The Sound Object Library Reference Manual

Victor Lazzarini

Music Technology Laboratory
National University of Ireland, Maynooth

Version 2.6.1

Preface

The SndObj sound object library started off in 1997 as a research project at the Universidade Estadual de Londrina, in South Brazil, funded by the Brazilian Research Council, CNPq. The project was developed at the Nucleo de Musica Contemporanea, by Victor Lazzarini, assisted by Fernando Accorsi. The first implementations of the library were developed under the IBM Risc2000 (AIX, with g++) and PC (Windows95, with Visual C++ and g++) platforms. In 1998, Dr Lazzarini continued the work at the National University of Ireland, Maynooth, where the library was tested on Solaris, Linux and IRIX. In 2001, version 2.00 was developed, as part of a major re-assessment of the code and aspects of design of the library, which include a rationalisation of the SndIO (input/output) class tree, improved code and vectorial processing. These changes have had a great impact on performance, but the new features also rendered the new library incompatible with previous versions. Nevertheless, only simple editing is required to make application code compatible with the new library definitions. The Windows, Linux (with Open Sound System, OSS or ALSA and Jack), Mac OSX and IRIX implementations feature realtime audio, which makes the SndObj library a very powerful toolkit for DSP research and development. This reference manual is intended to facilitate the use of the library for researchers and musicians.

The Sound Object Library is an object-oriented audio processing library. It is a collection of classes for synthesis and processing of sound. These can be used to build applications for computer-generated music. The source code is multi-platform and can be compiled under any C++ compiler. POSIX compliance (and the presence of the pthread library) is necessary for the SndThread class. The Cygwin GNU C++ compiler supports this feature on Windows and the winpthread library is also available for other compilers (Mac OSX and UNIXs are POSIX-compliant). The cygwin compiler is freely available at http://ww.cygnus.com/cygwin.

Acknowledgements

I would like to thank the Music Department of National Unversity of Ireland, Maynooth, for its support of this and related research. Thanks also to colleagues from the Computer Science Department, Tom Lysaght and Joe Timoney for their help and encouragement. Part of this work was supported with funds from the NUI New Researcher Award.

Copyright Notice

License Notice

This software is licenced under the following terms below.

GNU GENERAL PUBLIC LICENSE
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannotimpose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems
or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will beguided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR

PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

Table of Contents

SndObj Programming Concepts

What is a SndObj?

A SndObj (pronounced 'Sound Object') is a programming unit that can generate signals with audio or control characteristics. It has a number of basic attributes, such as an output vector, a sampling rate, a vectorsize and an input connection (which points to another SndObj). Depending on the type of SndObj, other attributes will also be featured: an oscillator will have an input connection for a function table, a delayline will have a delay buffer, etc..

SndObjs contain their own output signal. So, at a given time, if we want to obtain the signal it generates, we can probe its output vector. This will contain a vecsize number of samples that the object has generated after it was asked to either process or synthesise a signal. This is a basic characteristic of SndObjs: signals are internal, as opposed to existing in external buffers or busses. SndObjs can interface very easily with external signals, but in a pure SndObj processing chain, signals are internal and hidden.

Generating output

The basic operation that a SndObj performs is to produce an output signal. This is done by invoking the public member function SndObj::DoProcess(). Each call will generate a new output vector full of samples, so to generate a continuous signal stream, the DoProcess() should be invoked repeatedly in a loop (known as the 'processing loop'). Programs will have to feature at least one such loop in order to generate audio signals.

As an alternative to directly programming a loop, users can avail of the SndThread services, which provide processing thread management and a hidden processing loop (see below). The DoProcess() method is overridable, so each different variety of SndObj will implement it differently so that different objects can generate different signals. In addition, other types of processing might be achieved with some overloaded operators (see below in 'Manipulating SndObjs').

Connecting SndObjs

Another basic programming concept found in this library is that SndObjs do not have direct signal inputs, because of the fact that signals are internal to them. Instead, they will have input connections to other SndObjs. So an object will read the output signal of another which is connected to it. Connections are made in the form of pointers (addresses) of SndObjs. So any type of signal input, either a processing input or a parameter modulator input is connected in the same way.

Certain processing parameters will then have two types of input: an *offset value* and a *SndObj connection*. The offset value, generally a single floating point value is added to whatever signal the connected SndObj has generated. In most cases, SndObj connections for parameters are optional: if they are not present, then only the offset value is used for it. In this case, they are in fact, not an 'offset', but the actual value for the parameter.

Manipulating SndObjs

Apart from invoking processing, users can manipulate SndObjs in other ways. The first obvious operation is parameter setting, for which different varieties of SndObjs will have different methods. However, an unified message-passing interface is defined by SndObj, with the SndObj::Set() and SndObj::Connect() methods. These can be used to change the status of SndObjs via the various messages defined for them. Messages are also inherited, so the derived object will have its own set, plus the ones defined for its superclass(es). Set() is used to set offset and single parameter values. Connect() is used to connect input objects, which can be of SndObj, SndIO (input and output objects) or Table (function table objects) types. Messages are C string constants.

Other simple operations which will modify the output of a SndObj can be made, such as the ones defined by the operators +, -,* , = , << etc.. Also the output signal buffer can be accessed with a variety of methods such as SndObj::Output(), SndObj::PushIn(), SndObj::PopOut(), etc.

Input and Output

Signal input and output is handled by SndIOs ('sound ios'), which are objects that can write and read to files, memory, devices, etc. They are modelled in similar ways to SndObjs: signals are internal, use object connections, etc.. However, they are designed to deal with a slightly different type of processing. Their main performing methods are SndIO::Read() and SndIO::Write(). When invoked, these will read or write a vectorsize full of samples from/to their source/destination, respectively. SndIOs can handle multichannel streams, so their output vector contains actually frames of samples (in interleaved format).

SndIOs interact with SndObjs in two basic ways. For signal input, SndIOs can be accessed via SndIn objects. Each channel of input audio has to be connected separately, because SndObjs in general handle only single signal streams. For signal output, SndObjs can be connected directly to SndIOs (again, one for each channel). This can be done at construction time, or more usually using SndIO::SetOutput(). SndIO input and output can also be performed more directly with the SndObj << and >> operators. For MIDI input, a number of specialist classes exist, derived from MidiIn, which work in a similar way to SndIn.

Function Tables

Certain SndObjs, for instance oscillators, will depend heavily on tabulated function tables. For this purpose, a special type of object can be used, a Table object. Tables are very simple objects whose most important attribute is their actual tabulated function, which is created at construction time. Tables can be updated at any time, by changing some of their parameters and invoking Table::MakeTable().

Frequency-domain issues

The Sound Object Library provides classes for time and frequency-domain (spectral) processing. For the latter, a few special considerations must be made. Time-domain and spectral SndObjs are designed to fit in together very snuggly in a processing chain. For this reason, a certain model was employed, which slightly limits the arrangement of such SndObjs.

For spectral processing, the FFT size must be always power-of-two multiple of the hopsize (usually a minimum four times that value). When connecting time- and frequency-domain SndObjs, the hopsize must be the same as the time-domain vectorsize. Generally for an efficient FFT, the analysis size is set to a power-of-two value. So, in practice, this limits the vectorsize/hopsize and FFT size values to a limited pairing of values. Although at first this looks limiting, it will in fact have little impact of the flexibility of spectral processing using the library. This model, in turn, will facilitate immensely the interaction between frequency- and time-domain SndObjs. Effectively, if this conditions are met, they can be inter-connected transparently, even though they are dealing with very different types of signals.

Processing threads

In addition to the basic types of objects discussed above, the Sound Object Library also includes a special thread management class, SndThread. With this type of object, a pthread-library based thread can be instantiated and run. This object encapsulates the main processing loop, calling the basic performing methods of each object that has been add to it.

Using SndThreads ('sound threads') is very simple. Once an object has been created and a chain of SndObjs/SndIOs has been defined, a processing list is initialised using SndThread::AddObj() or SndThread::Insert(). To start processing a signal,

SndThread::ProcOn() is invoked. To stop processing, SndThread::ProcOff() can be used. SndObjs can be deleted from the processing list using SndThread::DeleteObj(). Multiple SndThreads can be used for parallel processing with SndBuffer objects being used to obtain the signals from each thread.

Library Classes

CLASSES – SUBCLASSES	DESCRIPTION
SndObj	base class
ADSR	envelope generator
IADSR	extended envelope generator
Balance	balancer/envelope follower
Buzz	band-limited pulse generator
Convol	FFT-based convolution
DelayLine	delay line
Comb	comb filter
Allpass	allpass filter
FIR	direct convolution
Pitch	pitch shifter
SndLoop	sampler/looper
StringFlt	string resonator
Pluck	plucked-string generator
Tap	delay tap
Tapi	interpolated delay tap
VDelay	variable delay line
FastOsc	power-of-two table oscillator
Osc	truncating oscillator
Osci	interpolating oscillator
FFT	short-time FFT
PVA	phase vocoder analysis
IFGram	instantaneous frequency analysis
Filter	fixed resonator
HiPass	1st order high-pass
LoPass	1st order low-pass
Lp	2nd order resonating low-pass
Reson	2nd order band-pass resonator
TpTz	general-purpose 2nd order filter
Ap	2nd order allpass
ButtBP	Butterworth-response band-pass
ButtBR	Butterworth-response band-reject
ButtHP	Butterworth-response high-pass
ButtLP	Butterworth-response low-pass
Gain	gain attenuation/boost
Hilb	Hilbert transform
IFFT	short-time IFFT
PVS	phase vocoder synthesis
PVRead	phase vocoder file reader
Interp	curve segment generator
Lookup	table lookup
Lookupi	interpolated table lookup
MidiIn	MIDI input
Bend	pitch bender
MidiMap	MIDI input mapping
Mixer	adder (signal mixing)
Oscil	basic oscillator
Oscili	interpolating oscillator
PhOscili	phase-mod interpolating oscillator
Oscilt	truncating oscillator
Pan	stereo panning
Phase	phase increment generator

Rand	noise
Randh	band-limited noise (sample & hold)
Randi	band-limited noise (interpolating)
Ring	general purpose multiplier
SinAnal	sinusoid analysis
SinSyn	cubic-interpolation additive synthesis
ResSyn	**cubic-interpolation (timescal/frq control)**
AdSyn	linear-interpolation additive synthesis
IFAdd	**additive synthesis from bin frames**
SndIn	audio input
SndRead	sound file reader
SpecIn	spectral file input
SpecMult	complex multiplication of spectra
PVBlur	blurring of PV spectral data
PVMix	mixing of PV spectral data
PVTransp	pitch transposition of PV spectral data
SpecCart	cartesian conversion of spectra
SpecCombine	phase & mag combiner
SpecInterp	spectral interpolation
PVMorph	phase vocoder morphing
PVMask	phase vocoder masking
PVFilter	phase vocoder filtering
SpecPolar	polar conversion of spectra
SpecSplit	phase & mag splitter
SpecThresh	thresholding
SpecVoc	cross-synthesis of spectra
SyncGrain	granular synthesis
Unit	unit/ramp generator

SndIO — **input/output base class**

SndASIO	ASIO audio
SndBuffer	signal buffering
SndFIO	raw soundfile IO
SndWave	RIFF-Wave soundfile IO
SndWaveX	RIFF-WaveX soundfile IO
SndPVOCEX	PVOCEX spectral file IO
SndSinIO	SINUSEX spectral file IO
SndAiff	AIFF soundfile IO
SndMidi	MIDI IO specs
SndMidiIn	MIDI input
SndRTIO	realtime IO (ADC/DAC)
SndCoreAudio	CoreAudio realtime IO
SndJackIO	Jack Connection Kit IO

Table — **function-table base class**

EnvTable — **envelope function table**

HammingTable	inverted-raised cosine windows
HarmTable	harmonic functions
ImpulseTable	FIR filter coefficients (impulse response)
LoPassTable	low-pass FIR coefficients

NoteTable	MIDI to Hz conversion
PlnTable	polynomials
PVEnvTable	PV-format spectral envelope
SpecEnvTable	complex-format spectral envelope
PVTable	PV-analysis table
TrisegTable	three-segment functions
SndTable	soundfiles
UsrDefTable	user-defined
UsrHarmTable	harmonic functions (user-definable)

SndThread	sound processing thread management

Class ADSR

Description

This object generates an **attack - decay - sustain - release** shaped signal at the output. Alternatively, it can similarly shape an input signal, acting as a modifier. Other parameters to it are max amplitude and total duration of the envelope period, after which the whole cycle is repeated.

Construction

ADSR()
ADSR(**float** att, **float** maxamp, **float** dec, **float** sus, **float** rel, **float** dur, **SndObj*** InObj = 0,
int vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Public Methods

parameter/state setting:
void SetMaxAmp(**float** maxamp)
void SetADSR(**float** att, **float** dec, **floa**t sus, **float** rel)
void SetDur(**float** dur)

envelope stage control:
void Release()
void Sustain()
void Restart()

Messages

[set] “attack”
[set] “decay”
[set] “sustain”
[set] “release”
[set] “maxamp”
[set] “duration”
[set] ”go to release”
[set] ”lock to sustain”
[set] “restart”

Details

construction

ADSR()
ADSR(**float** att, **float** maxamp, **float** dec, **float** sus, **float** rel, **float** dur, **SndObj*** InObj = 0,
int vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

ADSR objects can be constructed either by the default constructor (which resets the envelope parameters to 0) or by the full constructor. Its arguments are:

float att: attack time, or rise time, in secs. Time taken for the signal to change from 0.f to *maxamp*.
float maxamp: maximum amplitude after rise time. It is a multiplier, in case of the shaping of an input signal.
float dec: decay time, in secs. Time taken for the signal to change from *maxamp* to *sus*.
float sus: sustain amplitude after decay time. Again, a multiplier, in case of envelope shaping. The sustain period is calculated on the basis of the difference between the total duration and the sum of the attack, decay and release times.

float rel: release time, in secs, after the sustain period, during which the signal changes from *sus* back to 0.f. It is calculated backwards from the end, taken from the total duration of the envelope.
float dur: total duration of the envelope, in secs, or the envelope period. This ADSR is designed to loop, so the whole shape will be repeated after the total duration is completed.
SndObj* InObj: input object, pointer to the location of a SndObj-derived object. Defaults to 0, which means *no input object*, so the ADSR object is used as a signal generator.
int vecsize: vector size in samples. Size of the internal DSP buffer, defaults to DEF_VECSIZE, 256.
float sr: sampling rate in HZ. Defaults to DEF_SR, 44100.f.

public methods

void SetMaxAmp(**float** maxamp)
void SetADSR(**float** att, **float** dec, **floa**t sus, **float** rel)
void SetDur(**float** dur)

These methods are used to set the different parameters that make up the ADSR envelope. Their intepretation is the same as found in the constructor.

void Release()
void Sustain()

These two methods control the sustain period. When **Sustain()** is called, the ADSR locks into the sustain phase when the envelope reaches it. **Release()** makes it immediately jump into the release phase, from anywhere in the envelope. These methods are designed for realtime applications, making the envelope respond to MIDI or other types of control. The envelope breakpoints behave normally if these methods are not invoked.

void Restart()

This method resets the internal envelope count, effectively making it restart from the attack phase. Its main application is to provide a way of controlling the envelope in realtime, when used in conjuction with **Release()** and **Sustain()**. The ADSR is set to restart automatically when the four phases are completed (normal behaviour when **Release()/Sustain()** are not invoked).

