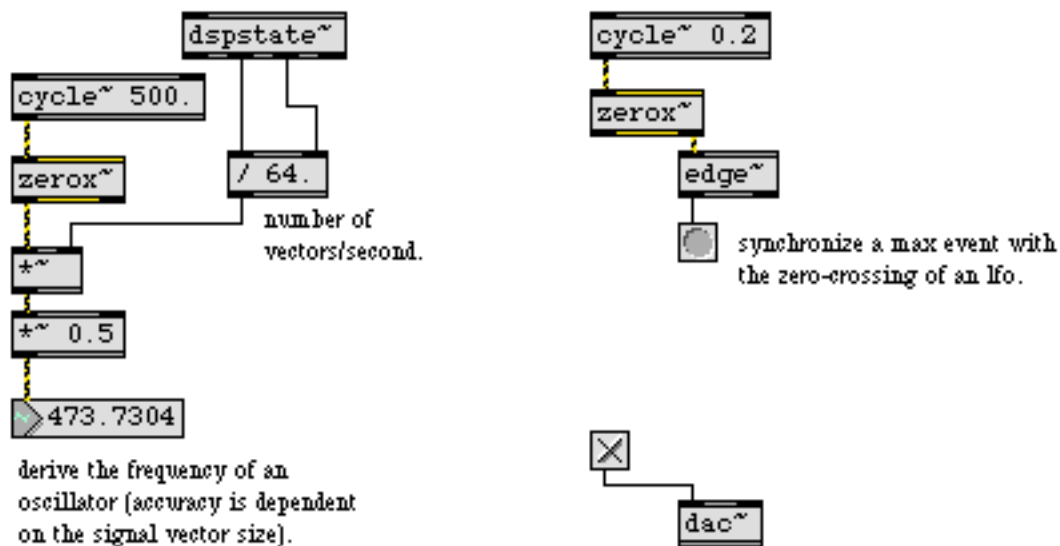
float Optional. Sets the output volume for the click sent out the right outlet. Volume values are in the range 0.0-1.0. The default value is 1.0.

Output

signal Out left outlet: A signal whose value corresponds to the number of zero crossings per signal vector which were detected during the period of the last signal vector.

Out right outlet: A click (impulse) whose volume is set by argument or by the set message is sent out the right outlet whenever a zero crossing is detected.

Examples



Use zerox~ *to count zero-crossings on an input signal*

See Also

change~	Report signal direction
edge~	Detect logical signal transitions
spike~	Report zero to non-zero signal transitions

zigzag~

The **zigzag~** object is similar to **line~**. While the **line~** object's stack-based implementation does not retain information after it has been output, **zigzag~** uses a linked list implementation. In addition to simply remembering the current "line", the **zigzag~** object lets you modify the list by inserting, deleting, or appending points.

Each element in the **zigzag~** object's linked list has a value (*y*), and a transition time value (*delta-x)*, which specifies the amount of time over which the transition from one value to another will occur. When **zigzag~** contains a list, this list can be triggered (the starting and ending points can be set and changed), traversed forwards or backwards at different speeds, and looped. The current position in the list can be jumped to, and also held.

Input

mode — The word mode, followed by a number in the range 0-3, specifies the way that the **zigzag~** object responds to messages and signal values. The modes of operation are summarized below:

mode 0 is the default mode of operation. When the **zigzag~** object receives a bang, it will jump to the start point (or end point if our direction is negative) and begin outputting values from there. The time value associated with this jump has its length defined by the bangdelta message. The default value for bangdelta is 0. If a signal is connected to the left inlet of the **zigzag~** object in this mode, the current index of the list is determined by the signal; any previously set speed, loopmode, start, and end messages are ignored.

mode 1 behavior for the **zigzag~** object is exactly the same as in mode 0 in terms of the effect of a bang. In mode 1, signal inputs are handled differently. If a signal is connected to the left inlet of the **zigzag~** object in mode 1, the input signal functions as a trigger signal; when the slope of the input signal changes from non-negative to negative, the object will be retriggered as though a bang were received.

mode 2 sets the **zigzag~** object to jump to the next index in the list (or the previous index, if the current direction is negative) and begin outputting values from there. The time value associated with this jump has its length defined by the bangdelta message. The default value for bangdelta is 0. If a signal is connected to the left inlet of the **zigzag~** object in mode 2, the input signal functions as a trigger signal; when the slope of the input signal changes from non-negative to negative, the object will be retriggered as though a bang were received.

bang — In left inlet: The **zigzag~** object responds to a bang message according to its mode of behavior, which is set using the mode message.

If the **zigzag~** object is set to *mode 0* or *mode 1*, a bang message will cause the **zigzag~** object to go to the start point (or end point if the direction is negative) and begin outputting values from there.

If the **zigzag~** object is set to *mode 2*, a bang message will cause the **zigzag~** object to jump to the next index in the list (or the previous index, if the current direction is negative) and begin outputting values from there.

signal — In left inlet: The **zigzag~** object responds to signal values according to its mode of behavior, which is set using the mode message.

If the **zigzag~** object is set to *mode 0*, the current index of the list is determined by the input signal value; any previously set speed, loopmode, start, and end messages will be ignored.

If a signal is connected to the left inlet of the **zigzag~** object in *mode 1*, the input signal functions as a trigger signal; when the slope of the input signal changes from non-negative to negative, the object will be retriggered as though a bang were received.

