

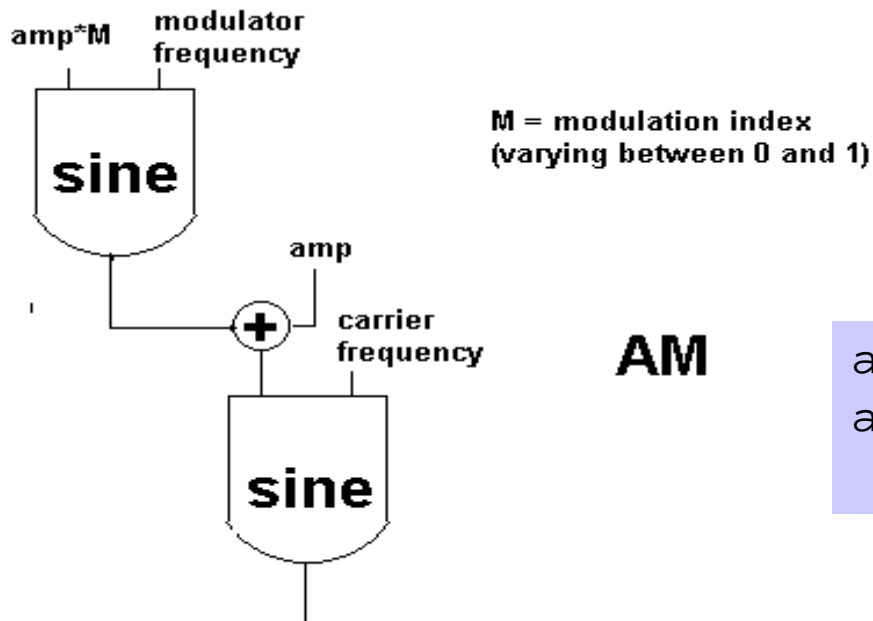
# Amplitude Modulation

When a parameter, such as the amplitude, is **cyclically** varied by an input signal, we say it is being *modulated*.

Modulation generally involves the output of a periodic signal generator, such as **oscillator**, controlling a parameter of another signal generator (which can also be another oscillator).

*Amplitude modulation* is the cyclic variation of **amplitude** of a signal by a **modulator**, a cyclically varying source.

