

## **Additive Synthesis**

Additive synthesis is the most basic synthesis technique. Its principle is derived from the fact that complex sound (any complex sound) is composed of a sum of sinusoidal waves of different amplitudes, frequencies and phases.

Theoretically, any sound can be built by the addition of sinusoidal waves, although in practice this might prove too difficult.

## Harmonics and Inharmonic components

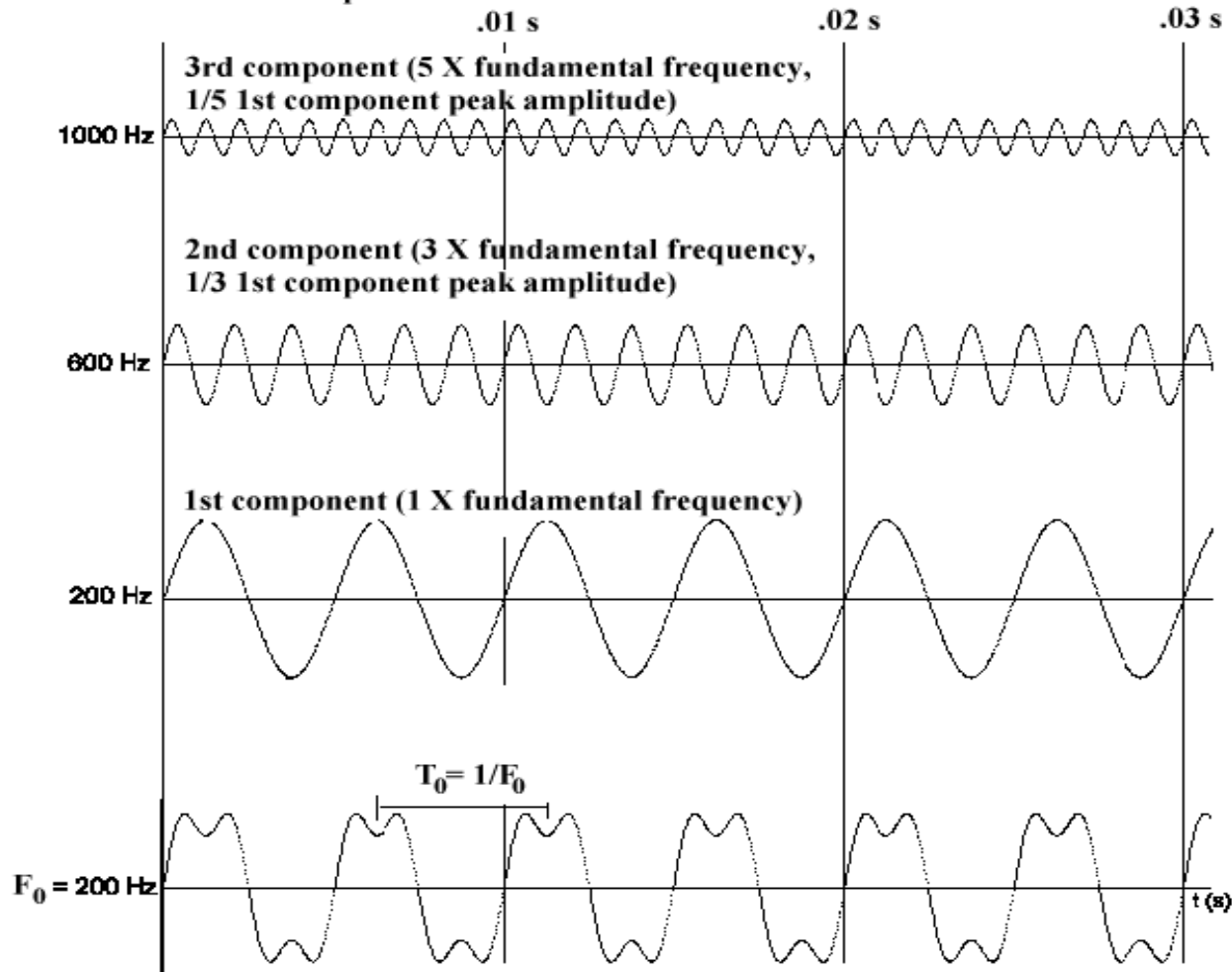
Any complex wave can be broken down into a set of sinusoidal components.

If the wave is periodic, these components are going to be **harmonics** (i.e. integer multiples) of a fundamental frequency.