If a signal is connected to the left inlet of the **zigzag~** object in *mode 2*, the input signal functions as a trigger signal; when the slope of the input signal changes from non-negative to negative, the object will be retriggered as though a bang were received.

signal — In right inlet: A signal value specifies the rate at which the value and time pairs will be output. A value of 1.0 traverses the list forward at normal speed. A playback rate of -1 traverses the list backwards (i.e. in reverse). A signal value of .5 traverses the linked list at half the normal speed (effectively doubling the delay time values). The value of the input signal is sampled once per input vector. Therefore, any periodic frequency modulation with a period which is greater than the current sample rate/(2*vector_size) will alias.

float — In left inlet: Each element in the **zigzag~** object's linked list is a pair that consists of a *target value* (*y*), followed by a second number that specifies a total amount of time in milliseconds (*delta-x*). In that amount of time, numbers are output regularly in a line from the current index value to the target value. The list 0 0 3.5 500 10 1000 describes a line which begins with a value of 0 at time 0, rises to a value of 3.5 a half second later, and rises again to a value of 10 in 1 second.

int — In left inlet: Converted to float.

int or float — In right inlet: Specifies the rate at which the value and time pairs will be output. A value of 1.0 traverses the list forward at normal speed. A playback rate of -1 traverses the list backwards (i.e. in reverse). A value of .5 traverses the linked list at half the normal speed (effectively doubling the delay time values).

append — In left inlet: The word append, followed by an int which specifies a position (where 0 is the first element) and a list, will insert new event pair(s) after the index specified. The message append 0 5 500 will create a new second entry in the linked list (at the 0 index) with a value of 5 and a time of 500 milliseconds.

bangdelta — In left inlet: The word bangdelta, followed by a float or int, specifies the time over which the transition between values occurs when the **zigzag~** object receives a bang. The default is 0 (i.e., and immediate transition).

bound — In left inlet: The word bound, followed by two numbers which specify start and end indices (where 0 is the first element), sets the start and end points of the **zigzag~** object's linked list.

delete — In left inlet: The word delete, followed by an int which specifies a position (where 0 is the first element), will delete the value and time pair associated with that index from the list. A list can follow the delete message if you want to remove multiple event pairs from the list. The message delete 0 will remove the current first value and time pair from the list; the second value and time pair (i.e. the value and time pair at index 1) will now become the first values in the list.

dump — In left inlet: The word dump will cause a list consisting of all currently stored value and time pairs in the form

index target value delta-x

to be sent out the **zigzag~** object's 3rd outlet.

end — In left inlet: The word end, followed by an int which specifies a position (where 0 is the first element), sets the point at which the **zigzag~** object ceases its output when triggered by a bang.

insert — In left inlet: The word insert, followed by an int which specifies a position (where 0 is the first element) and a list, will insert new event pair(s) before the index specified. The message insert 0 5 500 will create a new first entry in the linked list (at the 0 index) with a value of 5 and a time of 500 milliseconds.

jump — In left inlet: The word jump, followed by an int which specifies a position (where 0 is the first element), skips to that point in the linked list and begins outputting value and time pairs from that point. An optional int can be used to specify the time, in milliseconds, over which the transition to the next value will occur (the default value is 0).

jumpend — In left inlet: The word jumpend causes the **zigzag~** object to immediately jump forward to the last value (*y*) on the linked list.

jumpstart — In left inlet: The word jumpstart causes the **zigzag~** object to immediately jump to the first value (*y*) on the linked list and then output the currently selected list or selected portion of the list.

loopmode — The word loopmode, followed by 1, turns on looping. loopmode 0 turns off looping. By default, looping is off. loopmode 2 turns on looping in "pendulum" mode, in which the value and time pairs are traversed in an alternating forward and reverse direction. By default, looping is off

next — In left inlet: The word next skips to the next value and time pair in the linked list. An optional int can be used to specify the time over which the transition to the next value will occur (the default value is 0).

prev — In left inlet: The word prev skips to the previous value and time pair in the linked list. An optional int can be used to specify the time over which the transition to the previous value will occur (the default value is 0).

print — In left inlet: The word print causes the current status and contents of the **zigzag~** object to be printed out in the Max window. The output consists of the current mode, loopmode, the start, end, and loop length of the current list, the pendulum state, and moving value of the object, followed by a listing of each index in the linked list, along with its *y* and *delta-x* values.

ramptime — In left inlet: The word ramptime, followed by a number, sets the ramp time, in milliseconds, at which the output signal will arrive at the target value.

setindex — In left inlet: The word setindex, followed by an int which specifies a position (where 0 is the first element) and a pair of floats, sets the *target value* (*y*) and transition time amounts (*delta-x*) for the specified position in the list.

skip — In left inlet: The word skip, followed by a positive or negative number, will skip the specified number of indices in the **zigzag~** object's linked list. Positive number values skip forward, and negative values skip backward. An optional int can be used to specify the time over which the transition to the next or previous value will occur (the default value is 0).

speed — In left inlet: The word speed, followed by a positive or negative floating-point number, specifies the rate at which the value and time pairs will be output. The message speed 1.0 traverses the list forward at normal speed, speed -1 traverses the list backwards, speed.5 traverses the linked list at half the normal speed (effectively doubling the delay time values).

start — In left inlet: The word start, followed by an int which specifies a position (where 0 is the first element), sets the point at which the **zigzag~** object begins its output when triggered by a bang.

