Gamma— C++ Generic Synthesis Library
Quick Tutorial (rev. 0.9.4)

November 11, 2010

Media Arts and Technology and the AlloSphere Research Facility

University of California, Santa Barbara

Author: Lance Putnam
e-mail: ljputnam@umail.ucsb.edu

Contents

1 Gamma Overview **2**
- 1.1 Genericity . . . 2
- 1.2 Generators and Filters . . . 2
- 1.3 Domain Synchronization . . . 2
- 1.4 Related Work . . . 2

2 Obtaining and Building **3**
- 2.1 Linux Compilation . . . 3
- 2.2 Mac OSX Compilation . . . 3
- 2.3 Windows Compilation . . . 3
- 2.4 Building With Make . . . 4

3 Gamma Objects **4**
- 3.1 Basic I/O . . . 4
- 3.2 Generators . . . 6
- 3.3 Filters . . . 8
- 3.4 Spectral Processing . . . 8
- 3.5 Effects . . . 10

4 Synthesis Examples **10**
- 4.1 Additive Synthesis . . . 10
- 4.2 Subtractive Synthesis . . . 10
- 4.3 Amplitude Modulation . . . 10
- 4.4 Frequency Modulation . . . 10
- 4.5 Physical Modeling . . . 11

5 Further Reading **11**

1 Gamma Overview

Gamma is a cross-platform, C++ library for doing generic synthesis and filtering of numerical data. It contains numerous mathematical functions, common algebraic types, such as vectors, complex numbers, and quaternions, an assortment of sequence generators and many objects for signal processing. It is oriented towards real-time sound and graphics rendering, but is equally useful for non-real-time tasks.

1.1 Genericity

Gamma objects are templated on their value type allowing them to be used with any arithmetic data types, such as, scalars, vectors, and complex numbers. Processing algorithms are designed as much as possible to be data type and domain independent. This allows a common set of functions and objects to be used for a variety of modalities, such as sound or graphics, and in arbitrary domains, such as time, space, or frequency. Algorithms are type generic which means they can process any type of data that has the appropriate mathematical operators defined (typically +, -, *, and /).

1.2 Generators and Filters

Processing objects are typically either a generator or a filter and can be either domain observers or not. Objects adopt a standard processing operator interface similar to functional syntax. Generators return their next value with the `()` operator and filters return their next value with the `(T)` operator, where `T` is a value type.

```
gen::RAdd rAdd(1, 0);                       // recursively add 1 starting at 0
for(int i=0; i<4; ++i) cout << rAdd();      // prints "1 2 3 4"
fil::Delay1 delay1(0);                      // 1 element delay starting with 0
for(int i=1; i<5; ++i) cout << delay1(i);   // prints "0 1 2 3"
```

1.3 Domain Synchronization

A sampling domain is a discretely-spaced independent variable along which a function is evaluated. In Gamma, domains are abstracted through an observer design pattern into a `Sync` subject and a `Synced` observer. The `Sync` holds a sampling rate/interval amount and notifies its attached `Synced`s whenever it changes value.

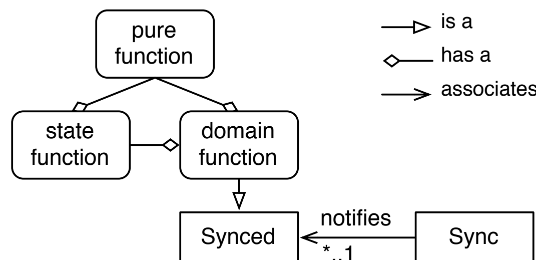


Figure 1: `Sync`/`Synced` observer pattern for domain synchronization

1.4 Related Work

The following are other cross-platform C++ sound synthesis libraries:

- CLAM, http://clam-project.org/
- CSL, http://fastlabinc.com/CSL/
- SndObj, http://sndobj.sourceforge.net/
- STK, https://ccrma.stanford.edu/software/stk/