# Carrier & modulator

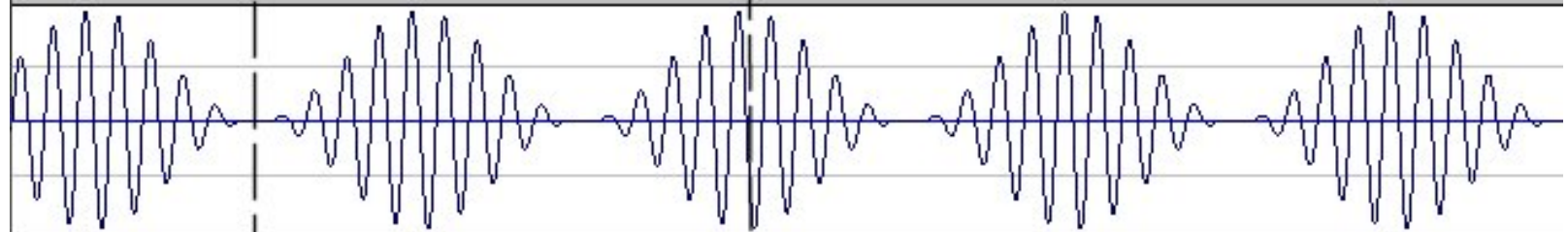
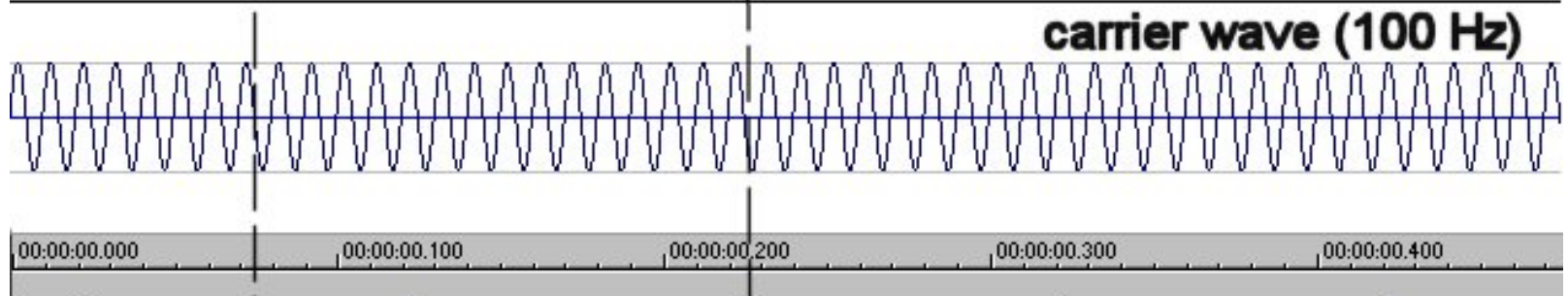
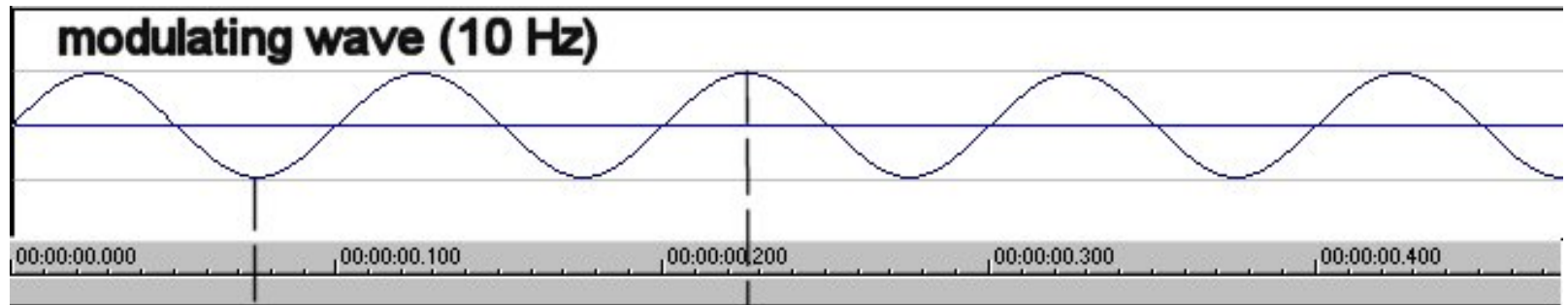


## csound code

```
amod  oscili  iamp*indx,imf,1  
acar  oscili  iamp+amod,icf,1  
out   acar
```

The **carrier** is the oscillator generating the audio signal. The **modulator** is the one modulating the amplitude.

In the above example the **carrier** amplitude will vary from  $amp - (amp * M)$  to  $amp + (amp * M)$ . The rate of modulation will be determined by the modulator frequency.



**100% modulation**

**minimum (0)**

**maximum ( $2 * \text{amp}$ )**

**amplitude modulation**

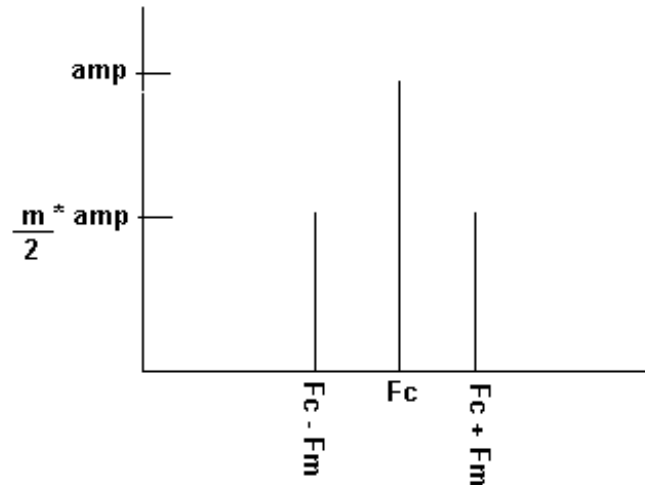
## Sub-audio AM and audio AM

When the **frequency of the modulator** is below the audio range ( $< 20$  Hz), resulting effect is a cyclically varying amplitude with a rate proportional to the frequency (something similar to *beats*).

When the modulator frequency is in the **audio range**, the output spectrum is composed of the **sum and difference** of the frequencies that compose both signals, plus the carrier frequencies.