Arguments

int or float — Optional. Sets an initial target value (*y*) for the **zigzag~** object.

Output

signal — Out 1st outlet: The current target value, or a ramp moving toward the target value according to the currently stored value and the target time.

Out 2nd outlet: The current delta-x value.

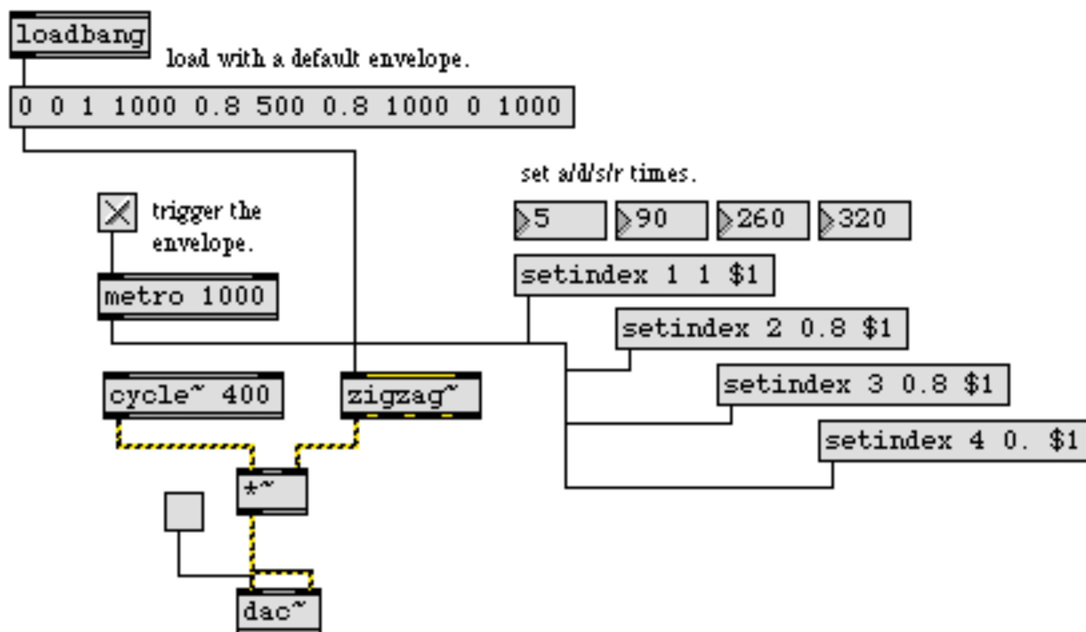
list — Out 3rd outlet: In response to the dump message, a list consisting of all currently stored value and time pairs in the form

index *target value* (*y*) *delta-x*

is output.

bang — Out right outlet: When looping, a bang message is sent out when the loop (retrigger) point is reached. A bang is also sent out when **zigzag~** has finished generating all of its ramps.

Examples



zigzag~ *can be used as a multi-purpose, editable ramp generator*

See Also

curve~	Exponential ramp generator
kink~	Distort a sawtooth waveform
line~	Linear ramp generator

The dsp Object

Controlling and Automating MSP

In order to provide low-level control over the MSP environment from within Max, a special object named dsp has been defined. This object is similar to the object **max** that can accept messages to change various options in the Max application environment. Sending a message to the dsp object is done by placing a semicolon in a message box, followed by dsp and then the message and arguments (if any). An example is shown below.

Turn the audio on or off without a **dac~** *or* **adc~** *object*

You need not connect the message box to anything, although you may want to connect something to the inlet of the message box to supply a message argument or trigger it from a loadbang to configure MSP signal processing parameters when your patcher file is opened.

Here is a list of messages the dsp object understands:

Message	*Parameters*
; dsp start	Start Audio
; dsp stop	Stop Audio
; dsp set N	N = 1, Start Audio;
	N = 0, Stop Audio
; dsp status	Open DSP Status Window
; dsp open	Open DSP Status Window
; dsp sr N	N = New Sampling Rate in Hz
; dsp iovs N	N = New I/O Vector Size
; dsp sigvs N	N = New Signal Vector Size
; dsp debug N	N = 1, Internal debugging on;
	N = 0, Internal debugging off
; dsp takeover N	N = 1, Scheduler in Audio Interrupt On;
	N = 0, Scheduler in Audio Interrupt Off
; dsp wclose	Close DSP Status window
; dsp inremap X Y	Maps physical device input channel *Y* to logical input *X*
; dsp outremap X Y	Maps logical output *X* to physical device output channel *Y*

; dsp setdriver D S	If *D* is a number starting at 0, a new audio driver is chosen based on its index into the currently generated menu of driverrs created by the adstatus driver object. If *D* is a symbol, a new driver is selected by name (if *D* names a valid driver). The second argument *S* is optional and names the "subdriver." For instance, with ASIO drivers, ASIO is the name of the driver and PCI-324 is an example of a subdriver name.
; dsp timecode N	N = 1 or 0 to start/stop timecode reading by the audio driver (only supported currently by ASIO 2 drivers).
; dsp optimize N	N = 1 or 0 to turn Altivec optimization on/off
; dsp cpulimit N	Sets a utilization limit for the CPU, above this limit, MSP will not process audio vectors until the utilization comes back down, causing a click. N is a number between 0 and 100. If N is 0 or 100, there is no limit checking.

