■ Miscellaneous`Audio`

Here's a list of functions defined in this package.

`ReadSoundfile`	`AmplitudeModulation`
`SoundList`	`RingModulation`
`Scale`	`FrequencyModulation`
`Wave`	`CascadeFM`
`FourierWave`	`ParallelFM`
`MakeWave`	

Functions defined in `Miscellaneous`Audio``.

This loads the package.

```
In[1]:= <<Miscellaneous`Audio`
```

You can use this package in at least two different ways: in a spirit of play to explore the world of sounds that can be created with *Mathematica*, or with attention to the constraints of digital audio. Three constraints, described briefly below, are: `SampleRate`, `SampleDepth`, and `PlayRange`.

The `SampleRate` is the number of times per second that the amplitude of a function is calculated. According to the sampling theory of Shannon and Weaver, n samples per second are required to accurately represent a waveform with a fundamental frequency of n=2 Hz. In other words, your sampling rate must be at least twice the magnitude of the highest frequency you'd like to hear. (Aliasing, the addition of low-frequency components to a sound, is the result of using a too-low sampling rate.) Since people register frequencies in the range of 20–20,000 Hz as audio signals, a sampling rate of 40,000 samples per second is the minimum required for the accurate encoding of the entire audio range.

Mathematica allows you to set the `SampleRate` to any positive number. However, the sound that will be created depends upon the sampling rate of your digital-to-analog converter. If your chosen sampling rate is not congruent with that of your converter, a software conversion to the converter's sampling rate will automatically take place, and you may get distortions.

For example, try the following `Play` statements to explore the consequences of different sampling rates:

Here's the pitch A-440, using the default `SampleRate`.

```
In[2]:= Play[Sin[2 Pi 440 t], {t,0,0.5}]
Out[2]=-Sound-
```

Here's the same sound with `SampleRate->1234`. Notice the additional noise.

```
In[3]:= Play[Sin[2 Pi 440 t], {t,0,0.5}, SampleRate->1234]
Out[3]=-Sound-
```

The `SampleDepth` (frequently called "quantization") is the resolution at which the amplitude of every sample is calculated. Setting the `SampleRate->8` gives you 2^8 different amplitude values, and `SampleRate->16` gives you 2^{16}. You should choose a value for `SampleDepth` that is the best for your hardware. Here are two examples of a `Play` statement using different values for `SampleDepth`.

Here's the pitch A-440 using the default setting of `SampleDepth`.

```
In[4]:= Play[Sin[2 Pi 440 t], {t,0,0.5}]
Out[4]=-Sound-
```

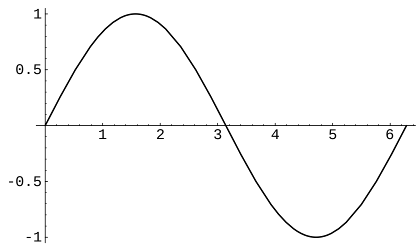
Here's the same pitch with `SampleDepth->4`. Notice how much noisier this one is.

```
In[5]:= Play[Sin[2 Pi 440 t], {t,0,0.5},SampleDepth->3]
Out[5]=-Sound-
```

The `PlayRange` controls the scaling of amplitudes passed to the `Play` or `ListPlay` functions. The `Audio` package automatically sets `PlayRange` to `All`, which is the optimal setting. The significance of `PlayRange` is best shown with a visual analogy.

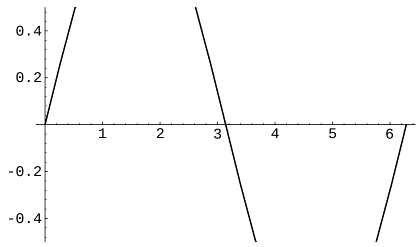
Here's a simple `Plot`.

```
In[6]:= Plot[Sin[x], {x,0,2Pi}]
```



Here's the same `Plot` with a restricted `PlotRange`.

```
In[7]:= Plot[Sin[x], {x,0,2Pi}, PlotRange->{-0.5,0.5}]
```



Notice that in the second plot above, when the value of the function exceeds the limits of the `PlotRange`, the values are cut-off, or "clipped". So, if the amplitude of your sound exceeds the setting of `PlayRange`, what you hear will not be an complete representation of your function. (It may well be enticing, but it won't be complete. This is called "waveshaping" if you're a composer, and "distortion" if you're an engineer.) Thus, your `PlayRange` should be set to a high enough level to allow for all the samples you generate to be present in the sound without distoring your waveform.