If the wave is not periodic, then the components can be of any frequency, and they are said to be **inharmonic**.

# Harmonics (in a square wave)

Square wave with 3 harmonics, shown as a linear sum of its components



## Opcodes for Additive synthesis

Since we will be trying to generate a number of sinusoidal waves and sum them together, the basic opcode for additive synthesis will be the oscillator.

Of course, one oscillator on its own is capable of reading any waveshape, thus producing a complex periodic sound (composed of many sinusoidal waves added together).

This way we will be creating a complex fixed waveform (and spectrum), but for interesting sounds we will need to create a wave with an evolving shape, a *dynamic spectrum*.

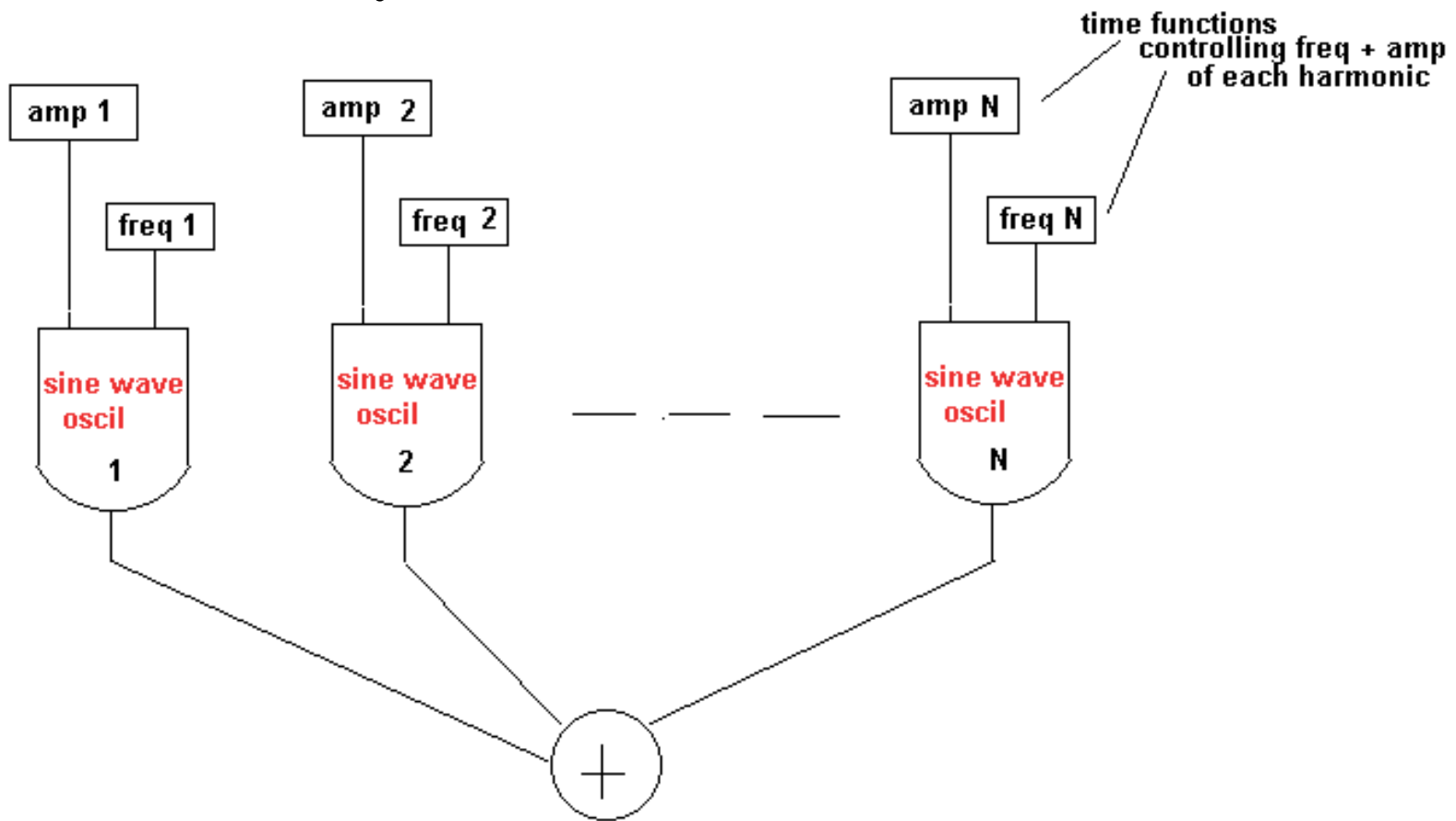
# Instrument Design

A typical additive synthesis instrument is created by adding the outputs of a number of sinusoidal oscillators, each one contributing to one component (or partial) of the sound spectrum.