	CLAM 1.4	CSL 5	Gamma 0.92	SndObj 2.6.6	STK 4.3.1
Spectral	DFT, STFT, PV	DFT	DFT, STFT, PV	DFT, STFT, PV	
Scalar proc.	No	No	Yes	No	Yes
Vector proc.	Yes	Yes	No	Yes	Yes
Graphs	Yes	Yes	No	Yes	No
Sample type	float	typedef	generic	float	typedef
Extra	MIDI, SDIF, Thread	MIDI, OSC, Exceptions, Instrument	Domain	MIDI, Thread	MIDI, SKINI Thread, Socket
Strength	spectral analysis	spatial audio	genericity	sin. synth.	phys. modeling
Missing	MUSIC 5		ADSR	pink noise	spectral
Dependencies	lib{ogg, mad, vorbis, sndfile} FFTW , PortAudio	JUCE, liblo*	libsndfile, PortAudio	FFTW*	
License	GPL	BSD	BSD	GPL	BSD

* dependency included in library

Table 1: Sound synthesis library comparison chart

2 Obtaining and Building

The Gamma source code can be downloaded from

```
http://mat.ucsb.edu/gamma/
```

or checked out through SVN here

```
svn co https://svn.mat.ucsb.edu/svn/gamma/trunk gamma
```

The simplest way to build the library is to use GNU Make. Instructions for setting up each platform with Make follow. Further instructions can be found in the `README` file in the root directory.

2.1 Linux Compilation

The simplest way to build the library is to use GNU Make (see section 2.4 below). Use either Synaptic or apt-get to install the most recent developer versions of libsndfile and PortAudio (v19).

2.2 Mac OSX Compilation

You can build the library using either GNU Make or Xcode. If you are using Make, see section 2.4 below. To use Xcode, you will need to install the developer tools from Apple. You can get them for free (after registration) from http://developer.apple.com/. The Xcode project is located at `project/xcode/gamma.xcodeproj`.

2.3 Windows Compilation

Option 1: Use Linux

To simplify this process, we will boot from the Linux live CD, Puredyne, which is bundled with several important media-related libraries. Once Puredyne successfully boots, you need to update it by executing the following commands in a terminal:

```
sudo apt-get update
sudo apt-get install g++-4.4
```

Option 2: DIY

You will need to either use Cygwin or MinGW with GNU Make or create a new Visual Studio project. Visual Studio Express can be downloaded for free from Microsoft. It will be necessary to link to libsndfile and PortAudio v19. Neither of these two methods have been tested, so you can make a contribution to Gamma if you come up with a working solution.

2.4 Building With Make

If you had to install Make, test that it is working by opening a terminal and typing 'make –version'. Before running Make, ensure that the correct build options are set in the file `Makefile.config`. These can be set directly in `Makefile.config` or passed in as options to Make as `OPTION=value`. If you are using Linux/Puredyne, you will also need to uncomment the line `USING_PUREDYNE = 1`. To build the Gamma library, run `make` and hope for the best.

3 Gamma Objects

3.1 Basic I/O

Audio Devices

[io/audioDevice]

Communication with audio devices is done through the `AudioIO` class. An `AudioIO` allows one to open a stream to any number of channels of input and/or output on a particular audio device. When the `AudioIO` is started, it will call a user supplied callback function at regular intervals. The rate at which the callback function is executed is the *sample rate* divided by the *block size* (both user specified). In the callback, it is up to the user to read samples from the input buffer(s) and/or write samples to the output buffer(s). The argument to the callback function is a reference to an `AudioIOData` object which holds onto the i/o buffers, user data, and other relevant information about the audio stream. Audio buffers are in a *non-interleaved* format so that samples within each channel are tightly packed.