*Ex.:* if the modulator and carrier are sine wave oscillators, with frequencies **400** and **500** Hz, respectively, the output signal will have the following component frequencies: **100** Hz, **500** Hz and **900** Hz.

## Output spectrum for AM (both oscillators using sinewaves)



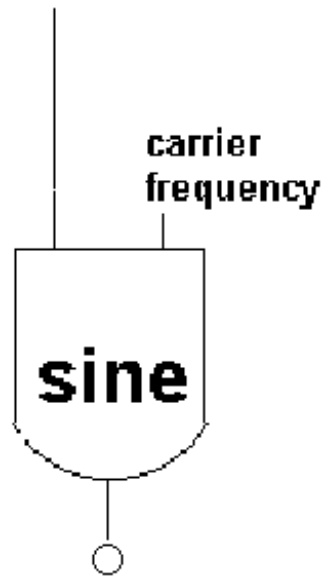
The spectrum is composed of the carrier frequency ( $F_c$ ) and two *sidebands* at the sum and difference of the two frequencies. Complex modulating spectra will generate more bands. For waves with more components (harmonics) the output will be the sums/differences between all components. A very complex spectrum.

# Ring Modulation

In *ring modulation* with a sine wave carrier, a signal is inserted directly in the amplitude input of the oscillator.

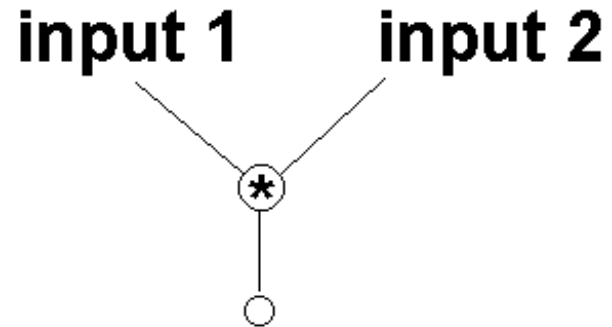
ring modulation (sine wave carrier)

**Signal Input**



```
aring oscili ainput,icf,1
```

general purpose ring modulator



```
aring = ainput1*ainput2
```

A general purpose ring modulator is a **multiplier**.

## Ring Modulation versus Amplitude Modulation

The main difference between the two techniques is that in the case of *ring modulation*, the amplitude of the signal will vary according to the modulator signal *only*.

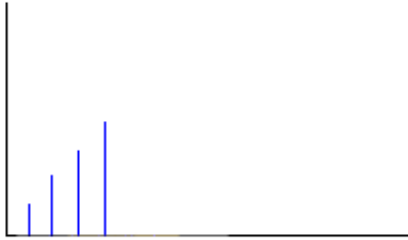
Ex.: if the modulator signal is a sine wave with amplitude *amp*, then the peak amplitude of the carrier wave will vary from 0 to *amp* to 0 to *-amp* and back to 0 every cycle.

A negative amplitude makes a wave invert its phase, but the signal will have the same strength as with a positive amplitude.

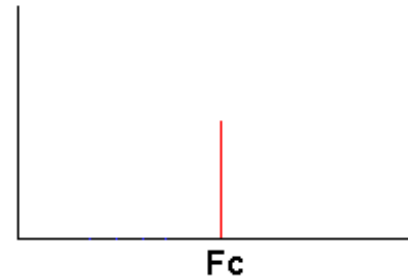
Also in the case of *audio*-range ring modulation, the output spectrum will be slightly different from the AM case.