In order to create an evolving spectrum, the amplitudes and frequencies will have to be controlled by separate envelopes (aka *time functions*).

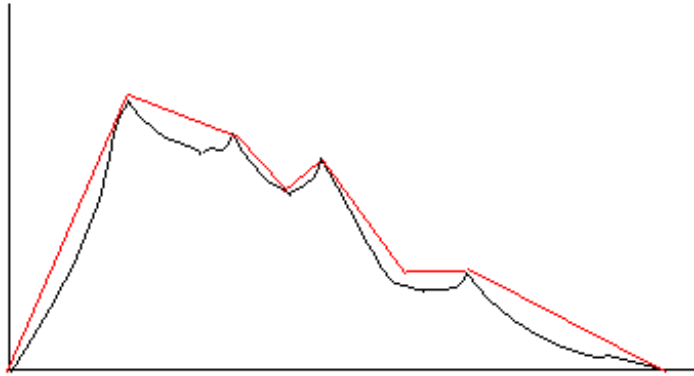
The classic design for an additive synthesis instrument will include  **$n$  oscillators** controlled by  **$n$  pairs of time functions** (one for amplitude and the other for frequency)

# An Additive Synthesis Instrument:

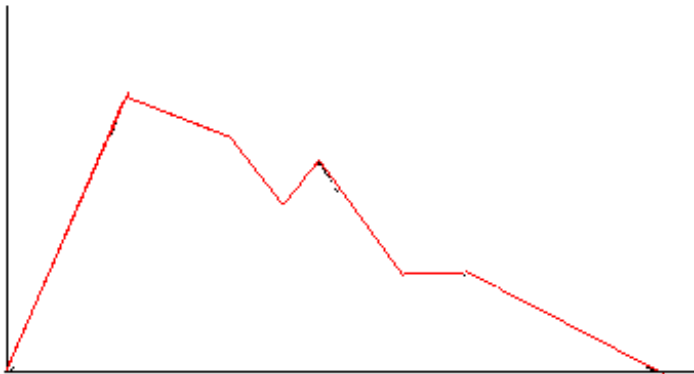


**Time functions** can be created by any **envelope generator**: oscillators reading tables, line/exponential envelopes, etc...

## Examples of time functions



(a) real amplitude envelope of  
a sound partial  
(data from analysis, in black)



(b) approximated time function  
with **linear segments**.  
This shape can now be created  
by a **GEN** and stored in a **table**

# In csound...

## An additive synthesis instrument

```
kamp1  oscil1  0, iamp1, idur, itable1  ; iamp1 = partial 1 max amp, itable1 = amp time function  
kfreq1  oscil1  0, ifreq1, idur, itable2  ; ifreq1 = partial 1 max freq, itable2 = freq time function  
apart1  oscil   kamp1, kfreq1, isine      ; apart1 = partial 1 audio output, isine = sinewave table
```

```
kamp2  oscil1  0, iamp2, idur, itable3  
kfreq2  oscil1  0, ifreq2, idur, itable4  
apart2  oscil   kamp2, kfreq2, isine
```

```
(...)          /* etc... define other oscillator+envelope pairs  
                one for each partial that will be synthesised */
```

```
out  apart1 + apart2 + ...      ; add (mix) all the oscillator outputs
```

## Difficulties

The strengths of additive synthesis do not need to be described: since it can create any imagined/existing sound it is the most powerful technique in existence.

The difficulty with it is its practical problems. Additive synthesis is very **raw** in design, it basically looks into the spectral model of sounds and try to recreate them.

Two main problems arise:

(1) **computational complexity**: very demanding on the computing side and not particularly efficient

(2) **data multiplication**: having two sets of functions for each oscillator (partial) makes it a lot of data when modelling sounds with lots of partials. Also, to achieve good results, these functions can be very complex.

## Solutions

(1) **Simpler additive instruments:** less partials; using only amplitude functions; simpler functions;

(2) **Efficiency measures:** turning off oscillators when not needed; using only significant partials; accounting for masking effects;

(3) **Analysis-resynthesis:** generating time function data by means of spectral analysis, resynthesing sounds with the generated time functions.