The following code example illustrates how to open an audio stream to the default device and define a callback to process audio input.

```
#include "Gamma/AudioIO.h"

struct MyStuff{};

void audioCB(AudioIOData& io){

    MyStuff& stuff = io.user<MyStuff>();

    while(io()){
        float inSample1 = io.in(0);
        float inSample2 = io.in(1);
        io.out(0) = -inSample1;
        io.out(1) = -inSample2;
    }
}

int main(){
    MyStuff stuff;
    AudioIO audioIO(
        128,        // block size
        44100,      // sample rate (Hz)
        audioCB,    // user-defined callback
        &stuff,     // user data
        2,          // input channels to open
```

```
        2           // output
    );
    audioIO.start();
}
```

Sound Files

[io/soundFile]

The `SoundFile` object is used to read/write various kinds of uncompressed sound file formats, such as WAV, AIFF, AU, and FLAC. A sound file is usually comprised of a header describing the format (sample rate, amplitude resolution, compression algorithm), length (in samples), and number of channels of the data followed by a sequence of samples comprising the waveform of the sound. The data is most commonly stored in an *interleaved* format meaning that the samples iterate fastest over the channels, then over time. For example, an interleaved stereo sound would be stored as $L_1, R_1, L_2, R_2, ..., L_N, R_N$ where L_n and R_n are the left and right channel samples, respectively, at time n and N is the length of the sound. The following example shows how to open a file and store its contents into a buffer.

```
SoundFile sf("example.wav");
sf.openRead();                  // open file in read mode

double frameRate = sf.frameRate();  // get frame rate
int frames = sf.frames();           // get number of frames
int channels = sf.channels();       // get number of channels

float buf[frames * channels];       // create a buffer to store all the frames

sf.read(buf, frames);               // copy file contents to buffer
                                    // channels are interleaved
```

The next code example demonstrates how to write data to a sound file.

```
SoundFile sf("example.wav");
sf.openWrite();                     // open file in write mode

sf.frameRate(44100);                // set frame rate
sf.channels(1);                     // set number of channels
sf.format(SoundFile::WAV);          // set file format (WAV, AIFF, AU, RAW, FLAC)
sf.encoding(SoundFile::PCM_16);     // set sample encoding (PCM_S8, PCM_16, PCM_24, PCM_32,
                                    // PCM_U8, FLOAT, DOUBLE, ULAW, ALAW)

float buf[bufSize];                 // create a buffer to store all the frames
sf.write(buf, bufSize);             // write buffer to file
                                    // channels are interleaved
```

Recording

[io/recording]

The `Recorder` object provides a means for buffering real-time audio so that it can be saved to a file. It has a `write()` method for storing samples from a real-time audio thread into its buffer and a `read()` method for reading samples from a lower priority thread.

```
SoundFile sf("recording.aif");  // sound file to record to
Recorder rec(2);                // set up to record stereo audio

void setup(){
    sf.openWrite();
}

// in audio sample loop
```

```
{
    float s1, s2;               // samples of left and right channels
    ...                         // generate samples
    rec.write(s1, s2);          // write stereo frame into buffer
}

// called periodically from non-audio thread
{
    float * buf;                // this will point to the read samples
    int n = rec.read(buf);      // copy samples from ring to read buffer
    sf.write(buf, n);           // write samples to the sound file
}
```

3.2 Generators

Oscillators (Accum, Sweep, LFO, Osc, Impulse, Saw, Square, DSF)

[generator/(oscAccum, oscSweep, oscLFO1, oscLFO2, oscOsc, oscImpulse)]

`Accum` accumulates an internal fixed-point phase at a specified frequency. Requiring only a single fixed-point addition per iteration, it is the fastest generator. It can be used as a timer or as the input to a waveshaping table or function.

`Sweep` cycles linearly through the interval [0, 1) at a specified frequency.

`LFO` (low-frequency oscillator) produces various waveforms derived from the phase of an `Accum`. Figure 2 shows the static waveforms that can be produced.