# Ring modulation output spectrum

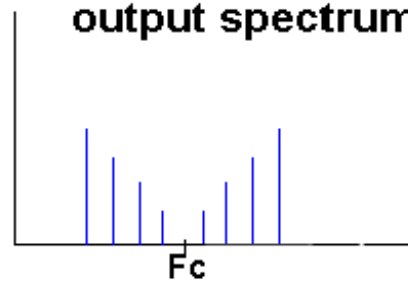
signal input spectrum



sine wave (carrier) spectrum



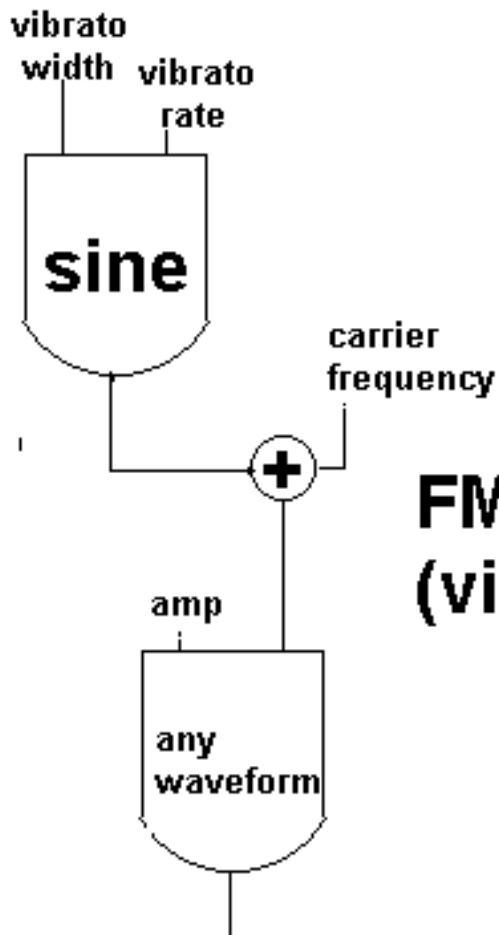
output spectrum



Only the sidebands are present:  $F_c + nF_m$  and  $F_c - nF_m$ ,  
no component at the carrier frequency!

Ring modulation of two complex spectra will generate a  
very dense spectrum.

# Frequency Modulation (vibrato)



## FM (vibrato)

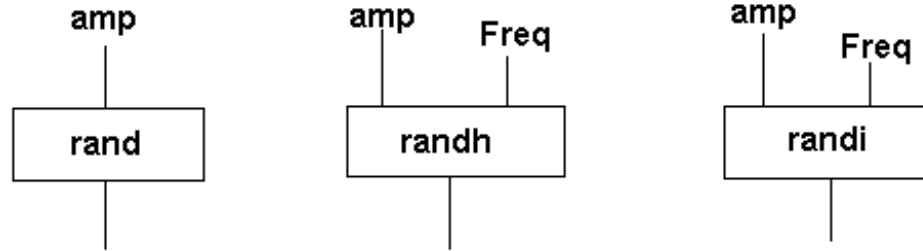
```
kmod  oscili  iwidth,ivrate,1
acar  oscili  iamp,icf+kmod,2
      out    acar
```

*Vibrato* is a small, cyclical and relatively slow variation of **frequency** (pitch). In order to simulate vibrato, we modulate the frequency of an oscillator with a (usually sinewave) oscillator.

The output frequency will vary between  $carrier\_freq + vibrato\_width$  and  $carrier\_freq - vibrato\_width$ .

The **rate** of vibrato is the frequency of the modulator.

# Noise generators



- **Rand**: produces non-band-limited (white) noise.
- **Randh**: produces band-limited noise, by sampling a random number *Freq* times per second and holding it until the next number is drawn.
- **Randi**: produces band-limited noise, by interpolating between the drawn numbers.

**Randh** and **Randi** can use the **freq** parameter as a “*bandwidth*” control. With them, **higher frequencies** can be attenuated, resulting in less bright sounds.

## Noise Generators (control & audio rate)

kr     **rand** kamp[ , iseed, iuse31 ]

ar     **rand** xamp[ , iseed, iuse31 ]

kr     **randh** kamp, kcps[ , iseed, iuse31 ]

ar     **randh** xamp, xcps[ , iseed, iuse31 ]

kr     **randi** kamp, kcps[ , iseed, iuse31 ]

ar     **randi** xamp, xcps[ , iseed, iuse31 ]

**iseed** is a value used to *seed* (ie. give an initial value to) the random number generator

**iuse31** makes the generators use an enhanced 31-bit algorithm (when set to 1, otherwise use the normal one).

## Using Noise Generators

Noise generators can be used in the generation of audio signals, often with filters in *subtractive synthesis* instruments.

One use of noise UGs to create percussive sounds was shown by JC Risset, in a *ring modulation* instrument design. With a sinewave oscillator, we ring-modulate a *band-limited noise* signal.

This has the effect of *centering* the noise band around the sinewave frequency. This then can be used to alter the characteristics of the sound. For deeper percussion sounds, we use lower frequencies, for brighter timbres, we can use higher frequencies.

The frequency control of the noise generator becomes then a noise bandwidth control ( $= 2 \times \text{freq}$ ).