A typical tool used for spectral analysis is the **Phase Vocoder** (PVOC). Csound has an utility program to analyse soundfiles and generate PVOC data for additive synthesis.

## Control of flow in Csound

Control of flow is something found in most programming languages. It is used to control the order in which statements are executed.

In **csound** they can be used to control when opcodes are active or not. They can be used to turn oscillators *on/off*, for instance. These are the basic control-of-flow constructs found in Csound:

```
igoto, kgoto, goto
if ... igoto
if ... kgoto
if ... goto
timeout
```

**igoto** *label*

jumps to *label* (**initialisation time only**)

**kgoto** *label*

same, k-time only

**goto** *label*

**at every sample** jump to *label*

example:

```
igoto def
```

```
(...)
```

```
def: i1 = 0
```

## IF statements

These test if a condition is true and then either jump (`goto`) to a label or ignore the jump. A *condition* usually involves testing one or more variables using `<`, `>`, `<=`, `>=`, `==`, `!=`, called *relational operators*.

**if** *condition* **igoto** *label*

i-time only. The conditional expression can only take i-time variables (eg. `ia<ib`, `ia==ib`, etc)

**if** *condition* **goto** *label*

tested at every sample. The conditional expression can only take i-time variables (eg. `ia<ib`, `ia==ib`, etc)

**if** *condition* **kgoto** *label*

tested at every control period. The conditional expression can take *k-rate* variables (**very useful!**).

## Example

The k-rate IF is probably the most useful construct for switching things on and off. For instance:

```
kcounter line 0, p3, p3           ; time counter
if kcounter > 1.5 kgoto second ; after 1.5 secs
a1 oscil 4000, 440, 1           ; first oscil
second:
a2 oscil 4000, 550, 1           ; second oscil
   out  a1 + a2
```

after 1.5 secs, the instrument switches 1st oscillator off, because it will keep jumping to the label *second*, ignoring the first oscillator line.

# Timeout

Timeout is an opcode that also controls the flow of an instrument. It makes the control jump to a label, after a certain time, for a certain duration:

**timeout** *istart, idur, ilabel*

Example:

```
timeout 1.5, p3 - 1.5, second ; after 1.5 secs, jump  
a1  oscil  4000, 440, 1  
second:  
a2  oscil  4000, 550, 1  
    out  a1 + a2
```

The jump will occur *after* 1.5 secs, lasting for  $p3 - 1.5$  (the *rest* of the *note duration*). Before that, the *flow is normal*.

## Applications

True additive synthesis can only be achieved by using a software synthesis program such as **csound**. Here are some examples of sounds typically created with this technique:

(a) bell/gong-like (inharmonic spectra) sounds, with a small set of partials

(b) harmonic/inharmonic dynamic textures

(c) recreation/modification of real sounds by analysis and resynthesis

## Example

*Sheppard tones*: these are psychoacoustic illusions, created by careful additive synthesis. This example shows a never-ending glissando, created by:

- (a) ten parallel sinewave oscillators tuned in octaves
- (b) their frequency slides 10 octaves from topmost to the lowest
- (c) their amplitude is controlled so that when they are the extremes of that range, they are silent.

The result is that we only hear the sound of the oscillators when they are at mid-range, so there is an illusion that the glissando never ends (we also cannot hear its beginning).

## Code detail

```
/* these generate a ramp from 0 to the
   table sizes (isize2, isize3 sizes of tables 2 & 3)
   in 120 secs, used as an index for table lookup */
k1  line    0, 120, ilen2
k2  line    0, 120, ilen3
/* this reads table 2 (the "bell" function),
   indexed by k1, whis is generated above */
kamp1  tablei k1,2,0,0,1
/* this reads table 3 (exponential) */
kfreq1 tablei k2,3,0,0,1
/* this is the first component glissando */
aout1  oscili kamp1*iamp,kfreq1*ifreq,1
/* second component: tables read as before
plus offset of 1/10th (0.1) table size */
kamp2  tablei k1,2,0,ilen2*0.1,1
kfreq2 tablei k2,3,0,ilen3*0.1,1
aout2  oscili kamp2*iamp,kfreq2*ifreq,1
(...)
/* tenth component: 9/10ths offset */
kamp10 tablei k1,2,0,ilen2*0.9,1
kfreq10 tablei k2,3,0,ilen3*0.9,1
aout10 oscili kamp10*iamp,kfreq10*ifreq,1
```