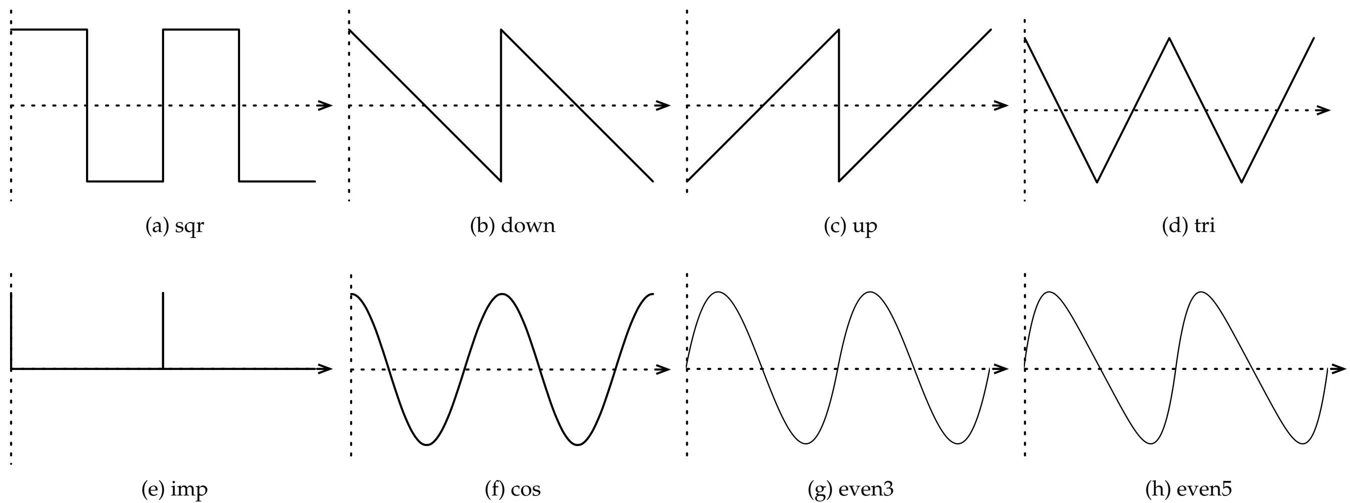


Figure 2: `LFO` static waveforms (two cycles shown)

Figure 3 shows the dynamic waveforms whose shape can be changed smoothly with the `mod` parameter.

`Osc` is an `Accum` that sweeps through values in a table. Osc is the most general type of oscillator as any waveform can be stored in its table. An `Osc` can either use its own internal table or reference an exisiting table.

`Impulse` generates a band-limited impulse having either all harmonics or only odd harmonics.