Examples

ADSR is usually constructed by setting the envelope parameters in the constructor. The example below creates an ADSR object which will shape the output of a previously declared object named oscillator for the period of 1 sec.

ADSR envelope(.01f, 16000.f, .2f, 12000.f, .05f, 1.f, &oscillator);

If an object input is not given, ADSR works as signal generator. This generates a signal that has a period of 4 secs, attack of 0.01, decay of 0.02, sustaining at 8000 for 3.87 secs and decaying for 0.1 secs.

ADSR envelope(.01f, 1000.f, .02f , 8000.f, .1f, 4.f);

Audio processing is performed by repeatedely invoking the DoProcess() (as in all SndObj-derived classes). For details on DoProcess() consult the manual page on Class SndObj. Placing DoProcess() in a loop (or using a SndThread object) will achieve this:

```
while(processing_on){

oscillator.DoProcess();
envelope.DoProcess();
```

```
output.Write();

}
```

Calling Restart(), Sustain() and Release() will control the envelope in realtime:

```
while(processing_on){

 if(noteon) {
  envelope.Restart();
  envelope.Sustain();
}
 if(noteoff)  envelope.Release()

oscillator.DoProcess();
envelope.DoProcess();
output.Write();

}
```

Class AdSyn

Description

The AdSyn class implements sinusoidal additive resynthesis, based on a standard linear interpolation algorithm. The class takes an input from a SinAnal-type class, which consists of a series of tracks containing amplitude, frequency and phase information. Tracks are identified by IDs given by the input object, which are then used to match them between hop periods. AdSyn objects can resynthesise any number of tracks up to the maximum tracks found at their input. AdSyn can modify the timescale and pitch of the resynthesis.

Construction

AdSyn()
AdSyn(**SinAnal*** input, **int** maxtracks, **Table*** table, float pitch=1.f, **float** scale=1.f,
int vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Public Methods

void SetPitch(**float** pitch)

Messages

[set] **"pitch"**

Details

construction

construction

AdSyn()
AdSyn(**SinAnal*** input, **int** maxtracks, **Table*** table, **float** pitch=1.f, **float** scale=1.f,
int vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

These methods construct a AdSyn object:

SinAnal* input: input object object, of the SinAnal type, from which the tracks to be resynthesised will be read.
int maxtracks: max resynthesis tracks, should be <= tracks generated by the input object.
Table* table: table object containing a wavetable to be used by each oscillator in the resynthesis, typically a cosine wave.
float pitch: pitch ratio used in the resynthesis (1.0 = unaltered).
float scale: amplitude scaling of output.
int vecsize: object vector size, also determines the synthesis hopsize between analysis frames (defaults to 256).
float sr: sampling rate in Hz (defaults to 44100).

public methods

void SetPitch(**float** pitch)

Examples

The following connections are a simple example of the use of AdSyn to resynthesise track data generated by SinAnal:

```
HarmTable table(4000, 1, 1, 0.75);    // cosine wave
```

```
HammingTable window(fftsize, 0.5); // hanning window

// input sound
SndWave input(infile,READ,1,16,0,0.f,decimation);
SndIn   insound(&input, 1, decimation);

// IFD analysis
IFGram ifgram(&window,&insound,1.f,fftsize,decimation);
// Sinusoidal analysis
SinAnal sinus(&ifgram,thresh,intracks);
// Sinusoidal resynthesis
AdSyn  synth(&sinus,outracks,&table,pitch,scale,interpolation);

// output sound
SndWave output(outfile, OVERWRITE,1,16,0,0.f,interpolation);
output.SetOutput(1, &synth);
```

This code takes an input sound, from a file and passes it through the analysis process and then the data is resynthesized. The timescale change is determined by the decimation:interpolation ratio. The pitch of the resynthesis is determined by the variable **pitch**. In order to implement processing, the programmer either needs to write a loop and call the reading/writing and processing methods, or use a SndThread object, passing these objects to it. This example is based on a simple modification of src/examples/sinus.cpp.

Class Allpass

Description

The allpass object implements an allpass filter. It recirculates a signal through an allpass network, rescaled by a feedback/feedforward gain factor. Its parameters are delay (loop) time, gain and input object.

Construction

Allpass()
Allpass(**float** gain, **float** delaytime, **SndObj*** InObj = 0, **int** vecsize=DEF_VECSIZE,
float sr=DEF_SR)

Details

construction

Allpass()
Allpass(**float** gain, **float** delaytime, **SndObj*** InObj = 0, **int** vecsize=DEF_VECSIZE,
float sr=DEF_SR)

Allpass objects can be created using either constructor. The default constructor sets all parameters to 0. The full constructor arguments are:

float gain: gain factor, which will rescale the signal before it re-enters the delay line. Normally < 1, anything over 1 will cause the signal to continually grow, with possibly disastrous results.
float delaytime: delay time, in seconds.
SndObj* InObj: input object, pointer to the location of a SndObj-derived object.
int vecsize: vector size in samples. Size of the internal DSP buffer, defaults to DEF_VECSIZE, 256.
float sr: sampling rate in HZ. Defaults to DEF_SR, 44100.f.

Examples

The Allpass class is derived from Comb, which in its turn is derived from DelayLine, so all messages and public methods defined for those classes are also available to objects of this class. This creates an allpass with gain of 0.5 and a delay loop of 10ms, processing an input object *inobj*:

Allpass ap(.5f, .01f, &inobj);

The main processing method, as with all SndObj classes, is DoProcess(), overriden to implement an allpass network. It is usually called within a loop (if SndThread is not being used) creating a vector of samples in the output.

```
while(processing_on){

inobj.DoProcess();
ap.DoProcess();
output.Write();

}
```

Class Balance

Description

Objects of this class balance two signals. They measure the rms power of the second input signal (the control signal), by a combination of rectification and low-pass filtering. That information is used to control the gain of the first signal. This signal is then fed to the output of the object.

Construction

Balance()
Balance(**SndObj*** input1, **SndObj*** input2, **float** fr=10.f, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Public Methods

void SetInput(**SndObj*** input1, **SndObj*** input2)
short SetLPFreq(**float** fr)

Messages

[set] **"lowpass frequency"**
[connect] **"comparator"**

Details

construction

Balance()
Balance(**SndObj*** input1, **SndObj*** input2, **float** fr=10.f, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Objects can be created using the default constructor **Balance()**, where the state is set to the default values. The full constructor has the following arguments:

SndObj* input1: input object 1, pointer to the location of a SndObj-derived object. This is the signal input, which will be balanced to the other input. It is set to 0 by the default constructor.
SndObj* input2: input object 2, pointer to the location of a SndObj-derived object. This is the control input, which will be used to balance the other signal. It is set to 0 by the default constructor.
float fr: cut-off frequency of the internal low-pass filter. Defaults to 10.
int vecsize: vector size in samples. Size of the internal DSP buffer, defaults to DEF_VECSIZE, 256.
float sr: sampling rate in HZ. Defaults to DEF_SR, 44100.f.

public methods

void SetInput(**SndObj*** input1, **SndObj*** input2)

This method connects the two inpus to this object. Input1 is the signal input, whereas input2 is the comparator. They are both pointers to objects that generate these signals.

short SetLPFreq(**float** fr)

This method sets the internal lowpass filter frequency which is used in the rms estimation process.

Examples

A Balance object is used to force a signal to keep to the same rms level as a comparator signal. It is usually created by passing the pointers to the two inputs to the constructor:

Balance balan(&inobj, &compobj);

One of its uses is to make the output of a filter as loud (or as quiet) as the input:

```
Filter fil(1000.f, 1.f, &inobj);
Balance bal(&fil, &inobj);

while(processing_on){

inobj.DoProcess();
fil.DoProcess();
bal.DoProcess();
output.Write();

}
```

Class Bend

Description

This object reads pitchbend messages from one MIDI channel from a SndMidiIn object and bend an input signal by an specified proportional amount. This modifies the signal values by a set percentage according to the pitchbend amount and the specified range.

Construction

Bend()
Bend(**SndMidiIn*** input**, SndObj** *inObj, **float** range, **short** channel=1, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Public Methods

void SetRange(**short** channel)

Messages

[set] **"range"**

Details

construction

Bend()
Bend(**SndMidiIn*** input**, SndObj** *inObj, **float** range, **short** channel=1, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Bend objects are constructed with the following parameters:

SndMidiIn* input: pointer to the location of a SndMidiIn object (midi message source).
SndObj* inObj: pointer to the location of an input SndObj-derived object (signal source)
float range: percentage of positive/negative change in the input signal. For example, 10.f will indicate 10% change when the pitchbend message received is max/min. A value of a 100.0 will be "bent" to a maximum of 110.0 or a minimum of 90.0
short channel: MIDI channel from which a message will be read. Defaults to channel 1.
int vecsize: vector size in samples. Size of the internal DSP buffer, defaults to DEF_VECSIZE, 256.
float sr: sampling rate in HZ. Defaults to DEF_SR, 44100.f.

public methods

void SetRange(**short** channel)

This method sets the percentage range of the pitchbend change, as discussed above.

Examples

Bend objects are usually used to bend frequency signals by a set percentage, but they can be used for any other purpose. A Bend object is created by passing it a SndMidiIn object pointer (which reads the MIDI input) and a signal object pointer which will generate the signal to be modified:

Bend pitchbend(&midiobj, &noteobj, 12.5f);

Here a SndMidiIn object *midiobj* and a SndObj (or derived) *noteobj* are passed to *pitchbend*, which will bend the signal to a max/min of 12.5%. If noteobj is mapping MIDI notes to

frequency, then the signal out of pitchbend can be used to control the frequency of, say, an oscillator (call it *oscilla*):

```
while(processing_on){

midiobj.Read();
noteobj.DoProcess();
pitchbend.DoProcess();
oscilla.DoProcess();
output.Write();

}
```

Class ButtBP

Description

The ButtBP class is a TpTz-derived class that implements a Butterworth-response band-pass filter. These types of filters have a maximally flat pass-band, providing superior precision and stopband attenuation.

Construction

ButtBP()
ButtBP(**float** fr, **float** bw, **SndObj*** inObj, **SndObj*** inputfreq = 0, **SndObj*** inputbw = 0, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Public Methods

void SetFreq(**float** fr, **SndObj*** inputfr=0)
void SetBW(**float** bw, **SndObj*** inputbw=0)

Messages

[set, connect] **"frequency"**
[set, connect] **"bandwidth"**

Details

construction

ButtBP()
ButtBP(**float** fr, **float** bw, **SndObj*** inObj, **SndObj*** inputfreq = 0, **SndObj*** inputbw = 0, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

These methods construct an object of the ButtBP class. Constructor arguments are:

float fr: centre frequency offset, in Hz, default constructor sets it to 1000.f.
float bw: bandwidth offset, in Hz, default constructor sets it to 250.f.
SndObj* inObj: pointer to an input SndObj-derived object.
SndObj* inputfreq: frequency control input, pointer to the location of a SndObj-derived object. The centre frequency can be controlled by a time-varying signal from another SndObj-derived object. A signal is fed from the input object and added to the frequency offset value. Defaults to 0, *no frequency input object*
SndObj* inputbw: bandwidth control input, pointer to the location of a SndObj-derived object. The bandwidth can be controlled by a time-varying signal from another SndObj-derived object. A signal is fed from the input object and added to the bandwidth offset value. Defaults to 0, *no bandwidth input object*
int vecsize: vector size in samples. Size of the internal DSP buffer, defaults to DEF_VECSIZE, 256.
float sr: sampling rate in HZ. Defaults to DEF_SR, 44100.f.

public methods

void SetFreq(**float** fr, **SndObj*** inputfr=0)
void SetBW(**float** bw, **SndObj*** inputbw=0)

These methods set the centre frequency and bandwidth, respectively, of the filter implemented by the object. The float parameters are offsets (or fixed scalar values) and the SndObj pointers are the modulating signal inputs.

Examples

ButtBP objects implement filters that can be used to selectively eliminate certain frequencies and emphasize others. The wider the bandwidth, the less selective the filter is. Narrower bandwidths will suffer from poor time response. A ButtBP object is usually created by passing values for the centre frequency/bandwidth, an input signal object pointer, and optionally pointers to objects that produce a signal to modulate those parameters:

ButtBP filter(1000.f, 10.f, &inobj, &infreq);

Here, a signal produced by *inobj* is filtered by this object with a BW of 10 Hz and a frequency offset of 1000.f. The signal generated by *infreq* modulates the frequency of the filter (it is added to the offset).

```
while(processing_on){

inobj.DoProcess();
infreq.DoProcess();
filter.DoProcess();
output.Write();

}
```

Class ButtBR

Description

The ButtBR class is a ButtBP-derived class that implements a Butterworth-response band-reject filter. The band-reject filter has the reverse effect of the band-pass filter ButtBP.

Construction

ButtBR()
ButtBR(**float** fr, **float** bw, **SndObj*** inObj, **SndObj*** inputfreq = 0, **SndObj*** inputbw = 0, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Details

construction

ButtBR()
ButtBR(**float** fr, **float** bw, **SndObj*** inObj, **SndObj*** inputfreq = 0, **SndObj*** inputbw = 0, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

These methods construct an object of the ButtBR class. Constructor arguments are:

float fr: centre frequency offset, in Hz, default constructor sets it to 1000.f.
float bw: bandwidth offset, in Hz, default constructor sets it to 250.f.
SndObj* inObj: pointer to an input SndObj-derived object.
SndObj* inputfreq: frequency control input, pointer to the location of a SndObj-derived object. The centre frequency can be controlled by a time-varying signal from another SndObj-derived object. A signal is fed from the input object and added to the frequency offset value. Defaults to 0, *no frequency input object*
SndObj* inputbw: bandwidth control input, pointer to the location of a SndObj-derived object. The bandwidth can be controlled by a time-varying signal from another SndObj-derived object. A signal is fed from the input object and added to the bandwidth offset value. Defaults to 0, *no bandwidth input object*
int vecsize: vector size in samples. Size of the internal DSP buffer, defaults to DEF_VECSIZE, 256.
float sr: sampling rate in HZ. Defaults to DEF_SR, 44100.f.