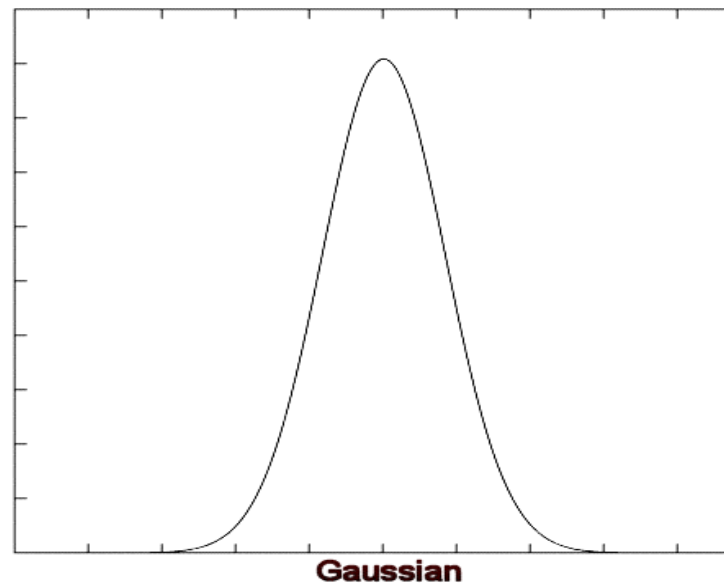
## Function Tables

The important point about this design is the function are used to control the amplitude envelope

We can use anything that tends towards zero at its edges, but the original function used was a gaussian envelope (the so-called “bell” function) created with **GEN20**:

**f2 0 512 20 6 1**

The other function table used is just a simple exponential line (GEN 5) from 1 to  $2^{-10}$  (ten octaves down)



## Table Lookup

The previous example also introduced a new UG, a *table lookup* unit generator (or *table reader*), defined by the opcodes `table` (truncating) and `tablei` (interpolating):

```
ir\kr\ar table  indx,ifn[,ixmode][,ixoff][,iwrap]
ir\kr\ar tablei indx,ifn[,ixmode][,ixoff][,iwrap]
```

They output a value read from a table `ifn` at position `indx`. Other important details are:

- Readout can be looped (`iwrap=1`) or limiting (`iwrap=0`).
- Indices can be offset by any amount [`ixoff`].
- Indices can be normalised (0-1, `ixmode=1`), or raw (0-tablesize, `ixmode=0`).
- There are three versions of each UG for *i-time*, *k-rate* or *a-rate*.

## An Oscillator in Pieces

*Table lookup* is one of the component operations of an oscillator. The other is the index increment, more technically known as *phase increment*. We can construct an oscillator by combining these two operations, plus the *amplitude scaling*:

```
aphs    phasor ifreq
asig    table  aphis, itable, 1, 0, 1
aosc    =      asig*iamp
```

The phasor opcode defines a phase increment UG, which creates a normalised (0-1) moving phase at a certain rate ifreq. This is effectively a ramping signal (sawtooth-shape).

## Synthesis of Inharmonic Spectra

Another typical application of additive synthesis is to generate discrete inharmonic spectra, such as *bells*, *gongs*, etc..

JC Risset has provided some examples of this applications. He has shown a bell instrument design using 11 partials with different *frequencies* and *amplitudes*.

The amplitude envelopes for all sounds has the same shape, a decaying exponential. However, some partials will have longer or shorter envelope *durations*.

We will be able to switch oscillators off when partials disappear.

## Risset's Bell Recipe

Here is a table with the frequencies, amplitudes and envelope durations for Risset's design. **Freqs/Amps** are relative to base values:

Partial Number	Frequency (Hz)	Amplitude (relative to a reference value)	Duration (secs)
1	$0.58 * \text{Freq}$	1	1
2	$0.58 * \text{Freq} + 1$	0.67	0.9
3	$0.91 * \text{Freq}$	10	0.65
4	$0.91 * \text{Freq} + 1.7$	1.8	0.55
5	$1.6 * \text{Freq}$	1.67	0.35
6	$1.2 * \text{Freq}$	2.67	0.325
7	$2 * \text{Freq}$	1.46	0.25
8	$2.7 * \text{Freq}$	1.33	0.2
9	$3 * \text{Freq}$	1.33	0.15
10	$3.75 * \text{Freq}$	1	0.1
11	$4.09 * \text{Freq}$	1.33	0.075