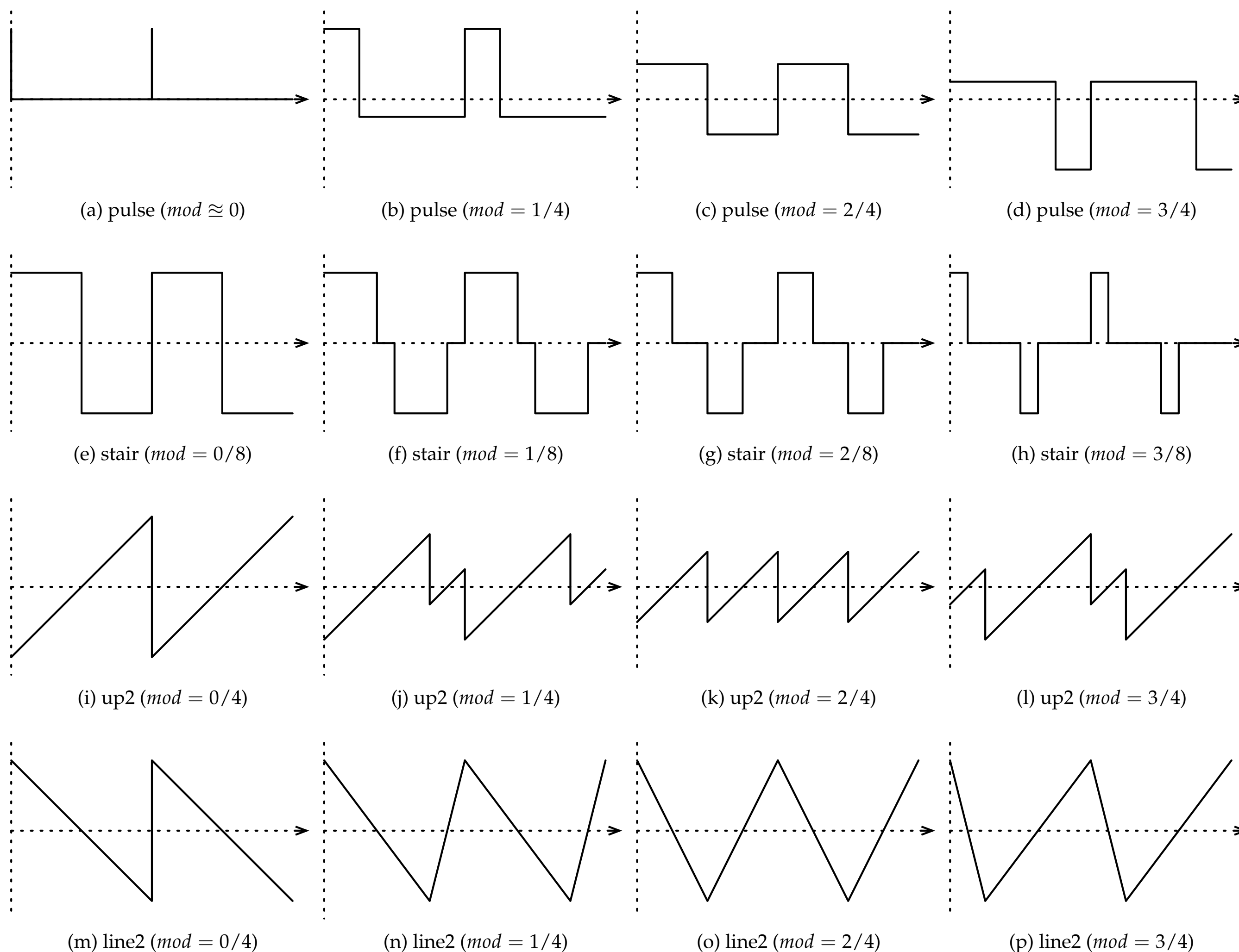
`Saw` generates a band-limited saw wave.

`Square` generates a band-limited square wave.

`DSF` (discrete summation formula) produces a band-limited harmonic series with variable amplitude and frequency ratios. The amplitude ratio determines the amplitude scaling factor of harmonics as frequency increases. The frequency ratio determines the spacing between harmonics with respect to the fundamental frequency.

Sample Player

[generator/player]



(a) pulse ($mod \approxeq 0$) (b) pulse ($mod = 1/4$) (c) pulse ($mod = 2/4$) (d) pulse ($mod = 3/4$)

(e) stair ($mod = 0/8$) (f) stair ($mod = 1/8$) (g) stair ($mod = 2/8$) (h) stair ($mod = 3/8$)

(i) up2 ($mod = 0/4$) (j) up2 ($mod = 1/4$) (k) up2 ($mod = 2/4$) (l) up2 ($mod = 3/4$)

(m) line2 ($mod = 0/4$) (n) line2 ($mod = 1/4$) (o) line2 ($mod = 2/4$) (p) line2 ($mod = 3/4$)

Figure 3: `LFO` dynamic waveforms (two cycles shown)

`Player` is used for playback and looping of sound samples. In the simplest scenario, its constructor takes a path to a sound file. Template parameters can be used to specify its interpolation type and looping mode. If the playback rate will not change, it is best to use no interpolation.

Noise (NoiseWhite, NoisePink, NoiseBrown)

[generator/noise]

The noise generators produce three different spectra classified by their falloff amount. `NoiseWhite` falls off at 0 dB/octave (uniform), `NoisePink` falls off at 3 dB/octave, and `NoiseBrown` falls off at 6 dB/octave.

Envelopes (Decay, AD, Seg, SegExp)

[generator/(envDecay, envAD, envSeg, envSegNoise, envSegExp)]

`Decay` produces an exponentially decaying envelope.

`AD` generates a two segment attack-decay envelope with exponential curvature. The curvature, c, of each segment can be specified where $c < 0$ is concave, $c = 0$ is linear, and $c > 0$ is convex.