## Example

```
kamp    expseg 0.1,iatt,iamp,idec,0.1,idur-(iatt+idec),0.1  
anoise  randi  kamp,ihbw  
aperc   oscili  anoise,icf,1
```



We use an exponential envelope that lasts for `iatt+idec`, ideally very short. That controls the noise amplitude.

The noise band half-bandwidth is `ihbw` and its centre frequency is `icf`, the sinewave oscillator.

By changing these values, we can get different timbral characteristics out of this design.

## Randomisation of controls

Another application of noise generators is to randomise parameters. In this case, we would be using control-versions of the noise generators.

It is important to know what kind of output to expect from the UGs. `Rand` will generate a new value anywhere between + and - *amp* every sample.

`Randh` will generate a new value between + and - *amp* at a certain rate. It will keep outputting the same value until a new one is generated.

`Randi` will generate lines between the random numbers. The duration of these lines will be  $1/\text{freq}$ .

# Examples

```
ilowest = 30
irange = (ilowest*3)/2
ibpm = 300
iamp = p4
kpitch randh irange,ibpm/60,2
kamp oscil iamp,ibpm/60,2
abass oscili kamp,kpitch+irange+ilowest,1
```



This generates a random bass-line. Notice we are seeding randh with the code 2 (seed from current time). This ensures a new set of values generated at every run of the instrument.

One oscillator creates an envelope, which is in sync with the new randh values. It will use an envelope table, such as:

```
f2 0 1024 5 0.1 24 1 1000 0.1
```

This generates a fast attack with an exponential decay

## Sampled-sound synthesis

Csound has a number of ways to access sampled sound stored in soundfiles. Two major types of access can be identified:

(a) *direct access*: by reading the soundfile directly  
**in, soundin, diskio**

(b) *table lookup*: loading the soundfile in a function table,  
and using an oscillator to play it.  
**loscil** is designed for this, but any oscillator  
or table reader can do it.

## Direct soundfile access: `in`, `ins`

These opcodes read sound in from a file. The first is simply

```
ar      in
```

This opcode takes the input from a file given as an option to the command-line:

```
csound (...) -i soundfile
```

Any soundfile format supported by `csound` can be used. The stereo version is (*nchnls* set to 2!)

```
a1, a2    ins
```

## Soundfile access: `soundin`, `diskin`

These opcodes read from an specified soundfile:

```
a1[,a2] soundin ifilcod[,iskptim][,iformat]
```

```
a1[,a2] diskin ifilcod,kratio[,iskiptim][,iwraparound][,iformat]
```

- **ifilcod** is the name of the soundfile under double quotes.
- **kratio** is the readout speed ( *kratio* < 1 means slower and *kratio* > 1 means faster, *negative means backwards*)
- *iskptime* is skiptime (from the beginning of the file)
- *iformat* is the format (not needed for self-describing formats)
- *iwraparound* is the wrap-around (looping) flag (1 => on, 0=>off)
- Remember to use the 2-channel opcode with stereo files!

## Soundfile access: table lookup

**GEN 1** defines a table which is filled from data off a soundfile.

Negative GEN code (-1) skip normalisation:

```
f# time size 1 filcod skiptime format channel
```

- **filcod** is the name of the soundfile in double quotes
- **skiptime** is the skiptime in seconds
- **format** is the format code, if 0 , the format is taken from the *soundfile header*
- **channel** is the channel number to be loaded in the table (0 means read all channels, interleaved)

## Deferred table allocation

With GEN01, there is a special provision for table allocation for use with soundfiles.

if the soundfile is AIFF, setting the *table size* to 0 will make the table *large enough* to accommodate the *whole soundfile*.

This type of table can only be used with the UG **l oscil**.

In addition other soundfile sampler-related data, such as base frequency and loop points can be stored in the table.

This is the easiest way of creating tables with GEN01 in sampler designs using the `l oscil` opcode.

## GEN01 example: RIFF-Wave files

```
f1 0 16384 1 "soundfile.wav" 0.5 0 1
```

- **ftable 1**

- 16384 samples read from *soundfile.wav* [371 msec at 44.1 KHz]
- skipping the first 0.5 secs
- format taken from soundfile header
- reading samples from channel 1.
- the samples are normalised (i.e. set to the range -1 to 1) after table generation (because **GEN** code is positive, 1).