Certain audio drivers can be controlled with the ; dsp driver message. Refer to the Audio Input and Output section for more information on drivers that support this capability.

Object Thesaurus

Objects listed by task keyword

Absolute value of all samples in a signal .... abs~
Access audio driver output channels .... adoutput~
Accumulator (signal) .... +=~
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~
Arc-cosine function for signals .... acos~
Arc-sine function for signals .... asin~
Arc-tangent function for signals .... atan~
Arc-tangent function for signals (two variables) .... atan2~
Arithmetic operators for signals acos~, acosh~, asin~, asinh~, atan~, atanh~, atan2~, cos~, cosh~, cosx~, sinh~, sinx~, tanh~, tanx~
Audio driver output channel access .... adoutput~
Audio driver settings, reporting and controlling .... adstatus
Average signal amplitude .... avg~
Backward sample playback .... groove~, play~
Bandpass filter .... noise~, pink~, rand~, reson~
Bit shifting for floating-point signals .... bitshift~
Bitwise "and" of floating-point signals .... bitand~
Bitwise "exclusive or" of floating-point signals .... bitxor~
Bitwise "or" of floating-point signals .... bitor~
Bitwise inversion of a floating-point signal .... bitnot~
buffer~ viewer and editor .... waveform~
Buffer-based FIR filter .... buffir~
Bypassing a signal .... gate~, mute~, pass~, selector~
Cartesian to Polar coordinate conversion (signal) .... cartopol~
Chorusing .... cycle~, tapout~
Clipping .... clip~, dac~, normalize~
Comb filter with feedforward and feedback delay control .... teeth~
Comb filter .... comb~
Compare two signals, output the maximum .... maximum~
Compare two signals, output the minimum .... minimum~
Comparing signals .... <~, ==~, >~, change~, meter~, scope~, snapshot~
Compute "running phase" of successive phase deviation frames .... frameaccum~
Compute phase deviation between successive FFT frames .... framedelta~
Compute the minimum and maximum values of a signal .... minmax~
Configure the behavior of a plug-in .... plugconfig
Constant signal value .... sig~
Control audio driver settings .... adstatus
Control function .... curve~, function, line~

Object Thesaurus

Objects listed by task keyword

Control **poly~** voice allocation and muting ... **thispoly~**
Convert Max messages to signals ... **curve~**, **line~**, **peek~**, **poke~**, **sig~**
Convert signals to Max messages ... **avg~**, **meter~**, **peek~**, **snapshot~**
Cosine function for signals (0-1 range) ... **cos~**
Cosine function for signals ... **cosx~**
Cosine wave ... **cos~**, **cycle~**
Create an impulse ... **click~**
DC offset ... **+~**, **-~**, **number~**, **sig~**
Define a plug-in parameter ... **pp**
Define a plug-in's audio inputs ... **plugin~**
Define a plug-in's audio outputs ... **plugout~**
Define a time-based plug-in parameter ... **pptime**
Define multiple plug-in parameters ... **plugmultiparam**
Define plug-in tempo and sync parameters ... **pptempo**
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~**
Divide two signals, output the remainder ... **%~**
Downsampling ... **avg~**, **number~**, **sah~**, **snapshot~**
Duty cycle of a pulse wave ... **<~**, **>~**, **train~**
Editing an audio sample ... **record~**, **peek~**, **poke~**
Envelope follower, vector-based ... **vectral~**
Envelope following ... **adc~**, **ezadc~**, **function**, **line~**
Envelope generator ... **curve~**, **function**, **line~**
Equalization ... **allpass~**, **biquad~**, **comb~**, **lores~**, **reson~**
Exponential curve function ... **curve~**, **gain~**, **linedrive**, **pow~**
Fast fixed filter bank ... **fffb~**
Feedback delayed signal ... **allpass~**, **biquad~**, **comb~**, **lores~**, **reson~**, **tapin~**, **tapout~**
Filter a signal logarithmically ... **slide~**
Filter ... **allpass~**, **biquad~**, **buffir~**, **comb~**, **lores~**, **noise~**, **pink~**, **reson~**, **vst~**
FIR filter, buffer-based ... **buffir~**
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~**
Generate parameter values from programs ... **plugmorph**
Global signal values ... **receive~**, **send~**
Graphical filter editor ... **filtergraph~**
Hertz equivalent of a MIDI key number ... **ftom**, **mtof**
Host ReWire devices ... **rewire~**
Host-synchronized sawtooth wave ... **plugphasor~**
Hyperbolic arc-cosine function for signals ... **acosh~**