`Seg` generates an envelope by interpolating between a sequence of breakpoints. A `Seg` can be used to downsample another generator, such as a noise generator, to produce a more slowly varying curve. The following example shows how to create a noise signal made of line segments:

```
Seg<float, iplSeq::Linear> seg(1.);
```

```
NoiseWhite<> noise;

// loop
seg(noise);
```

`SeqExp` generates an envelope with variable exponential curvature between two breakpoints.

3.3 Filters

[filter/freqResponses]

IIR (OnePole, Biquad, BlockDC)

[filter/(onePole, biquad, blockDC)]

An IIR (infinite impulse response) filter produces new values from a linear combination of its previous inputs and outputs. IIRs can produce sharp and "ringing" frequency responses more efficiently than FIRs. Their main disadvantage is that the have a non-linear phase response which can smear sharp transients in the input signal if one is not careful.

`OnePole` is a single pole filter that is useful for smoothing and integrating signals.

`Biquad` is a general-purpose resonant filter that can operate in low-pass, band-pass, high-pass, or notch modes. The default `Biquad` filter uses a Butterworth design for a maximally flat magnitude response. The resonance amount controls the "presence" or steepness of the filter at its cut-off frequency.

`BlockDC` is used for eliminating DC bias from signals. Sometimes very low frequency components will sneak into a signal, i.e., from positive feedback loops, which can create unwanted distortion due to clipping on the DAC. A `BlockDC` can be put at the end of the signal chain to eliminate these low-frequency components without coloring the rest of the signal.

Delays (Delay, Comb)

[filter/(delay, comb), effects/flanger]

`Delay` is an all-pass filter whose only effect is to delay its input. The delay time can be varied dynamically as there are several interpolation modes possible— `ipl::Linear`, `ipl::Cubic`, and `ipl::AllPass`. If the delay time is not to be varied, then it is best to use either the `ipl::Trunc` or `ipl::AllPass` modes as they will not color the magnitude spectrum. `Delay` has a `write()` method to place a sample at the beginning of the delay-line and a `read()` method that returns samples in the delay-line at specified delay amounts.

`Comb` is a delay-line with variable feedforward and feedback. Feedforward creates notches in the spectrum while feedback creates resonant peaks. The delay time can be varied in a smooth way as with `Delay`.

Hilbert transform

[filter/HilbertFilter]

`Hilbert` produces a phase quadrature (complex) signal from a real signal.

3.4 Spectral Processing

Discrete Fourier Transform (DFT)

[transform/(RFFT, CFFT)]

There are two classes for performing the discrete Fourier transform, `RFFT` for real-to-complex transforms and `CFFT` for complex-to-complex transforms. The DFT can be any size, however, it will perform best when the size is a product of small primes (e.g., powers of two). Transforms occur in-place meaning that the input and output sequences of the transform use the same memory. Single- and double-precision floating point are supported.

`RFFT` example:

```
RFFT<float> dft(N);
float samples[N];           // real time/position samples

dft.forward(samples);       // perform forward transform to complex frequency domain
samples[0];                 // DC real component
samples[1];                 // 1st harmonic real component
samples[2];                 // 1st harmonic imaginary component
samples[3];                 // 2nd harmonic real component
samples[4];                 // 2nd harmonic imaginary component
...
samples[N-1];               // Nyquist real component

dft.inverse(samples);       // transform back to real time/position domain
```

CFFT example:

```
CFFT<float> dft(N);
Complex<float> samples[N];  // complex time/position samples

dft.forward(samples);       // perform forward transform to complex frequency domain
samples[0];                 // complex harmonic +0
samples[1];                 // complex harmonic +1
...
samples[N-2];               // complex harmonic -2
samples[N-1];               // complex harmonic -1

dft.inverse(samples);       // transform back to complex time/position domain
```

Short-time Fourier Transform (STFT)

[transform/STFT]