Examples

ButtBR objects implement filters that can be used to selectively eliminate certain frequencies within a particular band. A ButtBR object is usually created by passing values for the centre frequency/bandwidth, an input signal object pointer, and optionally pointers to objects that produce a signal to modulate those parameters:

ButtBR filter(500.f, 100.f, &inobj);

Here, a signal produced by *inobj* is filtered by this object with a BW of 100 Hz and a frequency of 500.f. Components falling within this band will be attenuated/eliminated.

```
while(processing_on){
inobj.DoProcess();
filter.DoProcess();
output.Write();
}
```

Class ButtHP

Description

The ButtHP class is a ButtBP-derived class that implements a Butterworth-response high-pass filter. The band-reject filter has the reverse effect of the low-pass filter ButtLP.

Construction

ButtHP()
ButtHP(**float** fr, **SndObj*** inObj, **SndObj*** inputfreq = 0, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Details

construction

ButtHP()
ButtHP(**float** fr, **SndObj*** inObj, **SndObj*** inputfreq = 0, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

These methods construct an object of the ButtHP class. Constructor arguments are:

float fr: cutoff frequency offset, in Hz, default constructor sets it to 1000.f.
SndObj* inObj: pointer to an input SndObj-derived object.
SndObj* inputfreq: frequency control input, pointer to the location of a SndObj-derived object. The centre frequency can be controlled by a time-varying signal from another SndObj-derived object. A signal is fed from the input object and added to the frequency offset value. Defaults to 0, *no frequency input object*
int vecsize: vector size in samples. Size of the internal DSP buffer, defaults to DEF_VECSIZE, 256.
float sr: sampling rate in HZ. Defaults to DEF_SR, 44100.f.

Examples

ButtHP objects implement filters that can be used to selectively eliminate certain frequencies below a particular frequency. A ButtHP object is usually created by passing a value for the cutoff frequency, an input signal object pointer, and optionally a pointer to an object that produce a signal to modulate the frequency:

ButtHP filter(1000.f, &inobj);

Here, a signal produced by *inobj* is filtered by this object with a frequency of 1000.f. Components falling below this frequency will be attenuated/eliminated.

```
while(processing_on){
inobj.DoProcess();
filter.DoProcess();
output.Write();
}
```

Class ButtLP

Description

The ButtLP class is a ButtBP-derived class that implements a Butterworth-response low-pass filter. The band-reject filter has the reverse effect of the high-pass filter ButtHP.

Construction

ButtLP()
ButtLP(**float** fr, **SndObj*** inObj, **SndObj*** inputfreq = 0, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Details

construction

ButtLP()
ButtLP(**float** fr, **SndObj*** inObj, **SndObj*** inputfreq = 0, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

These methods construct an object of the ButtLP class. Constructor arguments are:

float fr: cutoff frequency offset, in Hz, default constructor sets it to 1000.f.
SndObj* inObj: pointer to an input SndObj-derived object.
SndObj* inputfreq: frequency control input, pointer to the location of a SndObj-derived object. The centre frequency can be controlled by a time-varying signal from another SndObj-derived object. A signal is fed from the input object and added to the frequency offset value. Defaults to 0, *no frequency input object*
int vecsize: vector size in samples. Size of the internal DSP buffer, defaults to DEF_VECSIZE, 256.
float sr: sampling rate in HZ. Defaults to DEF_SR, 44100.f.

Examples

ButtLP objects implement filters that can be used to selectively eliminate certain frequencies above a particular frequency. A ButtLP object is usually created by passing a value for the cutoff frequency, an input signal object pointer, and optionally a pointer to an object that produce a signal to modulate the frequency:

ButtLP filter(2500.f, &inobj);

Here, a signal produced by *inobj* is filtered by this object with a frequency of 2500.f. Components falling above this frequency will be attenuated/eliminated.

```
while(processing_on){
inobj.DoProcess();
filter.DoProcess();
output.Write();
}
```

Class Buzz

Description

This object is a broadband signal generator, implemented using a discrete summation formula. It basically generates a pulse wave with an user-specified number of harmonics. Other required parameters are fundamental frequency and amplitude, which can be controlled by the output of other SndObj-derived objects.

Construction

Buzz()
Buzz(**float** fr, **float** amp, **int** harm, **float** inputfr=0, **float** inputamp=0, **int** vecsize=DEF_VECSIZE. **float** sr=DEF_SR)

Public Methods

void SetHarm(**short** harms)
void SetFreq(**float** fr, **SndObj*** InFrObj=0)
void SetAmp(**float** amp, **SndObj*** InAmpObj=0)

Messages

[set, connect] **“frequency”**
[set, connect] **“amplitude”**
[set] **“harmonics”**

Details

construction

Buzz()
Buzz(**float** fr, **float** amp, **int** harm, **float** inputfr=0, **float** inputamp=0, **int** vecsize=DEF_VECSIZE. **float** sr=DEF_SR)

These methods construct an object of the Buzz class. Construction parameters are:

float fr: fundamental frequency offset, in Hz. Initialised to 440 by the default constructor.
float amp: amplitude offset. Initialised to 1.f .
short harms: number of harmonics. Initialized to 10.
SndObj* InFrObj: frequency control input, pointer to the location of a SndObj-derived object. The fundamental frequency can be controlled by a time-varying signal from another SndObj-derived object. A signal is fed from the input object and added to the frequency offset value. Defaults to 0, *no frequency input object*
SndObj* InAmpObj: amplitude control input, pointer to the location of a SndObj-derived object. Defaults to 0, which means *no amplitude input object*, so the amplitude is fixed to the amplitude offset value. This is added to the amplitude control input signal when a SndObj-derived object is patched in.
int vecsize: vector size in samples. Size of the internal DSP buffer, defaults to DEF_VECSIZE, 256.
float sr: sampling rate in HZ. Defaults to DEF_SR, 44100.f.

public methods

void SetHarm(**short** harms)
void SetFreq(**float** fr, **SndObj*** InFrObj=0)
void SetAmp(**float** amp, **SndObj*** InAmpObj=0)

These methods set the parameters that make up the state of a Buzz object. **SetHarm()** sets the number of harmonics present in the pulse wave.It has the side-effect of causing a discontinuity in the signal (perceived as an 'attack') because of the way the phases of the component oscillators are reset. **SetFreq()** and **SetAmp()** set the parameters relating to frequency and amplitude, respectively. The float arguments are offsets and the SndObj pointers are the input signal objects which will modulate the parameters.

Examples

A Buzz implements a band-limited pulse. It produces a harmonic-rich signal, which can be shaped by filters etc:

Buzz blp(100.f, 16000.f, 30);
Filter fil(1200.f, 50.f, &blp);

This example shows a Buzz object *blp* which generates a wave with a fundamental frequency of 100 Hz and 30 harmonics (100, 200, …, 3000). The Filter object *fil* shapes the signal with a reson-type band-pass shape centered around 1200 Hz.

```
while(processing_on){

blp.doProcess();
fil.DoProcess();
output.Write();

}
```

Class Comb

Description

Objects fo the class Comb implement a Comb filter. They recirculate a signal through a delay line, rescaled by a feedback gain factor. Their parameters are delay (loop) time, gain and input object.

Construction

Comb()
Comb(**float** gain, **float** delaytime, **SndObj*** InObj, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Public Methods

void SetGain(**float** gain)

Messages

[set] **"gain"**

Details

construction

Comb()
Comb(**float** gain, **float** delaytime, **SndObj*** InObj, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

These methods construct an object of the Comb class. Construction parameters are:

float gain: gain factor, which will rescale the signal before it re-enters the delay line. Normally < 1, anything over 1 will cause the signal to continually grow, with possibly disastrous results.
float delaytime: delay time, in seconds.
SndObj* InObj: input object, pointer to the location of a SndObj-derived object.
int vecsize: vector size in samples. Size of the internal DSP buffer, defaults to DEF_VECSIZE, 256.
float sr: sampling rate in HZ. Defaults to DEF_SR, 44100.f.

public methods

void SetGain(**float** gain)

This method sets the value for the feedback gain.

Examples

A comb filter basically recirculates a signal through a delay line, scaling it by a feedback gain factor. If this factor is below 1, then the signal will eventually die off after a certain time. Comb filters can be used as component reverberators to implement diffuse-field reverberation.

Comb rev(.9f, .1f, &inobj);

This creates a comb filter with gain of 0.9 and a delay loop of 100ms, processing an input object *inobj.*

while(processing_on){

inobj.DoProcess();

```
rev.DoProcess();
output.Write();

}
```

Class Convol

Description

The class Convol implements fast convolution using the fft of an impulse response and an input signal. The impulse response is taken from a Table object of any size, it is padded with zeros to the next power-of-two size and the fft is calculated. The input signal is transformed using the same size fft and the two spectra are multiplied together. The result is transformed back into time domain and successive frames are overlap-added. As a result the convolution output will have a delay of fftsize samples in relation to the input.

Construction

Convol()
Convol(**Table*** impulse, **SndObj*** input, **float** scale, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Public Methods

void SetImpulse(**Table*** impulse, **float** scale);

Messages

[set] **“scale”**
[connect] **“impulse”**

Details

construction

Convol()
Convol(**Table*** impulse, **SndObj*** input, **float** scale, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

These methods construct a Convol object. The arguments to the full constructor are:

Table *impulse: a pointer to a Table-derived object containing the impulse response.
SndObj * input: a pointer to a SndObj-derived object which will generate the signal to be transformed by this object.
float scale: a scaling factor to be applied to the impulse response. This can be used to boost/attenuate the impulse response gain, if necessary.
int vecsize: vector size in samples. Size of the internal DSP buffer, defaults to DEF_VECSIZE, 256.
float sr: sampling rate in HZ. Defaults to DEF_SR, 44100.f.

public methods

void SetImpulse(**Table*** impulse, **float** scale);

This method connects the Table-derived object containing the impulse response to this object and also sets the scaling factor for it.

Examples

The fast convolution process is often used with longer impulse responses. For shorter ones, the direct convolution (class FIR) will do. The process is reasonably fast, but a delay is introduced (as explained above) due to the fft process. A Convol object is constructed by passing a table object pointer holding the impulse, an input signal object and a scaling factor:

Convol cvlve(&impobj, &inobj, 16000.f);

This object will process an input object *inobj* using the impulse response in *impobj*. Tables are usually normalised, so that we will scale the impulse response to around –6dB (in 16-bit).

```
while(processing_on){

inobj.DoProcess();
cvlve.DoProcess();
output.Write();

}
```

Class DelayLine

Description

The DelayLine object is a simple delay processor. It delays a signal by an user-specified time. Its parameters are delay time and input object.

Construction

DelayLine()
DelayLine(**float** delaytime, **SndObj*** InObj, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Public Methods

void SetDelayTime(**float** delaytime)
void Reset()

Messages

[set] **“max delaytime”**

Details

construction

DelayLine()
DelayLine(**float** delaytime, **SndObj*** InObj, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

These methods construct an object of the DelayLine class. Construction parameters are:

float delaytime: delay time, in seconds.
SndObj* InObj: input object, pointer to the location of a SndObj-derived object.
int vecsize: vector size in samples. Size of the internal DSP buffer, defaults to DEF_VECSIZE, 256.
float sr: sampling rate in HZ. Defaults to DEF_SR, 44100.f.

public methods

void SetDelayTime(**float** delaytime)

This method sets the maximum delay time. It has the side-effect of clearing the delay buffer as it re-sizes it.

void Reset()

This method clears the delay buffer, setting all its samples to zero. It is called automatically by the constructor and when the delay line is resized.

Examples

The delay line simply delays a signal by a certain time. It can be used to create slap-back echo effects or to time-align a signal. It can also be tapped (by Tap/Tapi objects). It is usually created by setting a delay time and connecting an input into it:

DelayLine delay(0.5f, &inobj);

This connects the input object *inobj* to the DelayLine object. The output will be delayed by 0.5 seconds.

```
while(processing_on){

inobj.DoProcess();
delay.DoProcess();
output.Write();

}
```

Class EnvTable

Description

The EnvTable object builds a function table based on supplied envelope parameters. The envelope is defined by a starting point and two arrays: (1) segment legnths and (2) end points of each segment. The segments can be either exponential or linear. The table values are normalised.

Construction

EnvTable()
EnvTable(**long** L**,** **int** segments**,** **float** start**,**
float* points**,** **float*** lengths**,****float** type = 0.f)

Public Methods

void SetEnvelope(**int** segments, **float** start, **float*** points, **float*** lengths,
float)

Details

construction

EnvTable()
EnvTable(**long** L**,** **int** segments**,** **float** start**,**
float* points**,** **float*** lengths**,****float** type = 0.f**,**
float sr=44100.f**,****float** nyquistamp=0.f)

Constructs a EnvTable object.

long L: table length.
int segments: number of envelope segments.
float start: starting value of envelope.
float* points: an array of floats, containing the end values of each segment. Must match the above number of segments.
float* lengths: an array of floats, containing the lengths of each segment. Must match the above number of segments. Segment lengths are normalised to the table size (added up and then each one is divided by that total and multiplied by the table size).
float type: type of curve. Linear = 0, inverse exponential < 0 < exponential.

public methods

void SetEnvelope(**int** segments, **float** start, **float*** points, **float*** lengths,
loat type, **float** nyquistamp)

This method sets the envelope parameters. MakeTable() is invoked by this method.

Class FastOsc

Description

FastOsc implements a fixed truncating table oscillator which works exclusively with power-of-two size tables (for other table sizes see class Oscil).

Construction

FastOsc()
FastOsc(**Table*** table, **float** fr, **float** amp, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Public Methods

void SetFreq(**float** fr)
void SetAmp(**float** amp)
void SetPhase(**float** phase)
void SetTable(**Table*** table)

Messages

[set] **"frequency"**
[set] **"amplitude"**
[set] **"phase"**
[connect] **"table"**

Details

construction

FastOsc()
FastOsc(**Table*** table, **float** fr, **float** amp, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

These methods construct a FastOsc object. Its parameters are:

Table* table: pointer to the location of a Table-derived object containing the function table to be scanned.
float fr: fundamental frequency, in Hz. Initialised to 440 by the default constructor.
float amp: amplitude. Initialised to 16000.f .
int vecsize: vector size in samples. Size of the internal DSP buffer, defaults to DEF_VECSIZE, 256.
float sr: sampling rate in HZ. Defaults to DEF_SR, 44100.f.

public methods

void SetFreq(**float** fr)
void SetAmp(**float** amp)
void SetPhase(**float** phase)
void SetTable(**Table*** table)

These methods set the parameters that make up a FastOsc object. Frequency is set in Hz, amplitude is arbitrary (it depends on the scaling/precision used) and the phase is in fractions of a cycle (0-1.0). **SetTable()** connects a table object to this oscillator.