Hyperbolic arc-sine function for signals ... asinh~
Hyperbolic arc-tangent function for signals ... atanh~
Hyperbolic cosine function for signals ... cosh~
Hyperbolic sine function for signals ... sinh~
Hyperbolic tangent function for signals ... tanh~
IIR filter ... allpass~, biquad~, comb~, lores~, reson~
Impulse generator ... click~
Input for a patcher loaded by **pfft~** ... fftin~
Input for a patcher loaded by **poly~** (message) ... in
Input for a patcher loaded by **poly~** (signal) ... in~
Input received in audio input jack ... adc~, ezadc~
Interpolating oscillator bank ... ioscbank~
Inverting signals ... *~, -~
Is greater than or equal to, comparison of two signals ... >=~
Is less than or equal to, comparison of two signals ... <=~
Level control ... *~, /~, gain~, normalize~
Level meter ... meter~, number~
Limit changes in signal amplitude ... deltaclip~
Limiter ... clip~, lookup~
Linked list function editor ... zigzag~
Logarithmic curve function ... curve~, gain~, linedrive, log~, pow~, sqrt~
Logical operations using signal values ... <~, ==~, >~, edge~
Lookup table ... buffer~, cycle~, function, index~, lookup~, peek~, wave~
Loop points in a sound file ... info~
Looping a sample ... 2d.wave~, groove~, info~, wave~
Lowpass filter ... lores~, noise~, pink~, rand~
Max messages converted to signals ... curve~, line~, peek~, poke~, sig~
Max messages derived from signals ... avg~, edge~, meter~, number~, peek~, snapshot~
Message input for a patcher loaded by **poly~** ... in
Message output for a patcher loaded by **poly~** ... out
MIDI control from MSP ... avg~, ftom, function, number~, snapshot~
MIDI control of MSP ... curve~, line~, mtof, sig~
Millisecond calculations ... mstosamps~, sampstoms~
Mixing signals ... +~
Modify plug-in parameter values ... plugmod
Multi-mode signal average ... average~
Multiple plug-in parameter definition ... plugmultiparam
Multiplying signals ... *~
Noise gate ... gate~
Noise ... noise~, pink~, rand~
Non-interpolating oscillator bank ... oscbank~
Normalization ... *~, /~, normalize~
Not equal to, comparison of two signals ... !=~
Numerical display of a signal ... capture~, number~, snapshot~
On/off audio switch ... adc~, dac~, dspstate~, ezadc~, ezdac~
Oscillator bank ... ioscbank~, oscbank~

Oscillator .... **2d.wave~**, **cycle~**, **phasor~**, **wave~**
Oscilloscope .... **scope~**
Output audio jack .... **dac~**, **ezdac~**
Output for a patcher loaded by **pfft~** .... **fftout~**
Output for a patcher loaded by **poly~** (message) .... **out**
Peak amplitude .... **meter~**
Periodic waves .... **2d.wave~**, **cycle~**, **phasor~**, **wave~**
Phase distortion synthesis .... **kink~**, **phasor~**
Phase modulation .... **phasor~**
Phase shifter .... **phaseshift~**
Pink noise generator .... **pink~**
Pitch bend .... **ftom**, **mtof**
Pitch-to-frequency conversion .... **mtof**
Playing audio .... **dac~**, **ezdac~**
Playing samples .... **2d.wave~**, **buffer~**, **groove~**, **index~**, **play~**, **sfplay~**, **wave~**
Plug-in audio inputs definition .... **plugin~**
Plug-in audio outputs definition .... **plugout~**
Plug-in development tools .... **plugconfig**, **plugin~**, **plugmod**, **plugmorph**, **plugmultiparam**, **plugout~**, **plugphasor~**, **plugreceive~**, **plugsend~**, **plugstore**, **plugsync~**, **pp**, **pptempo**, **pptime**
Plug-in in VST format used in MSP .... **vst~**
Plug-in parameter definition .... **pp**
Plug-in tempo and sync parameters definition .... **pptempo**
Polar to Cartesian coordinate conversion (signal) .... **poltocar~**
Polyphony management .... **in**, **in~**, **out**, **out~**, **poly~**, **thispoly~**
Polyphony/DSP manager for patchers .... **poly~**
Pulse wave .... **<~**, **>~**, **clip~**, **train~**
Ramp signal .... **curve~**, **line~**
Random signal values .... **noise~**, **pink~**, **rand~**
Receive audio from another plug-in .... **plugreceive~**
Recording audio samples .... **adc~**, **ezadc~**, **poke~**, **record~**, **sfrecord~**
Remainder (signal) .... **%~**
Repetition at sub-audio rates .... **cycle~**, **phasor~**, **train~**
Report and control audio driver settings .... **adstatus**
Report host synchronization information .... **plugsync~**
Report information about for a patcher loaded by **pfft~** .... **fftinfo~**
Report intervals of zero to non-zero transitions .... **spike~**
Report milliseconds of audio processed .... **dsptime~**
Resonant filter .... **allpass~**, **biquad~**, **comb~**, **lores~**, **reson~**
Reverberation .... **allpass~**, **comb~**, **tapin~**, **tapout~**
Reversed sample playback .... **groove~**, **play~**
ReWire device hosting .... **rewire~**
Ring modulation .... ***~**
Sample and hold .... **sah~**
Sample index in a buffer .... **count~**, **index~**
Sample playback .... **2d.wave~**, **buffer~**, **groove~**, **index~**, **play~**, **sfplay~**, **wave~**