	`SampleRate`	`SampleDepth`
Apple (Apple Sound Chip)	5563.6363	8
	7418.1818	8
	11127.2727	8
	22254.5454	8
NeXT	8012.8210513	8
	22050	8 or 16
	44100	8 or 16
SGI Iris Indigo	8000	8
	11025	8
	16000	8
	22050	8 or 16
	32000	8 or 16
	44100	8 or 16
	48000	8 or 16
Sony News	8000	8 or 16
	9450	8 or 16
	18900	8 or 16
	37800	8 or 16
Sun (AM79C30A Digital Subscriber Controller chip)	8000	8 or 12
Windows (SoundBlaster or MultiMedia Extension)	11025	8
Windows (SoundBlaster Pro)	44100 (mono)	8
	22050 (stereo)	8

Optimal `SampleRate` and `SampleDepth` settings for selected machines.

Reading a Sound File from Disk

`ReadSoundfile["`*soundfile*`"]`	read in the specified sound file, and return the sequence of samples as a list of integers between -32768 and +32767

Reading a sound file into *Mathematica*.

`ReadSoundfile` lets you read a sound file directly into *Mathematica*. The list of amplitudes are translated into signed, 16-bit integers.

If the sound file is stereo, `ReadSoundfile` returns a list of lists for the right and left channels.

The returned list can be heard with `ListPlay`, or seen with `ListPlot`.

This reads the sound file "Voice.snd", and returns its samples to the list "voice".

```
In[8]:= voice = ReadSoundfile["Voice.snd"];

Soundfile: Voice.snd
Size: 25312 bytes
Samples: 6328
Duration: 0.286984 seconds
Format: 16-bit linear
SamplingRate: 22050
Channels: 2
Converting to signed 16-bit integers.
Returning stereo data.
```

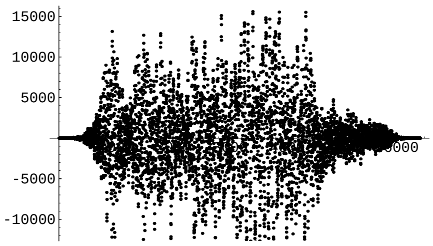
Now you can play "voice" with `ListPlay`.

```
In[9]:= ListPlay[voice,SampleRate->22050,SampleDepth->16];

Out[9]=-Sound-
```

You can plot the sum of right and left channels with `ListPlot`.

```
In[10]:= ListPlot[Plus @@ voice];
```



Playing a Sequence of Sounds

`SoundList[{`s_1`, `s_2`,..., `s_x`}]`	play a list of sounds in the given sequence.

Playing an arbitrary sequence of sounds.

SoundList gives you a means of playing sounds in sequence, without having breaks occur between them. The sounds in the list should be all of the same type: either all mono or all stereo.

Here is an example of a 440 Hz sine tone played for a quarter of a second, followed by a frequency-modulated tone for half a second, followed by a 550 Hz tone for a quarter of a second.

Here's a list of mono sounds passed to SoundList.

```
In[11]:= SoundList[{
          Play[Sin[2 Pi 440 t], {t,0,0.25}],
          Play[Sin[2 Pi 440 t + 10 * Sin[2 Pi 550 t]],
          {t,0,0.5}],
          Play[Sin[2 Pi 550 t], {t,0,0.25}]
          }]
Out[11]=-Sound-
```

Here's an example of using SoundList with the command Wave.

```
In[12]:= SoundList[{
          Wave[Sawtooth,1200,0.25],
          Wave[Square,100,0.3]
          }]
Out[12]=-Sound-
```

Stereo sounds can also be played with SoundList. This is an example of a 440 Hz sine tone played in the left channel, followed by a 622 Hz sine tone in the right channel, followed by a 880 Hz sine tone in both channels.

Here's an example of playing stereo sounds with SoundList.

```
In[13]:= SoundList[{
          Play[{Sin[2 Pi 440 t],0}, {t,0,0.25}],
          Play[{0,Sin[2 Pi 622 t]}, {t,0,0.25}],
          Play[{Sin[2 Pi 880 t],Sin[2 Pi 880 t]}, {t,0,0.25}]
          }]
Out[13]=-Sound-
```

Playing a Sequence of Intervals

Scale[*IntervalList*, *f*, *d*]	play a list of intervals starting at *f* Hz, where the interval list is assumed to be in cents measured from the starting frequency, for a duration of *d* seconds.