Examples

FastOsc can be used to generate any type of periodic (and one-shot) signals. One typical use is to generate a pitched signal by scanning a table containing a certain wave shape. The table is required to be power-of-two size (2,4,8,…,512, 1024, 2048,…, 2^{28}).

HarmTable sinobj(1024, SINE,1);
FastOsc oscilla(&sinobj, 100.f, 16000.f);

The example above shows an object named *oscilla* which is connected to a table object *sinobj*. The object will generate a sinewave signal with 100Hz frequency and amplitude 16000.f (ca. –6dB in 16-bit):

```
while(processing_on){

oscilla.DoProcess();
output.Write();

}
```

Class FFT

Description

The FFT class implements short-time fourier transform (STFT). An input signal is windowed, transformed by the FFT and scaled. This process happens at regular intervals, determined by the hopsize, which is equivalent to the time-domain vectorsize. The output of this class holds a vector with the size of the FFT containing a real, imaginary pair for every frequency point on the positive side of the spectrum. The real parts for the 0 and SR/2 Hz points are packed together as the first pair in the vector [0,1] (these points are purely real). The limitations for hopsize and fftsize are as follows: (a) the hopsize has to match the time-domain vector size used by the input object (b) the FFT size has to be a power-of-two multiple of the hopsize. As a consequence of this FFT objects (and other spectral processing objects) can be freely combined with time-domain objects in the same processing loop (and within a SndThread object). The FFT size determines the size of the output vector and the hopsize the time interval (in samples) between successive FFT frames.

Construction

FFT()
FFT(**Table*** window, **SndObj*** input, **float** scale=1.f, **int** fftsize=DEF_FFTSIZE, **int** hopsize=DEF_VECSIZE, **float** m_sr=DEF_SR)

Public Methods

parameter/state access:
int GetFFTSize()
int GetHopSize()

parameter/state setting:
void SetWindow(**Table*** window)
void SetScale(**float** scale)
void SetFFTSize(**int** fftsize)
void SetHopSize(**int** hopsize)

Messages

[set] **“scale”**
[set] **“fft size”**
[set] **“hop size”**
[connect] **“window”**

Details

construction

FFT()
FFT(**Table*** window, **SndObj*** input, **float** scale=1.f, **int** fftsize=DEF_FFTSIZE, **int** hopsize=DEF_VECSIZE, **float** m_sr=DEF_SR)

These methods construct a FFT object. Its parameters are:

Table* window: pointer to a Table-derived object containing a window shape to be used in the analysis.
SndObj* input: input signal object (SndObj-derived). Its vector size should match the hopsize set for this FFT object.

float scale: scaling factor. The overall scaling, after transformation is *scale/N*, where *N* is the FFT size.
int fftsize: the FFT size, the number of frequency points in the analysis, which will also determine the output vector size. Defaults to DEF_FFTSIZE (1024).
int hopsize: the hopsize, or decimation, which determines the number of samples in between successive FFT analysis frames. Defaults to DEF_VECSIZE (256).
float sr: the sampling rate for this object. Defaults to DEF_SR (44100.f).

public methods

int GetFFTSize()
int GetHopSize()

These methods retrieve the FFT size and hopsize, which determine the most important aspects of analysis: the number of frequency points and the decimation.

void SetWindow(**Table*** window)
void SetScale(**float** scale)
void SetFFTSize(**int** fftsize)
void SetHopSize(**int** hopsize)

These methods set the parameters that make up a FFT object. They are usually set before performance.

Examples

FFT objects are used to transform a time-domain signal into a frequency-domain signal. The resulting output is a series of spectral frames generated every hopsize/SR seconds (after a call to FFT::DoProcess()). These frames will contain fftsize/2 + 1 frequency points between 0 and SR/2 (inclusive). With the exception of 0 and SR/2 Hz, each frequency point is a complex pair of values containg the real and imaginary values for that frequency. In order to fit all points in a single fftsize vector, the real parts of the 0 and SR/2 Hz points are packed together in the first two positions of the array. This is possible because they are purely real (imaginary parts are 0). The FFT output is normalised, ie. scaled by 1/fftsize, and an optional scaling factor is also applied. An FFT object is usually constructed by passing a window table and an input to it:

FFT analysis(&winobj, &inobj);

Its output can be further transformed by a spectral processing object and the result can be then transformed back into the time-domain (using **IFFT)**. The example below multiplies two spectra:

FFT spec1(&winobj, &inobj1);
FFT spec2(&winobj, &inobj2);
SpecMult mult(&spec1, &spec2);
IFFT tdsig(&winobj, &mult);

This transforms two inputs (from *inobj1* and *inobj2*), multiplies their spectra and transforms the result.

```
while(processing_on) {

inobj1.DoProcess();
inobj2.DoProcess();
spec1.DoProcess();
spec2.DoProcess();
mult.DoProcess();
```

```
tdsig.DoProcess();
output.Write();

}
```

The key aspect of FFT (and all spectral processing classes) is that they output spectral frames, instead of time-domain vectors. Otherwise, their processing behaviour is similar to other time-domain SndObj classes, producing a new output vector when DoProcess() is invoked. Classes can be developed to transform this signal, provided that they take and process a spectral frame packed in the form described above.

Class Filter

Description

Filter is the base class for a number of filtering objects in the library. It provides basic methods to access elements of the filter model. It implements a fixed-frequency and -bandwith standard second-order band-pass filter. This filter is optimised for fixed filtering applications and should be used instead of Reson when there is no need for dynamically-variable parameters

Construction

Filter()
Filter(**float** fr, **float** bw, **SndObj*** InObj, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Public Methods

void SetFreq(**float** fr)
void SetBW(**float** bw)

Messages

[set] **"frequency"**
[set] **"bandwidth"**

Details

construction

Filter()
Filter(**float** fr, **float** bw, **SndObj*** InObj, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

These methods construct an object of the Filter class. Construction parameters are:

float fr: centre frequency, in Hz.
float bw: bandwidth, in Hz.
SndObj* InObj: input object, pointer to the location of a SndObj-derived object.
int vecsize: vector size in samples. Size of the internal DSP buffer, defaults to DEF_VECSIZE, 256.
float sr: sampling rate in HZ. Defaults to DEF_SR, 44100.f.

public methods

void SetFreq(**float** fr)
void SetBW(**float** bw)

These methods set the frequency and bandwidth of a Filter object.

Examples

Filter objects implement basic bandpass filters. The frequency and bandwidth cannot be modulated by a signal, but they can be set at any tiime. A Filter object is usually created by passing values for the centre frequency/bandwidth and an input signal object pointer:

Filter bp(1000.f, 10.f, &inobj,);

Here, a signal produced by *inobj* is filtered by this object with a BW of 10 Hz and a frequency offset of 1000.f.

```
while(processing_on){

inobj.DoProcess();
bp.DoProcess();
output.Write();

}
```

Class FIR

Description

FIR is a DelayLine-derived object that implements direct convolution of an impulse response signal with a signal input from a SndObj object. The impulse response (FIR coefficients) can be given as an array of samples or as a Table object. This is a time-domain process (unlike Convol) and it can be slow for long impulse responses. At every sample, delayed and scaled input samples are summed to produce the output.

Construction

FIR()
FIR(**Table*** coeftable, **SndObj*** input, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)
FIR(**float*** impulse, **int** impulsesize, **SndObj*** input, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Public Methods

void SetTable(**Table*** coeftable)
void SetImpulse(**float*** impulse, **int** impulsesize)

Messages

[set] **“impulse size”**
[connect] **“impulse”**
[connect] **“table”**

Details

construction

FIR()

The default constructor, parameters set to default values.

FIR(**Table*** coeftable, **SndObj*** input, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

This constructor creates a FIR object which will take its impulse response from a Table-derived object connect to it.

**Table ***coeftable: a Table-derived object containing the impulse response (the FIR coefficients).

FIR(**float*** impulse, **int** impulsesize, **SndObj*** input, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

This constructor creates a FIR object that takes an array containing the impulse response and its size and fills an internal table to use as its impulse response.

float* impulse: an impulse response vector (the FIR coefficients).
int impulsesize: the size of the impulse response.

Both constructors will take the other parameters:

SndObj* input: an input SndObj which will generate the signal to be modified.
int vecsize: vector size in samples. Size of the internal DSP buffer, defaults to DEF_VECSIZE, 256.
float sr: sampling rate in HZ. Defaults to DEF_SR, 44100.f.

public methods

void SetTable(**Table*** coeftable)

This method connects the object to a table object containing the impulse response. It has the same effect as **Connect(**"table", coeftable**)**.

void SetImpulse(**float*** impulse, **int** impulsesize)

This method fills an internal table with the impulse response *impulse*, of size *impulsesize*. It has the same effect as calling **Set(**"impulse size", impulsesize**)** followed by **Connect(**"impulse", impulse**)**.

Examples

FIR objects can be constructed with external or internal tables. In terms of efficiency and elegance, it is advisable to use external tables, which can be connected to multiple FIR objects, if necessary.

FIR lowpass(&coefobj, &inobj);

This object takes its coefficients from a *coefobj* object connected to it and processes the input signal *inobj.*

```
while(processin_on){

inobj.DoProcess();
lowpass.DoProcess();
output.Write();

}
```

Class Gain

Description

The Gain class is a simple tool for controlling the amplitude signal of an object. It boosts/cuts the input signal by a specified amount (in dB or by a multiplier).

Construction

Gain()
Gain(**float** gain, **SndObj*** InObj = 0, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Public Methods

void SetGain(**float** gain)
void SetGainM(**float** val)

Messages

"gain"
"gain multiplier"

Details

construction

Gain()
Gain(**float** gain, **SndObj*** InObj = 0, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

These methods construct an object of the Gain class. Construction parameters are:

float gain: amount of cut/boost in dB (the signal amplitude is doubled/halved every 6dB of change).
SndObj* InObj: input object, pointer to the location of a SndObj-derived object.
int vecsize: vector size in samples. Size of the internal DSP buffer, defaults to DEF_VECSIZE, 256.
float sr: sampling rate in HZ. Defaults to DEF_SR, 44100.f.

public methods

void **SetGain**(float gain)
void **SetGainM**(float val)

Sets the gain. **SetGain()** expects the gain in dB, whereas **SetGainM()** takes a gain multiplier

Examples

The Gain object adjusts the amplitude of a signal, according to its gain setting. The gain setting is usually defined in the dB scale, according to:

$$gain_{dB} = 20\log\frac{output\_amp}{input\_amp}$$

In this case, a gain setting of –6dB will attenuate the input peak amplitude of a signal by around 0.5. A Gain object constructed to perform this operation will look like this:

Gain atten(-6.f, &inObj);

Similarly, if we use the **SetGainM()** method, we can set the gain to a specific multiplier (in this case, 0.5):

Gain atten(0.f, &inObj);
atten.SetGainM(0.5f);

A call to **Gain::DoProcess()** will, as with all SndObjs, process the input sound and output the attenuated signal.

Class HammingTable

Description

The HammingTable class implements a generalised Hamming window, according to *w(n) = α + (1 - α)cos(2π n/N)* over *-(N-1)/2 <= n <= (N-1)/2*.

Construction

HammingTable()
HammingTable(**long** L, **float** alpha)

Public Methods

void SetParam(**long** L, **float** alpha=.54)

Details

construction

HammingTable()
HammingTable(**long** L, **float** alpha)

Constructs a HammingTable object.

long L: table length.
float alpha: the value of the constant α. Alters the shape of the window, 0.54 for *Hamming*, 0.5 for*Hanning* window types. Defaults to 0.54.

public methods

void SetParam(**long** L, **float** alpha=.54)

Sets the HammingTable parameters. MakeTable() needs to invoked after any parameter change for table re-building.

Class HarmTable

Description

Harmonic function table. Generates four preset types of waveforms: sine, saw, square and buzz (pulse), with any number of harmonics.

Construction

HarmTable()
HarmTable(**long** L, **int** harm, **int** type, **float** phase=0.f)

Public Methods

void SetHarm(**int** harm, **int** type)
void SetPhase(**float** phase)

Details

construction

HarmTable()
HarmTable(**long** L, **int** harm, **int** type, **float** phase=0.f)

Constructs a HarmTable object.

long L: table length.
int harm: number of harmonics. Defaults to 1
int type: preset waveshape. Any of the following: SINE, SAW, SQUARE or BUZZ.

public methods

void SetHarm(**int** harm, **int** type)
void SetPhase(**float** phase)

These methods set the function table parameters. MakeTable() should be invoked after any parameter resetting.

Class Name

Description

Hilb implements a Hilbert Filter Transformer, separating a real signal into real and imaginary parts. These can be combined to generate a signal without negative frequencies. Useful for implementing SSB modulators, phase shifters, etc. The real and imaginary outputs are available as SndObj object pointers.

Construction

Hilb()
Hilb(**SndObj*** input, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Public Member Variables

SndObj* real
SndObj* imag

Details

construction

Hilb()
Hilb(**SndObj*** input, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

These methods construct an object of the Hilb class. Construction parameters are:

SndObj* input: pointer to a SndObj object whose signal will be processed by the Hilb object.
int vecsize: vector size in samples. Size of the internal DSP buffer, defaults to DEF_VECSIZE, 256.
float sr: sampling rate in HZ. Defaults to DEF_SR, 44100.f.

public member variables

SndObj* real
SndObj* imag

These member variables are pointers to the output objects containing the real and imaginary signals. These pointers to a SndObj class should be used when patching the real or imaginary output of the **Hilb** class to another object(s) of the library. The **Hilb** object itself outputs a signal that is the sum of its real and imaginary outputs.

Class HiPass

Description

HiPass models a first-order high-pass IIR filter. It has a gentle slope of –3dB per octave and it can be used to enhance high frequencies in a sound.

Construction

HiPass()
HiPass(**float** freq, **SndObj *** inObj, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Details

construction

HiPass()
HiPass(**float** freq, **SndObj *** inObj, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

The construction parameters are:

float freq: cutoff frequency in Hz of the high-pass filter.
SndObj* inObj: input signal generator, a pointer to SndObj (or derived) object.
int vecsize: the output vector size in samples, defaults to DEF_VECSIZE.
float SR: the sampling rate in Hz, defaults to DEF_SR.

Examples

The HiPass object can be used as a low-frequency attenuator and a high-frequency emphasiser. Given its gentle slope, its results are not dramatic, but it can be used as an overall tone control.