Sample storage .... **buffer~**, **record~**, **sfrecord~**
Sampling rate .... **adc~**, **buffer~**, **count~**, **dac~**, **dspstate~**, **mstosamps~**, **sampstoms~**
Sawtooth oscillator .... **phasor~**
See the maximum amplitude of a signal .... **peakamp~**
Send audio to another plug-in .... **plugsend~**
Signal accumulator (signal) .... **+=~**
Signal arithmetic operators .... **acos~**, **acosh~**, **asin~**, **asinh~**, **atan~**, **atanh~**, **atan2~**, **cos~**, **cosh~**, **cosx~**, **sinh~**, **sinx~**, **tanh~**, **tanx~**
Signal capture and granular oscillator .... **stutter~**
Signal comparison, output the maximum .... **maximum~**
Signal comparison, output the minimum .... **minimum~**
Signal division (inlets reversed) .... **!/~**
Signal folding, variable range .... **pong~**
Signal input for a patcher loaded by **poly~** .... **in~**
Signal mixing matrix .... **matrix~**
Signal output for a patcher loaded by **poly~** .... **out~**
Signal quality reducer .... **degrade~**
Signal remainder .... **%~**
Signal routing matrix .... **matrix~**
Signal subtraction (inlets reversed) .... **!-~**
Signal tangent function (signal) .... **tanx~**
Sine function for signals .... **sinx~**
Sine wave .... **cos~**, **cycle~**
Single-pole lowpass filter .... **onepole~**
Smooth an incoming signal .... **rampsmooth~**
Soft-clipping signal distortion .... **overdrive~**
Sound Designer II saving and playing (Macintosh only) .... **buffer~**, **info~**, **sfplay~**, **sfrecord~**
Spectral domain processing. **cartopol~**, **fftin~**, **fftinfo~**, **fftout~**, **frameaccum~**, **framedelta~**, **pfft~**, **phasewrap~**, **poltocar~**, **vectral~**
Spectral-processing manager for patchers .... **pfft~**
Spectrum measurement .... **fft~**, **ifft~**
Start and end point of a sample .... **2d.wave~**, **groove~**, **index~**, **play~**, **wave~**
State-variable filter with simultaneous outputs .... **svf~**
Store multiple plug-in parameter values .... **plugstore**
Subpatch control .... **mute~**, **receive~**, **send~**
Subtractive synthesis .... **allpass~**, **biquad~**, **comb~**, **lores~**, **noise~**, **pink~**, **rand~**, **reson~**
Switching signal flow on and off .... **gate~**, **mute~**, **pass~**, **selector~**
Table lookup .... **buffer~**, **cycle~**, **function**, **index~**, **lookup~**, **peek~**, **wave~**
Tangent function for signals .... **tanx~**
Text file of signal samples .... **capture~**
Time-based plug-in parameter definition .... **pptime**
Transfer function .... **cycle~**, **lookup~**
Transient detector .... **zerox~**
Trapezoidal wavetable .... **trapezoid~**
Triangle/ramp wavetable .... **triangle~**
Triggering a Max message with an audio signal .... **edge~**, **thresh~**

Trigonometric operators for signals..... acos~, acosh~, asin~, asinh~, atan~, atanh~, atan2~, cos~, cosh~, cosx~, sinh~, sinx~, tanh~, tanx~
Truncate the fractional part of a signal.................... trunc~
Two-dimensional wavetable.................... 2d.wave~
Variable range signal folding.................... pong~
Varispeed sample playback.................... groove~, play~
Vector size.................... adc~, dac~, dspstate~
Vector-based envelope follower.................... vectral~
Velocity (MIDI) control of a signal.................... curve~, gain~, line~, sig~
View a signal.................... buffer~, capture~, number~, scope~, snapshot~
Waveshaping.................... lookup~
Wavetable synthesis.................... ,2d.wave~ buffer~, cycle~, wave~
Wavetables.................... trapezoid~, triangle~
White noise.................... noise~
Windowing a portion of a signal.................... index~, cycle~, gate~, lookup~, wave~
Wrap a signal between -π and π.................... phasewrap~
Zero-cross counter.................... zerox~

Index

Symbols
220, 221
!/~ 210
!=~ 211
!-~ 209
%~ 213
*~ 48, 214
+=~ 217
+~ 216
/~ 218
==~ 222
>=~ 225
>~ 224
-~ 215

Numerics
2d.wave~ 226

A
abs~ 229
absolute value 229
absorption of sound waves 192
access the hard disk 117
acos~ 230
adc~ 104, 232
adding signals together 56, 216
Additive synthesis 81
additive synthesis 20, 81
Adjustable oscillator 48
adoutput~ 234
adstatus 235
AIFF 265
aliasing 16, 69, 205
allpass~ 241
amplification 214, 333, 374
amplitude 9, 157
amplitude adjustment 48
amplitude envelope 13, 78, 82, 121, 352
Amplitude modulation 89
amplitude modulation 85, 89, 161
analog-to-digital conversion 15, 104, 232, 302
ASCII 119
asin~ 243, 244
ASIO 29
ASIO drivers, controlling with messages 40
atan~ 245
atan2~ 247
atanh~ 246
AtodB subpatch 63
attack, velocity control of 133
audio driver selection 29
audio driver settings override 30
audio input 232, 302
audio output 292, 304
audio processing off for some objects 70, 251, 336, 467
audio sampling rate, setting 30
average~ 249
avg~ 250

B
balance between stereo channels 150
band-limited noise 450
band-limited pulse 205
bandpass filter 456, 500
beats 60, 164
begin~ 70, 251
bell-like tone 83
biquad~ 252
bitand~ 254
bitnot~ 256
bitor~ 258
bitshift~ 260
bitwise and 254, 256
bitwise operators
  & 254
  bitnot~ 256
bitwise or 258, 260, 262
bitxor~ 262
bold type, displaying numbers in 377
buffer~ 53, 105, 264
buffir~ 269