Playing a list of intervals in cents.

Scale can play a predefined list of intervals, or a user-defined list of numbers, given as the first argument.

This is a major scale in just intonation that starts at 440 Hz and lasts for 3 seconds.

```
In[14]:= Scale[JustMajor, 440, 3]
Out[14]=-Sound-
```

`QuarterTone`	`SixthTone`
`PythagoreanMajor`	`JustMajor`
`PythagoreanChromatic`	`JustMinor`
`MeanMajor`	`TemperedMajor`
`MeanMinor`	`TemperedMinor`
`MeanChromatic`	`TemperedChromatic`

Pre-defined sequences of intervals in cents.

`Scale` assumes that its first argument is a list of intervals measured in cents from the starting frequency.

The "cents" scale was invented by Alexander J. Ellis around 1885. Cents are additive, therefore they are related logarithmically to pitch ratios:

$$\textit{Pitch Interval} = K \times \log_{10}(f_1/f_2)$$

where f_1 and f_2 are the frequencies making up the interval. By definition, there are 1200 cents in an octave:

$$1200 = K \times \log_{10}(2/1)$$

giving K a value of 3986.314.

So, if you want to calculate a frequency and know the interval in cents, use the following formula:

$$\textit{Frequency} = \textit{StartingFrequency} \times 10^{(\textit{Interval}/K)}$$

where:

$$\begin{aligned} \textit{Frequency} &= \text{desired frequency in Hertz} \\ \textit{StartingFrequency} &= \text{known frequency in Hertz} \\ \textit{Interval} &= \text{known interval in cents} \end{aligned}$$

The list of intervals passed to `Scale` does not have to be in continuously ascending or descending order. Here is an example of using `Scale` to play an arbitrary list of intervals starting at 880 Hz.

This interprets a series of intervals and plays them.

```
In[15]:= Scale[{100,950,350,1200},880,1]
Out[15]=-Sound-
```

Creating Standard Waveforms

`Wave[`*type*`, `*f*`, `*d*`]`	play a wave of type *type* (which can be `Sine`, `Sawtooth`, `Triangle`, or `Square`), with a fundamental of *f* Hz for *d* seconds.
`FourierWave[`*type*`, `*f*`, `*p*`, `*d*`]`	play a wave of type *type* (which can be `Sawtooth`, `Square`, or `Triangle`) with a fundamental of *f* Hz, *p* number of partials, for *d* seconds.

Two ways to generate standard waveforms.

`Wave` provides computationally efficient methods for generating standard waveforms. These waveforms have a theoretically infinite number of partials.

This plays a sawtooth wave at 440 Hz for 0.1 seconds.

```
In[16]:= Wave[Sawtooth,440,0.1]
Out[16]=-Sound-
```

`FourierWave` allows you to specify the number of partials in a standard waveform.

The following equations are used to construct the waveforms:

sawtooth $$e(t) = \sum_{n=1}^{N} \frac{A}{n} \times \sin(n \times 2\pi f t)$$

square $$e(t) = \sum_{n=0}^{N} \frac{A}{(2n+1)} \times \sin((2n+1) \times 2\pi f t)$$

triangle $$e(t) = \sum_{n=0}^{N} \frac{A}{(2n+1)^2} \times \cos((2n+1) \times 2\pi f t)$$

where

$e(t)$ = amplitude at time t
N = number of the highest partial
A = peak amplitude
f = fundamental frequency in Hertz
t = time in seconds

A sawtooth wave consists of a fundamental and a theoretically infinite number of partials, containing all harmonics of the fundamental. The amplitude of each partial is the inverse of its harmonic number. Many instrumental sounds, such as string sounds, are approximated by sawtooth waves.

This plays a sawtooth wave with a fundamental of 440 Hz and 10 partials for 0.5 seconds.

```
In[17]:= FourierWave[Sawtooth,440,10,0.5]
Out[17]=-Sound-
```

A square wave consists of a fundamental and a theoretically infinite number of partials. The partials consist of the odd-numbered harmonics of the fundamental. Thus, a square wave has half as many partials present as a sawtooth wave. The amplitude of each partial is the inverse of its harmonic number. The clarinet is an instrument that emphasizes odd harmonics.