# Code detail

```
/* first and longest-lasting partial */
k1  expon    1, idur, exp(-10)
a1  oscil    k1*iamp, ifr*.56, isine

/* second partial, turned on only for 90% of
the total duration */
if kcount > idur*0.9*kr kgoto mix
k2  expon    1, idur*0.9, exp(-10)
a2  oscil    k2*iamp*0.67, ifr*0.56+1, isine

(...)

/* 11th partial, on for only 7.5% of the duration */
if kcount > idur*0.075*kr kgoto mix
k11 expon    1, idur*0.075, exp(-10)
a11 oscil    k11*iamp*1.33, ifr*4.07, isine

/* output mix of all partials */
mix: out      a1+a2+a3+a4+a5+a6+a7+a8+a9+a10+a11
```

## Analysis-Synthesis Methods

Additive synthesis is probably best handled by methods using a previous analysis of sound spectra. These can generate the time-functions for amplitude and frequency needed to drive a bank of sinusoid oscillators. Csound offers two analysis tools and a number of unit generators for additive synthesis.

The most common analysis tool is the **Phase Vocoder**, which can be understood as a *filterbank* that obtains amplitude & frequency values for a sound at regular time intervals.

Another tool that can be used is the **Heterodyne** filter, which also generates amp & freq functions, but works in a different way.

## Phase Vocoder Analysis

In a general way, PV analysis has the following characteristics:

- A bank of filters (*csound default: 512*) analyses equal-sized spectral bands (bandwidth =  $(SR/2)/bands$ , default: 43.3Hz).
- Bands are perceptually larger at lower frequencies.
- New amp & freqs for each band are obtained at a regular interval (*default: every 256 samples*).
- Amp & freq data are organised in time-ordered frames, themselves organised in freq. bands from 0 Hz to  $SR/2$ .
- PV analysis is actually a *Fourier Transform*-based process, implemented using a FFT (Fast Fourier Transform) algorithm.

## Csound PV analysis utility

Csound offers a utility program, called *pvanal*, which generates PV datafiles from soundfiles. The generated files can be used with PV additive synthesis unit generators.

```
> csound -U pvanal soundfile.wav analysisfile.pv
```

This analyses the first channel (if stereo) of soundfile *soundfile.wav* with 512 bands (1024 framesize) every 256 samples. These analysis parameters can be changed (see the csound manual for details).

The PV analysis data is written to *analysisfile.pv*. This will be in the analysis directory (d:/audio), ready to be used by csound instruments.

## Csound PV-data additive synthesis

There are a number of opcodes that use PV data. The ones relevant to additive synthesis are:

```
ar pvadd ktimpnt, kfmod, ifilcod, ifn, ibins
```

This opcode performs additive synthesis using a bank of oscillators. *ktimpnt* is a time pointer or counter that controls the reading from a data file.

*kfmod* is a pitch control (affects all partials the same way)

*ifilcod* is the data filename in double quotes.

*ifn* is the wavetable for the oscillators

*ibins* is the number of frequency bands, amp & freq pairs (bins) used in the synthesis.

```
kfrq, kamp pvread ktimpnt, ifilcod, ibin
```

This one reads control functions from a PV file, for a specific frequency band *ibin*.

## Adsyn and Hetro

Adsyn is another opcode for additive synthesis in csound. It takes an input datafile with amplitude and frequency functions and synthesizes a sound with a bank of oscillators:

```
asig adysn kampscale, kpitch, ktimescale, ifilcod
```

The input datafile is a series of amplitude/frequency tracks, in *breakpoint* format (time, value), one for each partial (*max 50*). This data can be either generated by the user, or by a Heterodyne filter analysis utility, *hetro*.

```
>csound -U hetro soundfile.wav analysisfile.het
```

Hetro is designed to work with harmonic spectra with clear fundamentals. Analysis of inharmonic spectra will not be successful in terms of fidelity.

## Summary and Key Concepts

- Additive synthesis is based on mixing sinusoidal sounds together.
- Each sinusoid is generated by an oscillator controlled by an independent amplitude & frequency function pair.
- Additive synthesis is computationally intensive and requires a large control dataset.
- It is a powerful method, capable of generating almost any type of sound.
- Additive synthesis is best handled by analysis-synthesis methods
- PV is one such method, which works generally well.
- Heterodyne filtering also can be used, with more limited applications and mixed results.