C
capture~ 162, 271
carrier oscillator 86
cartopol~ 273
change~ 275
Chorus 200
chorus 200
click~ 276

Index

client, ReWire 458
clip~ 277
clipping 19, 48
clock source for audio hardware 30, 40
Comb filter 203
comb filter 191, 203, 278, 316, 507
comb~ 203, 278
comparing signal values 211, 220, 221, 222, 224, 225, 364, 368, 511
complex tone 10, 81
composite instrument sound 57
control rate 23
convolution 85
cos~ 280
cosh~ 231, 282
cosine wave 44, 280, 284, 290
cosx~ 284
count~ 106, 286
CPU limit option 33
CPU utilization indicator 30
critical band 87
crossfade 57
    constant intensity 153
    linear 152
    speaker-to-speaker 154
Csound 6
cue sample for playback 118
current file for playback 118
curve~ 288
cycle~ 44, 290

D

dac~ 292
dBtoA subpatch 127
DC offset 90, 215, 216, 375, 487
decibels 14, 62, 127, 334
default values 51
degrade~ 294
delay 189, 295
Delay line 189
delay line 295, 504, 505
delay line with feedback 193, 201, 203
Delay lines with feedback 192
delay time modulation 198
delay~ 295
delta~ 296, 326
deltaclip~ 297
difference frequency 60, 87, 164
digital audio overview 8
digital-to-analog converter 15, 44, 292, 304
diminuendo 79
disable audio of a subpatch 73
disk, soundfiles on 117
display signal 464
display signal amplitude 361, 365, 493
display signal as text 271
display signal graphically 163
display the value of a signal 157, 375
divide one signal by another 210, 218
Dodge, Charles 176
Doppler effect 197
downsamp~ 298
DSP Status window 28
dspstate~ 163, 299
dsptime~ 300
duty cycle 512

E

echo 189
edge~ 301
envelope 55
envelope generator 82, 249, 269, 294, 328, 348, 352, 369, 381, 388, 394, 396, 429, 434, 449, 451, 491, 497, 519, 528, 538
equal to comparison 211, 222
equalization 241, 252, 278, 316, 359, 379, 456, 500, 507
exponent in a power function 124
exponential curve 127, 128, 134, 288, 334, 354
ezadc~ 104, 302
ezdac~ 53, 304

F

fade volume in or out 51
feedback in a delay line 193, 201, 203
fffb~ 306
fft~ 166, 308
fftin~ 310
fftinfo~ 312
fftout~ 314
file search path of Max 265, 473, 475, 479

Index

file, record AIFF 117, 485
filter
  allpass 241
  comb 278, 316, 507
  lowpass 359
  resonant bandpass 456, 500
  two-pole two-zero 252, 379
filtergraph~ 316
Flange 196
flange 278, 316, 507
flanging 198
float-to-signal conversion 375, 487
FM 93, 95
foldover 16, 69, 205
Fourier synthesis 340
Fourier transform 12, 166, 308
frameaccum~ 325
framedelta~ 326
Freqeuency modulation 95
frequency 9, 45
frequency domain 85, 166
frequency modulation 93, 95
frequency-to-MIDI conversion 327
ftom 327
function 328
function object 82

G
G4 vector optimization 33
gain~ 206, 333
gate~ 60, 336
greater than comparison 224, 225, 364
groove~ 109, 121, 137, 338

H
hard disk, soundfiles on 117
harmonically related sinusoids 11, 74
harmonicity ratio 95
hertz 9

I
I/O mappings in MSP 32
ifft~ 167, 340
in 342
in~ 343
index~ 106, 344
info~ 110, 346
input 232, 302
input source 104
interference between waves 60, 164
interpolation 45, 54, 107, 160, 290, 332, 352, 375
inverse fast Fourier transform 167, 340
ioscbank~ 348

J
Jerse, Thomas 176

K
key region 137
kink~ 350

L
LED display 157
less than comparison 220, 221, 368
level meter 361, 365
level of a signal 48
LFO 128
limiting amplitude of a signal 277, 297, 374
line segment function 54
line~ 49, 352
linear crossfade 152
linear mapping 126
linedrive 354
localization 150
log~ 356
logarithmic curve 288, 334, 354, 356
logarithmic scale 14, 62
logical I/O channels 32, 33
logical signal transitions 301
lookup table 99, 114, 357, 389
lookup~ 99, 357
loop an audio sample 109, 338
lores~ 359
loudness 14, 127
low-frequency oscillator 128
lowpass filter 359
lowpass filtered noise 450

M
map subpatch 127
mapping a range of numbers 126

Index

Mapping MIDI to MSP 125
masking 254, 256
matrix~ 361
Max messages 46
maximum~ 364
meter~ 157, 365
metronome 512
MIDI 6, 125, 130
MIDI note value 327
MIDI panning 150
MIDI-to-amplitude conversion 198, 205, 333
MIDI-to-frequency conversion 132, 371
millisecond scheduler of Max 23, 43
minimum~ 368
minmax~ 369
mixer, ReWire 458
mixing 56
mixing signals 216
modulation
  amplitude 89
  delay time 198
  frequency 93, 95
  ring 85
modulation index 95
modulation wheel 126, 130
modulator 86
modulo 213
MP3 file conversion for buffer~ 264
MPG3 file conversion for buffer~ 264
MSP audio I/O overview 28
MSP overview 22
mstosamps~ 370
mtof 132, 371
multiply one signal by another 85, 214
mute audio of a subpatch 72, 372, 387
mute~ 72, 372