This creates a square wave with a fundamental of 880 Hz and 20 partials for 0.5 seconds.

```
In[18]:= FourierWave[Square,880,20,0.5]
Out[18]=-Sound-
```

A triangle wave has a fundamental and a theoretically infinite number of partials. Like a square wave, the partials consist of the odd-numbered harmonics of the fundamental, but unlike the square wave, each partial's amplitude is the inverse of the square of its harmonic number. This makes the upper harmonics much weaker than those of a square wave, and so makes its sound closer to that of a sine wave. The triangle wave approximates the sound of a flute.

This plays a triangle wave with a fundamental of 100 Hz and 100 partials for 0.5 seconds.

```
In[19]:= FourierWave[Triangle,100,100,0.5]
Out[19]=-Sound-
```

Creating Unusual Waveforms

`MakeWave[{{`h_1`, `a_1`}, {`h_2`, `a_2`}, ... , {`h_x`, `a_x`}}, `f`, `d`]`	play a wave with a fundamental of f Hz, every harmonic h_x at the relative amplitude a_x, for d seconds.

Creating unusual waveforms.

`MakeWave` allows you to specify the frequency and relative amplitude of all the partials of a fundamental. The first argument is a list of pairs of numbers, specifying the partial number h and its relative amplitude a.

With `MakeWave` you can specify partials that are not part of the natural overtone series of a given fundamental. (You can even specify waves in which the first partial has an amplitude of 0, and so the fundamental will not be heard at all.) The frequency of each partial is equal to h / f, where h is the partial number and f is the fundamental frequency. Here is an example of `MakeWave`, in which the resulting sound consists of three frequencies: 440, 660, and 770 Hz.

This is an example of building a wave in which the partials are non-harmonic and of equal amplitude.

```
In[20]:= MakeWave[{{1,1},{1.5,1},{1.75,1}},440,1]
Out[20]=-Sound-
```

The amplitude of every partial is given by a number between 0 and 1. Since you can individually specify the amplitude of every partial you want, you can generate "upside-down" tones in which the upper partials are stronger than the lower partials.

Here's an "upside-down" sound in which the upper partials are stronger than the lower ones.

```
In[21]:= MakeWave[{{1,0.5},{1.5,0.75},{1.75,1}},440, 1]
Out[21]=-Sound-
```

Amplitude Modulation

`AmplitudeModulation[`f_c, f_m, *mi*, *d*`]`	play an instance of amplitude modulation of sine waves, with carrier and modulating frequencies of f_c and f_m in Hertz, modulation index of *mi*, for *d* seconds.
`RingModulation[`f_c, f_m, *mi*, *d*`]`	play an instance of ring modulation of sine waves, with carrier and modulating frequencies of f_c and f_m in Hertz, modulation index of *mi*, for *d* seconds.

Amplitude modulation and ring modulation.

The following equations are used for amplitude modulation and ring modulation:

amplitude modulation $e(t) = A \cdot (1 + mi \cdot \cos(2\pi f_m t)) \cdot \cos(2\pi f_c t)$
ring modulation $e(t) = A \cdot (mi \cdot \cos(2\pi f_m t)) \cdot \cos(2\pi f_c t)$

where

$e(t)$ = amplitude at time t
A = peak amplitude
mi = modulation index
f_m = modulation frequency in Hertz
f_c = carrier frequency in Hertz
t = time in seconds

Amplitude modulation generates in three output frequencies: the carrier frequency, the *sum* of the carrier and modulating frequences, and the *difference between* the carrier and modulating frequencies.

The "depth" or "intensity" of the modulation is controlled by the modulation index *mi*:

- If $mi = 1$, the carrier amplitude varies between $2A$ and 0.
- If $mi = 0.5$, the carrier amplitude varies between $1.5A$ and $0.5A$.
- If $mi > 1$, the modulated signal is no longer sinusoidal in form, and the carrier is said to be overmodulated.

When the modulating frequency is sub-audio (below 20 Hz), the result is called "tremolo." Here is an example of tremolo, where the modulating frequency is 6 Hz, and the modulation index is 1.

This is an example of tremolo.

```
In[22]:= AmplitudeModulation[200,6,1,1]
Out[22]=-Sound-
```

Ring modulation is a type of amplitude modulation in which only the sum and difference of the carrier and modulator are present. Here is an example of ring modulation, where both the carrier and the modulator are in the audio range.