HiPass treble(3000.f, &inObj);

The example above shows a HiPass filter with a cutoff frequency of 3000 Hz. A call to HiPass::DoProcess() will filter the frequencies below that one with a slope of –3dB per octave.

```
while(processing_on){

inObj.DoProcess();
treble.DoProcess();
output.Write();

}
```

Class IADSR

Description

This object generates an **attack - decay - sustain - release** shaped signal at the output, with initial and end amplitude values. Alternatively, it can similarly shape an input signal, acting as a modifier. The difference between this class and its parent ADSR is that it expects values for the initial and end amplitudes, which in the latter case are always 0. This object can be used to generate envelopes that do not necessarily start and end in 0, such as, a frequency envelope.

Construction

IADSR()
IADSR(**float** init, **float** att, **float** maxamp, **float** dec, **float** sus, **float** rel, **float** end, **float** dur, **SndObj*** InObj = 0, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Public Methods

void SetInit(**float** init)
void SetEnd(**float** end)

Messages

[set] **“init”**
[set] **“end”**

Details

construction

IADSR()
IADSR(**float** init, **float** att, **float** maxamp, **float** dec, **float** sus, **float** rel, **float** end, **float** dur, **SndObj*** InObj = 0, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

These methods construct an object of the IADSR class. Construction parameters are:
float init: initial amplitude.
float att: attack time, or rise time, in secs. Time taken for the signal to change from **init** to **maxamp**.
float maxamp: maximum amplitude after rise time. It is a multiplier, in case of the shaping of an input signal.
float dec: decay time, in secs. Time taken for the signal to change from **maxamp** to **sus**.&
float sus: sustain amplitude after decay time. Again, a multiplier, in case of envelope shaping. The sustain period is calculated on the basis of the difference between the total duration and the sum of the attack, decay and release times.
float rel: release time, in secs, after the sustain period, during which the signal changes from **sus** to **end**. It is calculated backwards from the end, taken from the total duration of the envelope.
float end: end amplitude.
float dur: total duration of the envelope, in secs, or the envelope period. This ADSR is designed to loop, so the whole shape will be repeated after the total duration is completed.
SndObj* InObj: input object, pointer to the location of a SndObj-derived object. Defaults to 0, which means *no input object*, so the ADSR object is used as a signal generator.
int vecsize: vector size in samples. Size of the internal DSP buffer, defaults to DEF_VECSIZE, 256.
float sr: sampling rate in HZ. Defaults to DEF_SR, 44100.f

public methods

void SetInit(**float** init)
void SetEnd(**float** end)

These methods set the parameters that IADSR adds to the ones inherited from its base class, ADSR, the initial and the end values.

Examples

IADSR is usually constructed by setting the envelope parameters in the constructor. The example below creates an IADSR object which will shape the output of a previously declared object named modulator for a period of 2.5 secs.

IADSR modndx(3.f, .01f, 5.f, .2f, 4.f, .05f, 1.5f, 2.5f, &modulator);

If an object input is not given, IADSR works as signal generator.

Class IFAdd

Description

The IFAdd class implements additive resynthesis, based on a cubic interpolation algorithm. The class takes an input from an IFGram object, which consists of amplitude, frequency and phase data for each DFT bin. IFAdd objects can resynthesise any number of bins up to fftsize/2. IFAdd can modify the timescale of the resynthesis by altering the hopsize between frames and modify the pitch by scaling the frequencies on each hopsize

Construction

IFAdd()
IFAdd(**IFGram*** input, **int** bins, **Table*** table, **float** pitch=1.f, **float** scale=1.f,
float tscale=1.f, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Details

construction

IFAdd()
IFAdd(**IFGram*** input, **int** bins, **Table*** table, **float** pitch=1.f, **float** scale=1.f,
float tscale=1.f, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

These methods construct a IFAdd object:

IFGram* input: input object.
int mbins: maximun number of bins to be resynthesised (up to fftsize/2)
Table* table: table object containing a wavetable to be used by each oscillator in the resynthesis, typically a cosine wave.
float pitch: pitch scaling of output (transposition ratio).
float scale: amplitude scaling of output.
float tscale: timescaling factor, the interpolation(synthesis hopsize):decimation ratio(analysis hopsize)
int vecsize: object vector size, also determines the synthesis hopsize between analysis frames (defaults to 256).
float sr: sampling rate in Hz (defaults to 44100).

Examples

The following connections are a simple example of the use of IFAdd to resynthesise an IFGram analysis signal.

```
HarmTable table(4000, 1, 1, 0.75);    // cosine wave
HammingTable window(fftsize, 0.5); // hanning window

// input sound
SndWave input(infile,READ,1,16,0,0.f,decimation);
SndIn   insound(&input, 1, decimation);

// IFD analysis
IFGram ifgram(&window,&insound,1.f,fftsize,decimation);
// IFAdd resynthesis
IFAdd  synth(&sinus,bins,&table,pitch,scale,(float) interpolation/decimation,
                    interpolation);

// output sound
```

```
SndWave output(outfile, OVERWRITE,1,16,0,0.f,interpolation);
output.SetOutput(1, &synth);
```

This code takes an input sound, from a file and passes it through the analysis process and then the data is resynthesised. The timescale change is determined by the decimation:interpolation ratio. In order to implement processing, the programmer either needs to write a loop and call the reading/writing and processing methods, or use a SndThread object, passing these objects to it.

Class IFFT

Description

The class IFFT implements the inverse short-time fourier transform (ISTFT). It takes a spectral-domain signal frame, transforms it into a time-domain signal, applies a window and overlap-adds the successive transformed frames. It expects an input composed of *fftsize* samples, in real fft format, wher e *fftsize* is the number of frequency points in the transform. The real parts of the 0Hz and SR/2 frequency points should be packed in positions 0 and 1, respectively, of the array and should be followed by the complex pairs for the positive side of the spectrum for all the other frequency points. An IFFT object restores the original signal analysed using an FFT object. The hopsize is equivalent to the output vector size and should always be set to power-of-two divisors of the fftsize.

Construction

IFFT()
IFFT(**Table*** window, **SndObj*** input, **int** fftsize=DEF_FFTSIZE, **int** hopsize=DEF_VECSIZE, **float** sr=DEF_SR)

Public Methods

parameter/state access:
int GetFFTSize()
int GetHopSize()

parameter/state setting:
void SetWindow(**Table*** window)
void SetFFTSize(**int** fftsize)
void SetHopSize(**int** hopsize)

Messages

[set] **“scale”**
[set] **“fft size”**
[set] **“hop size”**
[connect] **“window”**

Details

construction

IFFT()
IFFT(**Table*** window, **SndObj*** input, **int** fftsize=DEF_FFTSIZE, **int** hopsize=DEF_VECSIZE, **float** sr=DEF_SR)

An IFFT object is constructed with the following parameters:

Table* window: pointer to a Table object which describes a window shape to be used in the resynthesis.
SndObj* input: input spectral object, a pointer to a SndObj (or derived) which produces the spectral signal to be transformed. Its output vector size should match the fftsize defined for this object.
int fftsize: the FFT size, the number of frequency points in the transform, which will also determine the output vector size. Defaults to DEF_FFTSIZE (1024).
int hopsize: the hopsize, or decimation, which determines the number of samples in between successive FFT input frames. Defaults to DEF_VECSIZE (256).
float sr: the sampling rate for this object. Defaults to DEF_SR (44100.f).

public methods

int GetFFTSize()
int GetHopSize()

These methods retrieve the FFT size and hopsize, which determine the most important aspects of the inverse transform: the number of frequency points and the decimation.

void SetWindow(**Table*** window)
void SetFFTSize(**int** fftsize)
void SetHopSize(**int** hopsize)

These methods set the parameters that make up an IFFT object. They are usually set before performance.

Examples

IFFT objects are used to transform a frequency-domain signal into a time-domain signal. The resulting output is the resynthesis of a series of input spectral frames effected every hopsize/SR seconds (after a call to IFFT::DoProcess()). The output frames are *fftsize* samples long, so the output will be the result of the overlap of *fftsize/hopsize* signal frames. The IFFT object is created by passing it a window object and an input object, as in:

IFFT tdsig(&winobj, &mult);

The following example show an application of IFFT to resynthesise a transformed signal:

```
FFT fdsig(&winobj, &input);
SpecThresh nr(0.1f, &fdsig);
IFFT tdsig(&winobj, &nr);

while(processing_on){

input.DoProcess();
fdsig.DoProcess();
nr.DoProcess();
tdsig.DoProcess();
output.Write();

}
```

Class IFGram

Description

The IFGram class implements Instantaneous Frequency Distribution analysis of a time-domain signal, plus its magnitude spectrum. Its output is presented in a similar format to the PVA (phase vocoder) class: with the exception of the first two values, the frequencies (in Hz) and magnitudes for each analysis channel (or bin). The first value refers to DC magnitude (0 Hz) and the second to the magnitude at the Nyquist frequency. The analysis process is similar to that of the STFT (FFT class), in that a signal is windowed and scaled. The process happens at regular intervals, determined by the hopsize, which is equivalent to the time-domain vectorsize. In addition to the frequencies and amplitudes, it also outputs the current phases in unwrapped format.

Construction

IFGram()
IFGram(**Table*** window, **SndObj*** input, **float** scale=1, **int** fftsize=DEF_FFTSIZE,
int hopsize=DEF_VECSIZE, **float** sr=DEF_SR)

Details

construction

IFGram()
IFGram(**Table*** window, **SndObj*** input, **float** scale=1, **int** fftsize=DEF_FFTSIZE, **int** hopsize=DEF_VECSIZE, **float** sr=DEF_SR)

These methods construct an IFGram object. Its parameters are:

Table* window: pointer to a Table-derived object containing a window shape to be used in the analysis.
SndObj* input: input signal object (SndObj-derived). Its vector size should match the hopsize set for this IFGram object.
float scale: scaling factor. The overall scaling, after transformation is *scale/N*, where *N* is the FFT size.
int fftsize: the FFT size, the number of frequency points in the analysis, which will also determine the output vector size. Defaults to DEF_FFTSIZE (1024).
int hopsize: the hopsize, or decimation, which determines the number of samples in between successive FFT analysis frames. Defaults to DEF_VECSIZE (256).
float sr: the sampling rate for this object. Defaults to DEF_SR (44100.f).

Examples

IFGram objects are used to transform a time-domain signal into its frequency-domain representation, containing frequency and amplitude values at a particular time-slice for each analysis channel. Their output is a sequence of frames, one for each time-slice. The interval between each time-slice is determined by the hopsize (in samples). The analysis output is very similar to the one of PVA objects, except that here phases are unwrapped (ie. not limited to the range of principal values $\pm\pi$). An IFGram object is constructed by at least passing pointers to window and input objects to it:

IFGram analysis(&winobj, &inobj);

Its output can be used by any object that takes PVA-format spectral signals, such as PVMorph:

IFGram analysis1(&winobj, &inobj1);
IFGram analysis2(&winobj, &inobj2);
PVMorph morph(0.5, 0.5, &analysis1, &analysis2);

IFGram can also provide the analysis input to the sinusoidal modelling classes:

IFGram analysis(&window,&insound,1.f,fftsize,decimation);
SinAnal sinus(&analysis,thresh,intracks);
AdSyn synth(&sinus,outracks,&table, pitch,scale,interpolation)

In this example, the variables *decimation* and *interpolation* control the timestrech ratio, whereas *pitch* and *scale* control transposition and amplitude scaling, respectively

```
while(processing _on){
input.Read();
insound.DoProcess();
analysis.DoProcess();
sinus.DoProcess();
synth.DoProcess();
output.Write();
}
```

See the FFT and PVA classes for more information on spectral analysis classes.

Class ImpulseTable

Description

The ImpulseTable object builds an impulse response function table based on a spectral magnitude envelope and a linear phase response. The envelope is defined by a starting point and two arrays: (1) segment legnths and (2) end points of each segment. The segments can be either exponential or linear. An optional window can be applied to the impulse response, supplied as a table object with the same length. The resulting table will contain the coefficients for a linear-phase FIR filter with a good approximation of the desired frequency response.

Construction

ImpulseTable()
ImpulseTable(**long** L, **int** segments, **float** start,
float* points, **float*** lengths,**float** type = 0.f,
table window=0, **float** nyquistamp=0.f)

Public Methods

void SetWindow(**Table*** window)

Details

construction

ImpulseTable()
ImpulseTable(**long** L, **int** segments, **float** start,
float* points, **float*** lengths,**float** type = 0.f, **Table*** window, **float** nyquistamp=0.f)

Constructs a ImpulseTable object.

long L: table length.
int segments: number of envelope segments.
float start: starting value of envelope (0 Hz magnitude).
float* points: an array of floats, containing the end values of each segment. Must match the above number of segments.
float* lengths: an array of floats, containing the lengths of each segment. Must match the above number of segments. Segment lengths are normalised to the table size (added up and then each one is divided by that total and multiplied by the table size).
float type: type of curve. Linear = 0, inverse exponential < 0 < exponential.
Table* window: table object of the same length, containing a time-domain window, which will be used to smooth the impulse response.
float nyquistamp: Nyquist magnitude.

public methods

void SetWindow(**Table*** window)

This method sets the table object used to smooth the impulse response.

Class Interp

Description

This class interpolates between two points during a specified space of time. The interpolation can be linear, exponential or inverse exponential. Once the final point is reached, the output will remain at that value if not reset.

Construction

Interp()
Interp(**float** initial, **float** final, **float** dur, **float** type = 0.f, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Public Methods

Restart()
void SetDur(**float** dur)
void SetCurve(**float** initial, **float** final, **float** type = 0.f)

Messages

[set] **“initial”**
[set] **“final”**
[set] **“duration”**
[set] **“type”**

Details

construction

Interp()
Interp(**float** initial, **float** final, **float** dur, **float** type = 0.f, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

These methods construct an object of the Interp class. Construction parameters are:

float initial: initial value.
float final: final value.
float dur: total duration of the interpolation process, in seconds.
float type: type of interpolation, linear = 0, exponential < 0 < inverse exponential.
int vecsize: vector size in samples. Size of the internal DSP buffer, defaults to DEF_VECSIZE, 256.
float sr: sampling rate in HZ. Defaults to DEF_SR, 44100.f.

public methods

Restart()

This method resets the object and the output starts again from the first point towards the final value.

void SetDur(**float** dur)
void SetCurve(**float** initial, **float** final, **float** type = 0.f)

These methods set the different parameters that make up an Interp object: total duration of interpolation (in seconds), initial and final values.

Examples

Interp objects are used as control functions for all different purposes. For instance an exponential interpolation between two frequency values can be created as follows:

Interp exp(1.f, 100.f, 150.f, 2.f)l

This can be used to control the frequency of an oscillator:

Oscili osc(&table, 0.f, 1000.f, &exp);

This will generate an exponential glissando between 100 and 150 Hz.

```
while(processing_on){

exp.DoProcess();
osc.DoProcess();
output.Write();

}
```

Class Lookup

Description

This class performs truncating table lookup controlled by an input SndObj-derived object. The input signal is used as the index for the lookup procedure. Other parameters are index offset and input Table-derived object. Lookup can be either by raw table entry (normalisation mode=RAW_VALUE) or normalised (NORMALISED), which expects signal varying between 0 and 1. If these values lie outside the lookup range, lookup will be either wrapped around (modulus operation) or limited (kept at the max or min values of the table).