N

noise 13, 56, 201, 450
noise~ 56, 373
non real-time mode 29
non-real time and MSP 41
normalize~ 194, 374
number~ 157, 375
number-to-signal conversion 375, 487
Nyquist rate 16, 69, 114, 205

O

on and off, turning audio 232, 292, 302, 304
onepole~ 379
open and close a subpatch window 509
oscbank~ 381
oscillator 45, 290
Oscilloscope 163
oscilloscope 163, 464
out 383
out~ 385
output 292, 304
overdrive, turning off and on 32
overdrive~ 386

P

Panning 150
panning 150
partial 11, 81
pass~ 387
Patcher, audio on in one 68
pcontrol to mute a subpatch 73
peak amplitude 9, 160, 194, 361, 365
peakamp~ 388
peek~ 389
period of a wave 9
pfft~ 391
phase distortion synthesis 350
phase modulation 350, 397
phase offset 64
phaseshift~ 394
phasewrap~ 396
phasor~ 55, 397
pink noise 398
pink~ 398
pitch bend 128, 130
pitch-to-frequency conversion 122, 128, 371
play audio sample 106, 109, 234, 338, 344, 399
play audio sample as waveform 226, 525
play~ 107, 399
Playback with loops 109
plugconfig 401
plugin~ 408
plugmidiin 409

Index

plugmidiout 410
plugmod 411
plugmorph 413
plugmultiparam 416
plugout~ 418
plugphasor~ 419
plugreceive~ 420
plugsend~ 421
plugstore 422
plugsync~ 423
poke~ 425
poltocar~ 427
poly~ 143, 429
polyphony 130, 137, 143
pong~ 434
pow~ 124, 436
PowerPC 25
pp 437
pp, color messages 438, 447
pp, hidden 438, 447
pp, text 437
pptempo 441
pptime 445
precision of floating point numbers 77
prioritize MIDI I/O over audio I/O 30
pulse train 512
pulse width 512

Q

Q of a filter 359, 456, 500
QuickTime 264
QuickTime file conversion for buffer~ 264

R

RAM 117
rampsmooth~ 449
rand~ 200, 450
random signal 56, 373, 450
rate~ 451
receive~ 59, 453
Record and play sound files 117
record audio 105, 454
record Sound Designer II 485
record soundfile 117, 485
record~ 105, 454
Recording and playback 104
reflection of sound waves 192
remainder 213
reson~ 456
resonance of a filter 359, 456, 500
Review 76, 121
ReWire 458
rewire~ 458
ring modulation 85
Roads, Curtis 8, 176
round~ 461
routing a signal 336
Routing signals 59
routing signals 60

S

sah~ 462
sample and hold 15, 462
sample number 286
sample stored in memory 264
sample, read single 234, 344, 389
sample, write single 389, 425
Sampler 137
sampler 137
sample-to-millisecond conversion 463
sampling rate 15, 23, 299
    of AIFF file 140, 346
sampstoms~ 463
save a sound file 106
sawtooth wave 56, 69, 397
scheduler in audio interrupt 32
scope~ 163, 464
search path 265, 473, 475, 479
selector~ 68, 467
semitone 122
send~ 59, 469
seq 470
sfinfo~ 473
sflist~ 475
sfplay~ 478
sfrecord~ 485
sidebands 87, 91, 95
sig~ 64, 487
signal network 6, 22, 43
signal of constant value 375, 487
Signal vector size 31
signal vector size 31

Index

signal-to-float conversion 375, 493
simple harmonic motion 9
sine wave 9, 64, 290
sinh~ 488
sinx~ 490
slapback echo 189
slide~ 491
snapshot~ 161, 493
sound 8, 346
sound input 104, 232, 302
Sound Manager and MSP 36
sound output 292, 304
spectrum 11, 85, 167
spike~ 494
sqrt~ 496
square root of signal value 496
stutter~ 497
subpatch
    opening the window of 509
subpatch, mute audio of 72, 372
sustain 328
svf~ 500
switch 68, 467
synthesis techniques 81
synthesis, additive 81
Synthesizer 130

T

tanh~ 502
tanx~ 503
tapin~ 189, 504
tapout~ 189, 505
teeth~ 507
temperament, equal 371
Test tone 43
text, viewing a signal as 271
thispoly~ 509
thresh~ 511
threshold detection 511
timbre 11
train~ 512
transfer function 99, 357
trapezoid~ 514
tremolo 86, 91, 161
Tremolo and ring modulation 85
triangle~ 516
trunc~ 518
tuning, equal temperament 371
turning audio off and on 29
Turning signals on and off 68

U

Using the FFT 166

V

variable speed sample playback 107, 109, 338, 399
Variable-length wavetable 112
vector size 299
vectral~ 519
velocity sensitivity 130, 205
velocity-to-amplitude conversion 333
vibrato 86, 93, 122, 128
Vibrato and FM 93
Viewing signal data 157
vst~ 521

W

wave~ 112, 525
waveform~ 528
Waveshaping 99
waveshaping synthesis 99, 116
Wavetable oscillator 53
wavetable synthesis 44, 53, 112, 226, 290, 525
white noise 13, 56, 373
windowing 169

Z

zerox~ 537
zigzag~ 538