The output sound has 440 Hz (the *difference* between the two) and 1320 Hz (the *sum* of the two).

```
In[23]:= RingModulation[440,880,2,1]
Out[23]=-Sound-
```

Frequency Modulation

`FrequencyModulation[`f_c, f_m, pd, d`]`	play a frequency modulated wave with a carrier of f_c Hz, a modulator of f_m Hz, a peak deviation of pd, for d seconds.
`CascadeFM[`f_c, f_{m1}, f_{m2}, pd_1, pd_2, d`]`	play a wave in which the carrier frequency f_c is modulated by f_{m1}, which is itself modulated by f_{m2}, and where pd_1 and pd_2 are the peak deviation of f_{m1} and f_{m2} respectively, for d seconds.
`ParallelFM[`f_c, f_{m1}, f_{m2}, pd_1, pd_2, d`]`	play a wave in which the carrier frequency f_c is modulated by two frequencies, f_{m1} and f_{m2} that do not affect each other, and where pd_1 and pd_2 are the peak deviation of f_{m1} and f_{m2} respectively, for d seconds.

Frequency modulation.

The general formula for frequency modulation is:

$$e(t) = A \cdot \sin(2\pi f_c t + mi \cdot \sin(2\pi f_m t))$$

where

$$\begin{aligned} e(t) &= \text{amplitude at time } t \\ A &= \text{peak amplitude} \\ f_c &= \text{carrier frequency in Hertz} \\ mi &= \text{modulation index} \\ f_m &= \text{modulation frequency in Hertz} \\ t &= \text{time in seconds} \end{aligned}$$

The *Modulation Index* is the ratio of the peak deviation to the modulating frequency:

$$mi = PeakDeviation = f_m$$

Higher values of *mi* result in more partials, a "brighter" sound. Very high values of *mi* will result in aliasing, and will add low-frequency components to the sound.

This is an example of frequency modulation with a very high modulation index.

```
In[24]:= FrequencyModulation[440,550,5500,1]
Out[24]=-Sound-
```

If the frequency of the modulator is sub-audio, the result of frequency modulation is called "vibrato." Here is an example of vibrato, with a 440 Hz carrier, a 6 Hz modulator, and a *peak deviation* of 36, resulting in a *mi* of 6, lasting for 1 second.

Here's an example of vibrato.

```
In[25]:= FrequencyModulation[440,6,36,1]
Out[25]=-Sound-
```

The *Modulation Ratio* is the ratio of the modulating frequency to the carrier frequency.

- If *mr* is an integer greater than zero, the resulting overtones and sidebands (the spectrum) will have harmonic relationships to each other, analogous to those of standard orchestral instruments.
- If *mr* is not an integer value (if it has a fractional part greater than zero) the resulting overtones and sidebands will have an inharmonic relationship to each other, analogous to percussion instruments and bells.

One speaks of a harmonic relationship when the overtones and sidebands have frequencies that coincide with the harmonics of some fundamental, which need not be the same frequency as the carrier.

`CascadeFM`, is a type of frequency modulation in which the modulating frequency is itself modulated by a third frequency.

$$e(t) = A \left(\sin\left(2\pi f_c t + \frac{pd_1}{f_{m1}} \left(\sin\left(2\pi f_{m1} t + \frac{pd_2}{f_{m2}} \left(\sin(2\pi f_{m2} t)\right)\right)\right)\right)\right)$$

This is an exaple of cascade frequency modulation.

```
In[26]:= CascadeFM[440,880,220,1760,440,1]
Out[26]=-Sound-
```

`ParallelFM` is a type of frequency modulation in which the carrier is modulated by two frequencies that do not affect each other.

$$e(t) = A \left(\sin\left(2\pi f_c t + \frac{pd_1}{f_{m1}} \left(\sin(2\pi f_{m1} t)\right) + \frac{pd_2}{f_{m2}} \left(\sin(2\pi f_{m2} t)\right)\right)\right)$$

This uses the same frequencies as the previous example, but with `ParallelFM`.

```
In[27]:= ParallelFM[440,880,220,1760,440,1]
Out[27]=-Sound-
```

Equal-Tempered Pitch and Frequency Equivalents

The `Miscellaneous`Audio`` package provides a list of equal-tempered pitch/frequency equivalents. Pitches are named in "pitch class/octave" notation, where the pitch class is given by a letter from "A" to "G" (with the appendage "f" for flat and "s" for sharp), and the octave is an integer between 0 and 7. Most chromatic equivalences are available, *e.g.,* Cf is the same as B, and Es is the same as F. Double-flats and double-sharps are not defined.