Construction

Lookup()
Lookup(**Table*** table, **long** offset, **SndObj*** InObj, **int** mode = WRAP_AROUND, **int** normal=RAW_VALUE, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Public Methods

void Offset(**long** offset)
void SetMode(**int** mode, **int** normal)
void SetTable(**Table*** table)

Messages

[set] **“read mode”**
[set] **“index type”**
[set] **“offset”**
[connect] **“table”**

Details

construction

Lookup()
Lookup(**Table*** table, **long** offset, **SndObj*** InObj, **int** mode = WRAP_AROUND, **int** normal=RAW_VALUE, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

These methods construct an object of the Lookup class. Construction parameters are:

Table* table: pointer to the location of a Table-derived object.
long offset: index offset.
SndObj* InObj: input object, pointer to the location of a SndObj-derived object. The output signal from this object is added to the offset value and then used as the index on the lookup process.
int mode: lookup mode, either WRAP_AROUND or LIMIT. The first option reads the table wrapping around, when index falls outside the table size. The second, limits the reading to the range of 0 < table_length.
int normal: normalisation mode, either RAW_VALUE, where the input signal is taken literally as raw table indices, or NORMALISED, where the input signal is expected to lie between 0 and 1.
int vecsize: vector size in samples. Size of the internal DSP buffer, defaults to DEF_VECSIZE, 256.
float sr: sampling rate in HZ. Defaults to DEF_SR, 44100.f.

public methods

void Offset(**long** offset)

This method determines the offset amount for the index. It can also be invoked with the message **“offset”** passed to **Set()**.

void SetMode(**int** mode, **int** normal)

This method sets the read mode (WRAPAROUND or LIMIT) and the normalisation (index type, RAW_VALUE or NORMALISED). Messages **“read mode”** and **“index type”** can be sent to the object to perform the same operation as this method.

void SetTable(**Table*** table)

This method connects a table object, which will be used in the lookup operation. The message **“table”** can be sent (using **Connect()**) to the object to perform the same operation.

Examples

Table lookup is one of the most basic operations in Computer Music. It is used to retrieve values from tables, for control or signal generation purposes. For instance, a simple truncating oscillator can be implemented by combining a Phase object (which will provide an incrementing index) with a table object:

```
Phase   phi(440.f)
Lookup  lup(&table, 0.f, &phi, WRAP_AROUND, NORMALISED);
```

This will generate a signal with a fundamental at 440 Hz.

```
while(processing_on){

phi.DoProcess();
lup.DoProcess();
output.Write();

}
```

Class Lookupi

Description

This object performs interpolating table Lookupi controlled by an input SndObj-derived object. It behaves in the same way as its parent class, Lookup.

Construction

Lookupi()
Lookupi(**Table*** table, **long** offset, **SndObj*** InObj, **int** mode = WRAP_AROUND, **int** normal=RAW_VALUE, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Details

construction

Lookupi()
Lookupi(**Table*** table, **long** offset, **SndObj*** InObj, **int** mode = WRAP_AROUND, **int** normal=RAW_VALUE, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

These methods construct an object of the Lookupi class. Construction parameters are:

Table* table: pointer to the location of a Table-derived object.
long offset: index offset.
SndObj* InObj: input object, pointer to the location of a SndObj-derived object. The output signal from this object is added to the offset value and then used as the index on the Lookupi process.
int mode: Lookupi mode, either WRAP_AROUND or LIMIT. The first option reads the table wrapping around, when index falls outside the table size. The second, limits the reading to the range of 0 < table_length.
int normal: normalisation mode, either RAW_VALUE, where the input signal is taken literally as raw table indices, or NORMALISED, where the input signal is expected to lie between 0 and 1.
int vecsize: vector size in samples. Size of the internal DSP buffer, defaults to DEF_VECSIZE, 256.
float sr: sampling rate in HZ. Defaults to DEF_SR, 44100.f.

Examples

Lookupi is simply an interpolating version of Lookup. For non-integral table positions, it outputs a linearly interpolated value between the two points. Phase modulation synthesis can be implemented by modulating a phase value with an oscillator signal:

Oscili mod(&sinetable, fm, index);
Phase phi(fc, mod)
Lookupi car(&sinetable, 0.f, &phi, WRAP_AROUND, NORMALISED);

This will generate a PM signal with modulating frequency *fm*, carrier frequency *fc* and index of modulation *index*.

```
while(processing_on){
mod.DoProcess();
phi.DoProcess();
car.DoProcess();
output.Write();
}
```

Class LowPass

Description

LoPass models a first-order low-pass IIR filter. It has a gentle slope of –3dB per octave and it can be used to enhance low frequencies in a sound.

Construction

LoPass()
LoPass(**float** freq, **SndObj ***inObj, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Details

construction

LoPass()
LoPass(**float** freq, **SndObj ***inObj, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

The construction parameters are:

float freq: cutoff frequency in Hz of the low-pass filter.
SndObj* inObj: input signal generator, a pointer to SndObj (or derived) object.
int vecsize: the output vector size in samples, defaults to DEF_VECSIZE.
float SR: the sampling rate in Hz, defaults to DEF_SR.

Examples

The LoPass object can be used as a high-frequency attenuator and a low-frequency emphasiser. Given its gentle slope, its results are not dramatic, but it can be used as an overall tone control.

LoPass bass(300.f, &inObj);

The example above shows a LoPass filter with a cutoff frequency of 300 Hz. A call to LoPass::DoProcess() will filter the frequencies above that one with a slope of –3dB per octave.

```
while(processing_on){

inObj.DoProcess();
bass.DoProcess();
output.Write();

}
```

Class LoPassTable

Description

This class implements a Table holding an impulse response for a low-pass FIR filter design.

Construction

LoPassTable(**int** impulsesize, **float** fr, **float** sr=44100)
LoPassTable()

Public Methods

void SetFreq(**float** freq)
void SetSr(**float** sr)

Details

construction

LoPassTable(**int** impulsesize, **float** fr, **float** sr=44100)
LoPassTable()

Constructs a LoPassTable object.

int impulsesize: size of the impulse response, which will determine the table size.
float fr: cut-off freq of the low-pass filter described by the impulse response.
float sr: sampling rate, in Hz.

public methods

void SetFreq(**float** freq)
void SetSr(**float** sr)

These methods set the function table parameters. MakeTable() should be invoked after any parameter resetting.

Class LP

Description

This class implements an analogue-style resonant low-pass filter. The filter frequency and resonance bandwidth can be either fixed or time-varying.

Construction

Lp()
Lp(**float** fr, **float** bw, **SndObj*** inObj, **SndObj*** inputfreq = 0, **SndObj*** inputbw = 0, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Details

construction

Lp()
Lp(**float** fr, **float** bw, **SndObj*** inObj, **SndObj*** inputfreq = 0, **SndObj*** inputbw = 0, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

These methods construct an object of the Lp class. Construction parameters are:

float fr: centre frequency offset, in Hz.
float bw: bandwidth offset, in Hz.
SndObj* inObj: pointer to an input SndObj-derived object.
SndObj* inputfreq: frequency control input, pointer to the location of a SndObj-derived object. The centre frequency can be controlled by a time-varying signal from another SndObj-derived object. A signal is fed from the input object and added to the frequency offset value. Defaults to 0, *no frequency input object*
SndObj* inputbw: bandwidth control input, pointer to the location of a SndObj-derived object. The bandwidth can be controlled by a time-varying signal from another SndObj-derived object. A signal is fed from the input object and added to the bandwidth offset value. Defaults to 0, *no bandwidth input object*
int vecsize: vector size in samples. Size of the internal DSP buffer, defaults to DEF_VECSIZE, 256.
float sr: sampling rate in HZ. Defaults to DEF_SR, 44100.f.

Examples

The LP object can be used as a resonating lowpass filter, which will provide a region of emphasis around the cutoff frequency. The filter will self-oscillate when the bandwidth of the resonance region is very small.

LP filter(0.f, 5.f, &inObj, &inFr);

The example above shows a LP filter with its cutoff frequency controlled by an input signal from inFr, a SndObj-derived object. Its bandwidth is set to 5 Hz.

```
while(processing_on){

inObj.DoProcess();
inFr.DiProcess();
filter.DoProcess();
output.Write();

}
```

Class MidiMap

Description

This class parses a MIDI input in a similar way to its parent class MidiIn and maps its output into a user-defined range. The mapping can be done in two ways: using an input map table or into a specified range. The latter option linearly maps the raw MIDI value in a range between the maximum and minimum values. The mapping table option requires a Table-derived object with 128 positions defining the output values for each raw MIDI number.

Construction

MidiMap()
MidiMap(**SndMidiIn*** input, **Table*** maptable, **short** message = NOTE_MESSAGE, **short** channel = 1, **int** vecsize = DEF_VECSIZE, **float** sr = DEF_SR)
MidiMap(**SndMidiIn*** input, **float** min, **float** max, **short** message = NOTE_MESSAGE, **short** channel = 1, **int** vecsize = DEF_VECSIZE, **float** sr = DEF_SR)

Public Methods

void SetTable(**Table*** maptable)
void SetRange(**float** min, **float** max)

Messages

[set] **"range min"**
[set] **"range max"**
[connect] **"map table"**

Details

construction

MidiMap()
MidiMap(**SndMidiIn*** input, **Table*** maptable, **short** message = NOTE_MESSAGE, **short** channel = 1, **int** vecsize = DEF_VECSIZE, **float** sr = DEF_SR)
MidiMap(**SndMidiIn*** input, **float** min, **float** max, **short** message = NOTE_MESSAGE, **short** channel = 1, **int** vecsize = DEF_VECSIZE, **float** sr = DEF_SR)

These methods construct an object of the MidiIn class. The second method is used when mapping employing a user-defined table and the third is used for linear mapping. Construction parameters are:

SndMidiIn* input: pointer to the location of a SndMidiIn object.
Table* maptable: pointer to a Table-derived object (size 128) containing the mapping values.
float min: minimum value for linear mapping.
float max maximum value for linear mapping.
short message: type of MIDI channel message which will be output by the object. Valid values are:

Any control change device number (values 0 - 127)
NOTE_MESSAGE, note on
PBEND_MESSAGE, pitchbend or controller 0
MOD_MESSAGE, modulation wheel (controller 1)
BREATH_MESSAGE, breath control (controller 2)
FOOT_MESSAGE, breath control (controller 4)
PORT_MESSAGE, portamento (controller 5)
VOL_MESSAGE, volume (controller 6)

BAL_MESSAGE, balance (controller 7)
PAN_MESSAGE , pan (controller 9)
EXPR_MESSAGE, expression (controller 10)
AFTOUCH_MESSAGE, monophonic aftertouch or channel pressure
POLYAFTOUCH_MESSAGE, polyphonic aftertouch
PROGRAM_MESSAGE, program change
VELOCITY_MESSAGE, velocity (from a note message)
NOTEOFF_MESSAGE, note off
Defaults to NOTE_MESSAGE

short channel: MIDI channel from which a message will be read. Defaults to channel 1.
int vecsize: vector size in samples. Size of the internal DSP buffer, defaults to DEF_VECSIZE, 256.
float sr: sampling rate in HZ. Defaults to DEF_SR, 44100.f.

public methods

This class provides methods for setting the range of the linear mapping, **SetRange()** and the **"range min"/"range max"** set messages, as well as for connecting a user-defined table for any type of mapping, **SetTable()** and the **"map table"** message. Whenever a mapping table exists and is connected to a MidiMap object, the output mapping will obey this table. Otherwise, mapping will be linear between the set min and max values .

Examples

MidiMap works in a similar way to its parent class, MidiIn. The only difference is the mapping of the raw MIDI values into an specific range, or according to a mapping table. The example below shows the mapping of a MIDI volume input on channel 2 into a linear range to control the gain of signal. For a mapping table example, please refer to the documentation on the **NoteTable** class.

MidiMap vol(&MidiObj, 0.f, 1.f, VOL_MESSAGE, 2);
Ring gain(&inObj, &vol);

The general-purpose ring modulator, which is a simple multiplier is used as gain control, taking its inputs from **InObj** and **vol**.

```
while(processing_on){

midiObj.Read();
vol.DoProcess();
InObj.DoProcess();
gain.DoProcess();
output.Write();

}
```

Class MidiIn

Description

This class reads one MIDI channel and parses a selected MIDI message from a **SndMidiIn** object.

Construction

MidiIn()
MidiIn(**SndObj*** InObj = 0, **short** message=NOTE_MESSAGE, **short** channel=1, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Public Methods

short SetInput(**SndMidiIn*** input)
void SetChannel(**short** channel)
void SetMessage(**short** message)

Messages

[set] **"message type"**
[set] **"channel"**
[connect] **"midi input"**

Details

construction

MidiIn()
MidiIn(**SndObj*** InObj = 0, **short** message=NOTE_MESSAGE, **short** channel=1, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

These methods construct an object of the MidiIn class. Construction parameters are:

SndMidiIn* input: pointer to the location of a SndMidiIn object.
short message: type of MIDI channel message which will be output by the object. Valid values are:

Any control change device number (values 0 - 127)
NOTE_MESSAGE, note on
PBEND_MESSAGE, pitchbend or controller 0
MOD_MESSAGE, modulation wheel (controller 1)
BREATH_MESSAGE, breath control (controller 2)
FOOT_MESSAGE, breath control (controller 4)
PORT_MESSAGE, portamento (controller 5)
VOL_MESSAGE, volume (controller 6)
BAL_MESSAGE, balance (controller 7)
PAN_MESSAGE , pan (controller 9)
EXPR_MESSAGE, expression (controller 10)
AFTOUCH_MESSAGE, monophonic aftertouch or channel pressure
POLYAFTOUCH_MESSAGE, polyphonic aftertouch
PROGRAM_MESSAGE, program change
VELOCITY_MESSAGE, velocity (from a note message)
NOTEOFF_MESSAGE, note off
Defaults to NOTE_MESSAGE

short channel: MIDI channel from which a message will be read. Defaults to channel 1.

int vecsize: vector size in samples. Size of the internal DSP buffer, defaults to DEF_VECSIZE, 256.
float sr: sampling rate in HZ. Defaults to DEF_SR, 44100.f.

public methods

In order to use MidiIn, a SndMidiIn object needs to be connected to it. This can be done by sending a connect **"midi input"** message to the object, together with a pointer to the SndMidiIn object. Alternatively **SetInput()** can be used. MidiIn also needs to know what message it is supposed to capture from the MIDI stream, you can use **SetMessage()** or send a set **"message type"** message to it. The constants associated with the different possible channel messages are shown above. Finally, **SetChannel** (and associated message) will define to which MIDI channel this object is listening.