The short-time Fourier transform uses overlapping windows to obtain better temporal resulotion and to avoid discontinuities between analysis windows during resynthesis. The STFT object can operate in a special stream mode which hides buffering and allows easy insertion into a sample loop.

```
// setup
STFT stft(
    2048,           // Window size
    2048/4,         // Hop size
    0,              // Pad size
    WinType::Hann,  // Window type: Bartlett, Blackman, BlackmanHarris,
                    //  Hamming, Hann, Welch, Rectangle
    Bin::Rect       // Format of frequency samples
);

// time domain loop
{
    float s = src();

    // stream in samples until we have enough for a DFT
    if(stft(s)){

        // frequency domain loop
        for(int k=0; k<stft.numBins(); ++k){
            stft.bins(k);   // the kth frequency sample
        }
    }

    // stream out sample from overlap add resynthesis
    s = stft();
}
```

3.5 Effects

Chorus

`Chorus` emulates multiple voices "singing" in unison (as in a choir) from a single source. Chorusing is useful for thickening up otherwise dry and monotonous sounds. `Chorus` operates by mixing its input with the input sent through two frequency modulated comb filters in parallel.

Waveshaping

Waveshaping is a process whereby a signal is mapped through a mathematical function. In many cases, a function table is used for efficiency and flexibility, but other times a direct function evaluation (such as raising to a power) is more appropriate.

Frequency shifting

`FreqShift` translates the spectrum of a signal. Frequency shifting preserves harmonic frequency distances, but not ratios. In contrast, pitch shifting preserves harmonic ratios, but not frequency distances.

4 Synthesis Examples

4.1 Additive Synthesis

[generator/(oscAddBeat, oscAddPacket)]

Additive synthesis is a technique where multiple sinusoids are summed together possibly with time-varying amplitudes and frequencies. In the simplest scenario, additive synthesis can be used to construct "beats" which emerge when two sinusoids with nearly equal frequency and amplitude are summed together. The rate of the beating is one-half the difference of the frequencies of the two component sinusoids. An example of more complex usage of additive synthesis is to resynthesize data obtained from sinusoidal analysis, such as a from a tracking phase vocoder.

4.2 Subtractive Synthesis

Subtractive synthesis is a technique where a sound source rich in harmonics, an excitation, is sent through a spectral filter having the effect of "subtracting" out harmonics. Subtractive synthesis is often used in classic analog synthesizers and for voice synthesis.

4.3 Amplitude Modulation

[generator/modAmp]

Amplitude modulation is a technique where one signal, the carrier, is multiplied by another, the modulator. In most cases, the carrier has an audible pitch, but the rate of the modulator can vary and determines whether to change the carrier's envelope (slow modulation) or timbre (fast modulation). The spectrum resulting from two multiplied signals is the sum and differences between all frequencies in each signal.

4.4 Frequency Modulation

[generator/(modFreqTransition, modFreqCM)]

Frequency modulation is a technique where the phase or frequency of one sinusoid, the carrier, is modulated by another, the modulator. At slow modulation rates, vibrato is produced. At fast modulation rates, a change in timbre is heard. The position of the frequency components depends on the ratio of the carrier frequency to modulator frequency. The amplitudes vary according to Bessel functions of the first kind and nth order, $J_n(x)$.

4.5 Physical Modeling

[filter/pluck]

A realistic sounding and efficient plucked string physical model is obtained through the Karplus-Strong algorithm. It operates by feeding a brief noise burst into a delay-line and recirculating it with an averaging filter in the feedback loop.

5 Further Reading

1. Mathews, M. (1969). The Technology of Computer Music. The M.I.T. Press, Boston.
2. Oppenheim, A. V. and Schafer, R. W. (1999). Discrete-time Signal Processing. Prentice Hall, New Jersey, second edition.
3. Roads, C. (1996). Computer Music Tutorial. MIT Press.
4. Smith, S. W. (2006). The Scientist and Engineer's Guide to Digital Signal Processing. California Technical Publishing.
5. Steiglitz, K. (1996). A Digital Signal Processing Primer. Addison-Wesley Publishing Company, Inc.