This plays the pitches of a diminished chord.

```
In[28]:= SoundList[{
           Play[Sin[2 Pi C4 t],{t,0,0.1}],
           Play[Sin[2 Pi Ef4 t],{t,0,0.1}],
           Play[Sin[2 Pi Gf4 t],{t,0,0.1}],
           Play[Sin[2 Pi A4 t],{t,0,0.1}]
          }]
Out[28]=-Sound-
```

This plays a sawtooth wave at the pitch `Gs6`.

```
In[29]:= Wave[Sawtooth, Gs6, 1]
Out[29]=-Sound-
```

A0 = 27.500000	Cs0 = Df0 = 34.647831
As0 = Bf0 = 29.135235	D0 = 36.708099
B0 = Cf0 = 30.867706	Ds0 = Ef0 = 38.890877
Bs0 = C0 = 32.703197	

Equal tempered pitch-frequency equivalents with frequencies in Hertz.

```
E0 = Ff0 = 41.203449             G3 = 391.995483
Es0 = F0 = 43.653534             Gs3 = Af3 = 415.304749
Fs0 = Gf0 = 46.249310            A4 = 440.000000
G0 = 48.999435                   As4 = Bf4 = 466.163757
Gs0 = Af0 = 51.913094            B4 = Cf4 = 493.883301
A1 = 55.000000                   Bs4 = C4 = 523.251160
As1 = Bf1 = 58.270470            Cs4 = Df4 = 554.365295
B1 = Cf1 = 61.735413             D4 = 587.329590
Bs1 = C1 = 65.406395             Ds4 = Ef4 = 622.254028
Cs1 = Df1 = 69.295662            E4 = Ff4 = 659.255188
D1 = 73.416199                   Es4 = F4 = 698.456543
Ds1 = Ef1 = 77.781754            Fs4 = Gf4 = 739.988953
E1 = Ff1 = 82.406898             G4 = 783.990967
Es1 = F1 = 87.307068             Gs4 = Af4 = 830.609497
Fs1 = Gf1 = 92.498619            A5 = 880.000000
G1 = 97.998871                   As5 = Bf5 = 932.327515
Gs1 = Af1 = 103.826187           B5 = Cf5 = 987.766602
A2 = 110.000000                  Bs5 = C5 = 1046.502319
As2 = Bf2 = 116.540939           Cs5 = Df5 = 1108.730591
B2 = Cf2 = 123.470825            D5 = 1174.659180
Bs2 = C2 = 130.812790            Ds5 = Ef5 = 1244.508057
Cs2 = Df2 = 138.591324           E5 = Ff5 = 1318.510376
D2 = 146.832397                  Es5 = F5 = 1396.913086
Ds2 = Ef2 = 155.563507           Fs5 = Gf5 = 1479.977905
E2 = Ff2 = 164.813797            G5 = 1567.981934
Es2 = F2 = 174.614136            Gs5 = Af5 = 1661.218994
Fs2 = Gf2 = 184.997238           A6 = 1760.000000
G2 = 195.997742                  As6 = Bf6 = 1864.655029
Gs2 = Af2 = 207.652374           B6 = Cf6 = 1975.533203
A3 = 220.000000                  Bs6 = C6 = 2093.004639
As3 = Bf3 = 233.081879           Cs6 = Df6 = 2217.461182
B3 = Cf3 = 246.941650            D6 = 2349.318359
Bs3 = C3 = 261.625580            Ds6 = Ef6 = 2489.016113
Cs3 = Df3 = 277.182648           E6 = Ff6 = 2637.020752
D3 = 293.664795                  Es6 = F6 = 2793.826172
Ds3 = Ef3 = 311.127014           Fs6 = Gf6 = 2959.955811
E3 = Ff3 = 329.627594            G6 = 3135.963867
Es3 = F3 = 349.228271            Gs6 = Af6 = 3322.437988
Fs3 = Gf3 = 369.994476           A7 = 3520.000000
```

Equal tempered pitch-frequency equivalents with frequencies in Hertz.

```
As7 = Bf7 = 3729.310059        E7 = Ff7 = 5274.041504
B7 = Cf7 = 3951.066406         Es7 = F7 = 5587.652344
Bs7 = C7 = 4186.009277         Fs7 = Gf7 = 5919.911621
Cs7 = Df7 = 4434.922363        G7 = 6271.927734
D7 = 4698.636719               Gs7 = Af7 = 6644.875977
Ds7 = Ef7 = 4978.032227
```

Equal tempered pitch-frequency equivalents with frequencies in Hertz.