Examples

MidiIn is the simplest class that implements MIDI input parsing in the library. It listens for specific messages on specific channels, from a SndMidiIn input. Once such a message is received it keeps outputing its value (in the range 0-127) until a new message is received. For instance, in the example below, a modulation wheel message on channel 5 is being parsed:

MidiIn mod(&midiObj, MOD_MESSAGE, 5);

Note that this class only outputs raw MIDI values, which in most cases needs to be scaled or mapped into a more useful range. Here the raw modulation wheel position is just being used to control a filter bandwidth:

LP filter(200.f, 1.f, &InObj, 0, &mod);

```
while(processing_on){

midiObj.Read();
mod.DoProcess();
InObj.DoProcess();
filter.DoProcess();
output.Write();

}
```

Class Mixer

Description

This object mixes together the outputs of any number of SndObj-derived objects and outputs the mixed signal. The input objects are added to a list which is used to perform the signal mixing.

Construction

Mixer()
Mixer(**int** ObjNo, **SndObj**** InObjs, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Public Methods

short AddObj(**SndObj*** InObj)
short DeleteObj(**SndObj*** InObj)
int GetObjNo()

Messages

[connect] **“mix”**
[connect] **“disconnect”**

Details

construction

Mixer()
Mixer(**int** ObjNo, **SndObj**** InObjs, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

These methods construct an object of the Mixer class. The default constructor has no inputs assigned to it, so the AddObj() should be used to add objects to the input list. The other constructor has the following parameters:

int ObjNo: number of pointers to input objects in the array.
SndObj** InObjs: an array of pointers to the locations of SndObj-derived objects.
int vecsize: vector size in samples. Size of the internal DSP buffer, defaults to DEF_VECSIZE, 256.
float sr: sampling rate in HZ. Defaults to DEF_SR, 44100.f.

public methods

Mixer works with a list of SndObj-derived objects whose output is mixed together. The main methods used with this object are the ones that add or delete an item to/from the list. The method **AddObj()** (and the connect message **“mix”**) add an object to the list, whereas **DeleteObj()** will take that object from the list (as will the connect message **“disconnect”**). In addition, the object provide the method **GetObjNo()** which will return the number of items in the mix list.

Examples

Usually, Mixer objects are created using their default constructor, as ‘empty’ objects. The mix list is then created by invoking **AddObj()**. Alternatively, a mix list could be created as an array of pointers to SndObj-derived objects and this can be passed to the constructor. The following is an example of the typical use of Mixer:

```
Mixer mix;
mix.AddObj(&inObj1);
mix.AddObj(&inObj2);

while(processing_on){

inObj1.DoProcess();
inObj2.DoProcess();
mix.DoProcess();
output.Write();

}
```

Class NoteTable

Description

The NoteTable class implements a conversion table for MIDI note numbers to equal-tempered frequencies. The user can set the frequency and note intervals, which will be used as the basis to calculate the equal-tempered conversion scale. The size of the table is set to 128.

Construction

NoteTable()
NoteTable(**short** lowernote, **short** uppernote, **float** lowerfreq, **float** upperfreq)

Public Methods

void SetNoteInterval(**short** lowernote, **short** uppernote)
void SetFreqInterval(**float** lowerfreq, **float** upperfreq)

Details

construction

NoteTable()
NoteTable(**short** lowernote, **short** uppernote, **float** lowerfreq, **float** upperfreq)

Constructs a NoteTable object. It builds a conversion table based on a note interval and a frequency interval. The latter provides the basic interval to be (logarythmically) subdivided. The base MIDI note interval provides the tuning reference for the conversion table and the number of equal steps into which the base frequency interval is subdvided. The table is extend to comprehend the whole range of MIDI notes (0 -127). The default constructor creates a 12-note equal-tempered conversion table, with A3 tuned to 440Hz (lowernote = 69, uppernote = 81, lowerfreq = 440, upperfreq = 880).

short lowernote: lowermost MIDI note of the base interval.
short uppernote: uppermost MIDI note of the base interval.
float lowerfreq: frequency of the lowermost note of the base interval.
float upperfreq: frequency of the uppermost note of the base interval.

public methods

void SetNoteInterval(**short** lowernote, **short** uppernote)
void SetFreqInterval(**float** lowerfreq, **float** upperfreq)

These methods set the function table parameters. MakeTable() should be invoked after any parameter resetting.

Class Osc

Description

Osc implements a truncating table oscillator which works exclusively with power-of-two size tables (for other table sizes see class Oscilt). Time-varyng signals can be used to control the amplitude and frequency of this oscillator.

Construction

Osc()
Osc(**Table*** table, **float** fr, **float** amp, **SndObj*** inputfr,
    SndObj* inputamp = 0, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Public Methods

void SetFreq(**SndObj*** inputfr)
void SetAmp(**SndObj*** inputamp)

Messages

[connect] **"frequency"**
[connect] **"amplitude"**

Details

construction

Osc()
Osc(**Table*** table, **float** fr, **float** amp, **SndObj*** inputfr,
    SndObj* inputamp = 0, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

These methods construct an object of the Osc class. Construction parameters are:

Table* table: pointer to the location of a Table-derived object.
float fr: fundamental frequency offset, in Hz. Initialised to 440 by the default constructor.
float amp: amplitude offset. Initialised to 1.f .
SndObj* inputfr: frequency control input, pointer to the location of a SndObj-derived object. The fundamental frequency can be controlled by a time-varying signal from another SndObj-derived object. A signal is fed from the input object and added to the frequency offset value. Defaults to 0, *no frequency input object*
SndObj* inputamp: amplitude control input, pointer to the location of a SndObj-derived object. Defaults to 0, which means *no amplitude input object*, so the amplitude is fixed to the amplitude offset value. This is added to the amplitude control input signal when a SndObj-derived object is patched in.
int vecsize: vector size in samples. Size of the internal DSP buffer, defaults to DEF_VECSIZE, 256.
float sr: sampling rate in HZ. Defaults to DEF_SR, 44100.f.

public methods

void SetFreq(**SndObj*** inputfr)
void SetAmp(**SndObj*** inputamp)

These two methods can be invoked to connect an input object to the frequency and amplitude inputs, respectively. Their functionality is replicated by the connect messages **"frequency"** and **"amplitude"**.

Examples

Osc is a simple truncating oscillator, with time-varying amplitude and frequency. The only restriction it has is that (as with its parent class FastOsc) it requires a power-of-two table, as it uses integer (instead of floating-point) sampling increments.

```
HarmTable sinobj(1024, SINE,1);
Osc  mod(&sinobj,  4.3f,  8000.f)
Osc  car(&sinobj, 100.f, 8000.f, 0, &mod);
```

The example above shows a simple AM setup, where the Osc object **mod** is connected to the amplitude input of the second Osc object, **car**.

```
while(processing_on){

mod.DoProcess();
car.DoProcess();
output.Write();

}
```

Class Osci

Description

Osci implements an interpolating table oscillator which works exclusively with power-of-two size tables (for other table sizes see class Oscili). Time-varyng signals can be used to control the amplitude and frequency of this oscillator.

Construction

Osci()
Osci(**Table*** table, **float** fr, **float** amp, **SndObj*** inputfr,
    SndObj* inputamp = 0, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Details

construction

Osci()
Osci(**Table*** table, **float** fr, **float** amp, **SndObj*** inputfr,
    SndObj* inputamp = 0, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

These methods construct an object of the Osci class. Construction parameters are:

Table* table: pointer to the location of a Table-derived object.
float fr: fundamental frequency offset, in Hz. Initialised to 440 by the default constructor.
float amp: amplitude offset. Initialised to 1.f .
SndObj* inputfr: frequency control input, pointer to the location of a SndObj-derived object. The fundamental frequency can be controlled by a time-varying signal from another SndObj-derived object. A signal is fed from the input object and added to the frequency offset value. Defaults to 0, *no frequency input object*
SndObj* inputamp: amplitude control input, pointer to the location of a SndObj-derived object. Defaults to 0, which means *no amplitude input object*, so the amplitude is fixed to the amplitude offset value. This is added to the amplitude control input signal when a SndObj-derived object is patched in.
int vecsize: vector size in samples. Size of the internal DSP buffer, defaults to DEF_VECSIZE, 256.
float sr: sampling rate in HZ. Defaults to DEF_SR, 44100.f.

Examples

Osci is an interpolating oscillator, with time-varying amplitude and frequency. The only restriction it has is that (as with its parent class Osc) it requires a power-of-two table, as it uses integer (instead of floating-point) sampling increments.

HarmTable sinobj(1024, SINE,1);
Osci mod(&sinobj, fm, index*fm)
Osci car(&sinobj, fc, amp, &mod);

The example above shows a simple FM setup, where the Osci object **mod,** with frequency **fm** and amplitude **index*fm**, is connected to the frequency input of the second Osci object, **car**.

```
while(processing_on){
mod.DoProcess();
car.DoProcess();
output.Write();
}
```

Class Oscil

Oscil is a basic (but fast) oscillator, using truncating lookup. Increment can only be positive. It samples its input signals once every time DoProcess() is called, effectively at a lower SR. This has two consequences: 1) input objects can be run with a vector size of 1 and a slower SR (by a factor of 1/vector_size) and 2) the lower SR also makes it awkward to modulate the amplitude/frequency at audio rates.

Construction

Oscil()
Oscil(**Table*** table, **float** fr=440.f, **float** amp=1.f, **SndObj*** inputfreq = 0, **SndObj*** inputamp = 0, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Public Methods

void SetFreq(**float** fr, **SndObj*** inputfreq = 0)
void SetAmp(**float** amp, **SndObj*** inputamp = 0)
void SetFreq(**SndObj*** inputfreq = 0)
void SetAmp(**SndObj*** inputamp = 0)
void SetPhase(**float** phase)
void SetTable(**Table*** table)

Messages

[set, connect] **“frequency”**
[set, connect] **“amplitude”**
[set] **“phase”**
[connect] **“table”**

Details

construction

Oscil()
Oscil(**Table*** table, **float** fr=440.f, **float** amp=1.f, **SndObj*** inputfreq = 0, **SndObj*** inputamp = 0, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

These methods construct an object of the Oscil class. Construction parameters are:

Table* table: pointer to the location of a Table-derived object.
float fr: fundamental frequency offset, in Hz. Initialised to 440 by the default constructor.
float amp: amplitude offset. Initialised to 1.f .
SndObj* inputfr: frequency control input, pointer to the location of a SndObj-derived object. The fundamental frequency can be controlled by a time-varying signal from another SndObj-derived object. A signal is fed from the input object and added to the frequency offset value. Defaults to 0, *no frequency input object*
SndObj* inputamp: amplitude control input, pointer to the location of a SndObj-derived object. Defaults to 0, which means *no amplitude input object*, so the amplitude is fixed to the amplitude offset value. This is added to the amplitude control input signal when a SndObj-derived object is patched in.
int vecsize: vector size in samples. Size of the internal DSP buffer, defaults to DEF_VECSIZE, 256.
float sr: sampling rate in HZ. Defaults to DEF_SR, 44100.f.

public methods

void SetFreq(**float** fr, **SndObj*** inputfreq = 0)
void SetAmp(**float** amp, **SndObj*** inputamp = 0)
void SetFreq(**SndObj*** inputfreq = 0)

void SetAmp(**SndObj*** inputamp = 0)
void SetPhase(**float** phase)
void SetTable(**Table*** table)

These methods set the parameters that make up a Oscil object. Frequency is set in Hz, amplitude is arbitrary (it depends on the scaling/precision used) and the phase is in fractions of a cycle (0-1.0). The input frequency and amplitude objects can also be connected to this object using **SetFreq()** and **SetAmp()** respectively. **SetTable()** connects a table object to this oscillator.

Examples

Oscil can be used to generate any type of periodic (and one-shot) signals. One typical use is to generate a pitched signal by scanning a table containing a certain wave shape. Any size table can be used.

```
HarmTable sinobj(2500, SINE,1);
Oscil  oscilla(&sinobj, 100.f, 16000.f);
```

The example above shows an object named *oscilla* which is connected to a table object *sinobj*. The object will generate a sinewave signal with 100Hz frequency and amplitude 16000.f (ca. –6dB in 16-bit):

```
while(processing_on){

oscilla.DoProcess();
output.Write();

}
```

Class Oscili

Oscili is an interpolating oscillator, derived from Oscil, providing full audio rate modulation capabilities.

Construction

Oscili()
Oscili(**Table*** table, **float** fr=440.f, **float** amp=1.f, **SndObj*** inputfreq = 0, **SndObj*** inputamp = 0, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Details

construction

Oscili()
Oscili(**Table*** table, **float** fr=440.f, **float** amp=1.f, **SndObj*** inputfreq = 0, **SndObj*** inputamp = 0, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

These methods construct an object of the Oscili class. Construction parameters are:

Table* table: pointer to the location of a Table-derived object.
float fr: fundamental frequency offset, in Hz. Initialised to 440 by the default constructor.
float amp: amplitude offset. Initialised to 1.f .
SndObj* inputfr: frequency control input, pointer to the location of a SndObj-derived object. The fundamental frequency can be controlled by a time-varying signal from another SndObj-derived object. A signal is fed from the input object and added to the frequency offset value. Defaults to 0, *no frequency input object*
SndObj* inputamp: amplitude control input, pointer to the location of a SndObj-derived object. Defaults to 0, which means *no amplitude input object*, so the amplitude is fixed to the amplitude offset value. This is added to the amplitude control input signal when a SndObj-derived object is patched in.
int vecsize: vector size in samples. Size of the internal DSP buffer, defaults to DEF_VECSIZE, 256.
float sr: sampling rate in HZ. Defaults to DEF_SR, 44100.f.

Examples

Oscili is an interpolating oscillator, with time-varying amplitude and frequency.

HarmTable sinobj(2500, SINE,1);
Oscili mod(&sinobj, fm, index*fm)
Oscili car(&sinobj, fc, amp, &mod);

The example above shows a simple FM setup, where the Osci object **mod,** with frequency **fm** and amplitude **index*fm**, is connected to the frequency input of the second Osci object, **car**.

```
while(processing_on){
mod.DoProcess();
car.DoProcess();
output.Write();
}
```

Class Oscilt

Oscilt is an interpolating oscillator, derived from Oscil, providing full audio rate modulation capabilities.