## GEN01 example: AIFF files

```
f2    0    0   -1  "sound.aif"   0  0  0
```

- **ftable 2**
- Table size determined by AIFF soundfile *sound.aif*
- reading from beginning of file
- format as per soundfile header
- all channels are read (interleaved)
- The samples are **NOT** normalised (the GEN code is, -1, negative).

## Table lookup using `loscil`

A specialised oscillator is supplied for sampled-sound table lookup:

```
ar1 [,ar2] loscil xamp, kcps, ifn[, ibas] [,imod1,ibeg1,iend1] [, imod2,ibeg2,iend2]
```

**loscil** reads a mono or stereo sampled-sound table **ifn**, and scales the samples with **xamp**.

It outputs either one or two channels, depending on the number of channels in the function table.

The value of **kcps** determines the fundamental frequency of the sound playback.

## Fundamental frequency and loops

With **loscil**, the fundamental frequency will depend on *kcps* and *ibas*.

The latter is the base frequency of the sound in the table, whereas the second determines, based on that, the output frequency of the sound. Whatever scaling is necessary, it is done automatically. The value of *Ibas* can be omitted, if GEN01 has been filled with an AIFF file with base frequency information. If not it is required.

Loops can be set, using the parameters *imod*, *ibeg* and *iend*, or by using the loop markers already set in an AIFF file (similarly to *ibas*).

## Ordinary oscillators and GEN01

Ordinary oscillators can be used to read GEN01, provided that:

- (1) a power-of-two sized table is used (size cannot be taken from soundfile)
- (2) a single audio channel is stored in the table
- (3) you remember the lookup formula:

$$SI = \text{tablesize} \times (\text{fundamental} / SR)$$

which will determine the speed of the readout operation.

To read at the original speed,  $SI = 1$ .  $SI=2$  will read the sound at twice the speed and  $SI=0.5$  at half the speed. If  $SI$  is negative, we'll be reading the table backwards.

## Multichannel output

Csound can produce multichannel output, by setting **nchnls** to anything above 1 (mono). The simplest type of this is stereo (2 channels), but more channels can be used.

For realtime output, the maximum number of channels will depend on the audio hardware.

When using multiple channels it is important to note that both inputs and outputs will be expected to match the required number of channels.

So in stereo, we should using **ins** and **outs** instead of **in** and **out** (mono versions).

## Output opcodes

In addition to the already seen **out** opcode, we have the following ones:

```
outs    asig1, asig2                (also outsl1/outsl2)
outq    asig1, asig2, asig3, asig4 (also outql etc)
outh    asig1, asig2, asig3, asig4, asig5, asig6
outo    asig1, ..., asig8
outx    asig1, ..., asig16
out32   asig1, ..., asig32
outc    asig1[,asig2,....] (any number up to nchnls)
outch   kch1, asig1, kch2, asig2, .... kchN sets the output channel)
```

Multichannel output opcodes can only be used if **nchnls** is set to the right number of channels.

## Pan control

Pan, as in a mixing desk, offers control over the sound distribution among channels.

It is basically implemented as a set of gain controls defined by a single variable, the pan position.

For instance, in stereo, a simple pan control can be defined as (with  $i_{pan}$  between -1 and 1):

```
ileft = (1-ipan)/2
iright = (ipan+1)/2
outs asig*ileft, asig*iright
```

## Smooth Pan

The simple pan formula does not offer constant-power and smoothness as the pan control varies from -1 to 1.

For this, we need a slightly modified version:

```
icons = sqrt(2)/2
kdiv = (sqrt(1+kpan*kpan))
kleft = icons*(1-kpan)/kdiv
kright = icons*(1+kpan)/kdiv
```

In addition, using k-rate variables we can time-vary the pan position to suit any need.

## Summary: Key Concepts

- Oscillators: **oscil, oscili**
- Function tables and GENS: **GEN10** creates waveshapes
- Envelopes: **linen, envlpx, line, linseg, expon, expseg**, etc.
- Line segments and envelopes: **GEN5, GEN7, GEN 25, GEN 27**
- Envelopes with one-shot table lookup: **oscil1, oscil1i**
- Modulation: **AM, Ring Modulation and vibrato**
- Noise generators: **rand, randh, randi**
- Soundfile access: **in, ins, soundin, diskin**
- Sampled-sound function table: **GEN1**
- Sampled-sound table lookup: **loscil**