Construction

Oscilt()
Oscilt(**Table*** table, **float** fr=440.f, **float** amp=1.f, **SndObj*** inputfreq = 0, **SndObj*** inputamp = 0, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Details

construction

Oscilt()
Oscilt(**Table*** table, **float** fr=440.f, **float** amp=1.f, **SndObj*** inputfreq = 0, **SndObj*** inputamp = 0, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

These methods construct an object of the Oscilt class. Construction parameters are:

Table* table: pointer to the location of a Table-derived object.
float fr: fundamental frequency offset, in Hz. Initialised to 440 by the default constructor.
float amp: amplitude offset. Initialised to 1.f .
SndObj* inputfr: frequency control input, pointer to the location of a SndObj-derived object. The fundamental frequency can be controlled by a time-varying signal from another SndObj-derived object. A signal is fed from the input object and added to the frequency offset value. Defaults to 0, *no frequency input object*
SndObj* inputamp: amplitude control input, pointer to the location of a SndObj-derived object. Defaults to 0, which means *no amplitude input object*, so the amplitude is fixed to the amplitude offset value. This is added to the amplitude control input signal when a SndObj-derived object is patched in.
int vecsize: vector size in samples. Size of the internal DSP buffer, defaults to DEF_VECSIZE, 256.
float sr: sampling rate in HZ. Defaults to DEF_SR, 44100.f.

Examples

Oscilt is a simple truncating oscillator, with time-varying amplitude and frequency.

```
HarmTable sinobj(1024, SINE,1);
Osc  mod(&sinobj,  150.f,  8000.f)
Osc  car(&sinobj, 270.f, 8000.f, 0, &mod);
```

The example above shows a simple audio-rate AM setup, where the Oscil object **mod** is connected to the amplitude input of the second Oscil object, **car**.

```
while(processing_on){

mod.DoProcess();
car.DoProcess();
output.Write();

}
```

Class Pan

Description

The Pan object pans an input signal between two channels. The stereo output of this object is made available in the form of two pointers to a **SndObj** class, by the public member variables **left** and **right**

Construction

Pan()
Pan(**float** pan, **SndObj*** InObj, **SndObj*** InPan = 0, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Public Members

SndObj* right
SndObj* left

Public Methods

void **SetPan**(float pan, SndObj* InPan=0)

Messages

[set, connect] **"pan position"**

Details

construction

Pan()
Pan(**float** pan, **SndObj*** InObj, **SndObj*** InPan = 0, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

These methods construct an object of the Pan class. Construction parameters are:

float pan: pan position offset, -1 means hard left and +1, hard right.
SndObj* InObj: input object, pointer to the location of a SndObj-derived object.
SndObj* InPan: variable pan input object, pointer to the location of a SndObj-derived object. A signal varying between -1 and 1 will move the sound between the two speakers.
int vecsize: vector size in samples. Size of the internal DSP buffer, defaults to DEF_VECSIZE, 256.
float sr: sampling rate in HZ. Defaults to DEF_SR, 44100.f.

public members

SndObj* right
SndObj* left

These are pointers to the two output objects generated by pan. They are used to obtain the signal for the two output channels, right and left.

public methods

void **SetPan**(float pan, SndObj* InPan=0)

This method (and the message **"pan position"**) can be used to set the fixed pan offset value and/or the variable pan input signal object.

Examples

Pan generates a two-channel signal out of a single-channel input. The relative amplitudes of the channels are determined by the pan position:

Pan panpot(0.5, &inObj);

The example above shows a signal being panned halfway between the centre and the right speaker on a normal stereo set-up. The two signals, left and right are obtained as two SndObj pointers:

```
output.SetOutput(0, panpot.left);
output.SetOutput(1. panpot.right);

while(processing_on){

inObj.DoProcess();
panpot.DoProcess();
output.Write();

}
```

Class Phase

Description

The Phase class implements a phase accumulator, aka phasor. It generates a constantly moving phase value between 0 and 1, according to its frequency.

Construction

Phase()
Phase(**float** freq, **SndObj*** FreqInput = 0 , **float** offset = 0.f ,**int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Public Methods

void SetFreq(**float** freq, **SndObj*** FreqInput = 0)
void SetPhase(**float** offset)

Messages

[set, connect] **"frequency"**
[set] **"phase"**

Details

construction

Phase()
Phase(**float** freq, **SndObj*** FreqInput = 0 , **float** offset = 0.f ,**int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

These methods construct an object of the Phase class. Construction parameters are:

float freq: phasor frequency, in Hz.
SndObj* FreqInput: pointer to a SndObj object, whose signal will be used to control the phasor frequency.
float offset: initial phase, in fractions of a cycle (0-1.0).
int vecsize: vector size in samples. Size of the internal DSP buffer, defaults to DEF_VECSIZE, 256.
float sr: sampling rate in HZ. Defaults to DEF_SR, 44100.f.

public methods

void SetFreq(**float** freq, **SndObj*** FreqInput = 0)
void SetPhase(**float** offset)

These methods can be used to set the fixed frequency and phase offset value or to connect an input frequency signal object.

Examples

A Phase object simply generates a moving phase between 0 and 1:

Phase phi(440.f);
Lookup sig(&sinetable, 0.f, &phi, WRAP_AROUND, NORMALISED);

The example above shows a Phase object used to implement a truncating lookup oscillator, in conjuction with a table reader.

```
while(processing_on){

phi.DoProcess();
sig.DoProcess();
output.Write();

}
```

Class PhOscili

Description

PhOscili is an inteprolating oscillator with a variable phase increment input, enabling phase modulation. This class was written by Frank Barknecht.

Construction

PhOscili()
PhOscili(**Table*** table, **float** fr=440.f, **float** amp=1.f, **SndObj*** inputfreq = 0, **SndObj*** inputamp = 0, SndObj* inputphase = 0, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Messages

[set, connect] **"phase"**

Details

construction

PhOscili()
PhOscili(**Table*** table, **float** fr=440.f, **float** amp=1.f, **SndObj*** inputfreq = 0, **SndObj*** inputamp = 0, SndObj* inputphase = 0, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

These methods construct an object of the PhOscili class. Construction parameters are:

Table* table: pointer to the location of a Table-derived object.
float fr: fundamental frequency offset, in Hz. Initialised to 440 by the default constructor.
float amp: amplitude offset. Initialised to 1.f .
SndObj* inputfr: frequency control input, pointer to the location of a SndObj-derived object. The fundamental frequency can be controlled by a time-varying signal from another SndObj-derived object. A signal is fed from the input object and added to the frequency offset value. Defaults to 0, *no frequency input object*
SndObj* inputamp: amplitude control input, pointer to the location of a SndObj-derived object. Defaults to 0, which means *no amplitude input object*, so the amplitude is fixed to the amplitude offset value. This is added to the amplitude control input signal when a SndObj-derived object is patched in.
SndObj* inputphase phase increment control input, pointer to the location of a SndObj-derived object. Defaults to 0, which means *no phase input object*. The expected phase signal should be in the range of –1 to 1, ie. fractions of a full cycle.
int vecsize: vector size in samples. Size of the internal DSP buffer, defaults to DEF_VECSIZE, 256.
float sr: sampling rate in HZ. Defaults to DEF_SR, 44100.f.

Examples

PhOscili can be used to generate any type of periodic (and one-shot) signals, in addition it enables phase modulation (PM).

HarmTable sinobj(2500, SINE,1);
Oscili mod(&sinobj, fm, index);
PhOscili car(&sinobj, fc, amp, 0, 0, &mod);

The example above shows a typical PM set-up, where a modulating oscillator signal is inserted into the carrier phase signal.

```
while(processing_on){

mod.DoProcess();
car.DoProcess();
output.Write();

}
```

Class Pitch

Description

This object implements a delayline-based pitch transposer. It provides methods for setting the pitch both as equal-tempered semitones or as frequency ratios.

Construction

Pitch()
Pitch(**float** delaytime, **SndObj*** InObj, **float** pitch = 1.f, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)
Pitch(**float** delaytime, **SndObj*** InObj, **int** semitones = 0, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Public Methods

void SetPitch(**float** pitch)
void SetPitch(**int** semitones)

Messages

[set] **"multiplier"**
[set] **"semitones"**

Details

construction

Pitch()
Pitch(**float** delaytime, **SndObj*** InObj, **float** pitch = 1.f, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)
Pitch(**float** delaytime, **SndObj*** InObj, **int** semitones = 0, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

These methods construct an object of the Pitch class. Construction parameters are:

float delaytime: max delay time of the buffer, in seconds (normally 0.1)
SndObj* InObj: input object, pointer to the location of a SndObj-derived object. The output signal from this object is added to the offset value and then used as the index on the lookup process.
float pitch: pitch transposition ratio.
int semitones: pitch transposition in semitones.
int vecsize: vector size in samples. Size of the internal DSP buffer, defaults to DEF_VECSIZE, 256.
float sr: sampling rate in HZ. Defaults to DEF_SR, 44100.f.

public methods

void SetPitch(**float** pitch)
void SetPitch(**int** semitones)

These two methods are used to adjust the output pitch, either as a frequency ratio (mulitplier) or as a semitone value. The two set messages, **"multiplier"** and "**semitones"** can also be sent to the object to change the pitch according to a frequency ratio or equal-tempered semitones, respectively.

Examples

Pitch is basically a variable-delay delayline, where the relative speeds of the read pointer in relation to the write pointer determines the amount of pitch transposition. In order to avoid echo-like effects, the delay is kept to around 0.1 secs. Shorter delay times might cause audible aliasing effects when transposition ratios are large.

Pitch transpo(0.1f, &inObj, 1.5f);

The example above transposes the input signal to a perfect fifth above the original, whereas

Pitch transpo(0.1f, &inObj, 7);

transposes the signal to a perfect equal-tempered fifth (7 semitones, 1.498 times) above.

```
while(processing_on){

inObj.DoProcess();
transpo.DoProcess();
output.Write();

}
```

Class PlnTable

Description

The HammingTable class draws a polynomial of any order, centred on 0, over a specified range.

Construction

PlnTable()
PlnTable(**long** L, **int** order, **double*** coefs, **float** range=1.f)

Public Methods

void SetPln(**int** order, **double*** coefs, **float** range=1.f)

Details

construction

PlnTable()
PlnTable(**long** L, **int** order, **double*** coefs, **float** range=1.f)

Constructs a PlnTable object.

long L: table length.
int order: order of the polynomial.
double* coefs: pointer to the first location of a an array of double-precision floats containing the polynomial coeficients.
float range: range of the polynomial, interval over which it will be drawn. Defaults to 1.

public methods

void SetPln(**int** order, **double*** coefs, **float** range=1.f)

This method sets the function table parameters. MakeTable() should be invoked after any parameter resetting.

Class Pluck

Description

The Pluck object is a plucked-string sound generator, based on the Karplus-Strong model. It takes parameters for frequency (offset and freq. control object), amplitude, feedback gain/decay factor, sampling rate and maxscale (max positive amplitude value in the system). The "string" is "plucked" when DoProcess() is called for the first time after construction and it will be replucked after calls to SetAmp() or RePluck().

Construction

Pluck()
Pluck(**float** fr, **float** amp, **float** fdbgain, **SndObj*** InFrObj = 0,**float** maxscale=32767.f, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)
Pluck(**float** fr, **float** amp, **SndObj*** InFrObj = 0, **float** decay=20.f, float maxscale=32767.f, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Public Methods

void SetAmp(**float** amp, **float** maxscale=32767)
void RePluck();

Messages

[set] **“amplitude”**
[set] **“maxscale”**
[set] **“repluck”**

Details

construction

Pluck()
Pluck(**float** fr, **float** amp, **float** fdbgain, **SndObj*** InFrObj = 0,**float** maxscale=32767.f, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)
Pluck(**float** fr, **float** amp, **SndObj*** InFrObj = 0, **float** decay=20.f, float maxscale=32767.f, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

These methods construct an object of the Pluck class. Construction parameters are:

float fr: frequency offset, in Hz. Defaults to 440 Hz.
float amp: amplitude of the noise signal which will fill the delay line. Defaults to 1.0 .
float fdbgain: gain factor of the internal comb filter, which will rescale the signal before it re-enters the delay line. Normally < 1, anything over 1 will cause the signal to continually grow, with possibly disastrous results. Defaults to 0.9 .
float decay: alternatively, the third constructor constructs an object whose decay factor can be directly controlled. The decay factor is given in dB/sec. This allows for stretching as well as shortening the decay, as well as maintaining the same decay time across all frequencies.
SndObj* InFrObj: frequency control input, pointer to the location of a SndObj-derived object. The fundamental frequency can be controlled by a time-varying signal from another SndObj-derived object. A signal is fed from the input object and added to the frequency offset value. Defaults to 0, *no frequency input object*
float maxscale: max positive amplitude possible in the system, 32767.f for 16-bit audio. Defaults to 32767.
int vecsize: vector size in samples. Size of the internal DSP buffer, defaults to DEF_VECSIZE, 256.
float sr: sampling rate in HZ. Defaults to DEF_SR, 44100.f.

public methods

void SetAmp(**float** amp, **float** maxscale=32767)
void RePluck();

These methods set the parameters of a Pluck object, the amplitude, the maximum scale (which is the max positive amplitude value used by the system), as well as setting it to be plucked again. The respective set messages for these actions are **“amplitude”, “maxscale”** and “**repluck”** (this one ignores the message argument, as it does not have a use for it).

Examples

Pluck is basically a string-sound generator. The parameters used to control its sound are amplitude, frequency and feedback gain. The first two are self-explanatory, the last one basically controls how the modelled string wave bounces off the ends of the string, so it affects the decay time of the sound. Values are usually set above 0.9 and below 1, the lower frequencies tend to have a slower decay time than the higher ones.

Pluck string(220.f, 10000.f, 0.95f);

The above example creates a Pluck object with a fundamental at 220 Hz. When Pluck::DoProcess() is called for the first time, the string will be plucked and its sound will decay until the RePluck() or SetAmp() methods (or associated messages) are invoked.

```
while(processing_on){

string.DoProcess();
output.Write();

}
```

Class PVA

Description

The PVA class provides the mechanism for Phase Vocoder Analysis. It takes a time-domain signal and transforms it into a sequence of frequency-domain frames, each one containing N/2 pairs of values, where N is the FFT framesize. Each frame corresponds to a time point in the original signal that is *hopsize* samples ahead of the previous frame. Except for the first pair of values, all bins will contain the measured amplitude (magnitude) and frequency (in Hz) of each analysis band. The first pair of values will contain only the amplitudes of the 0Hz and Nyquist components, respectively, of the signal. As with its parent class, FFT, the limitations for hopsize and fftsize are as follows: (a) the hopsize has to match the time-domain vector size used by the input object (b) the FFT size has to be a power-of-two multiple of the hopsize. As a consequence of this FFT objects (and other spectral processing objects) can be freely combined with time-domain objects in the same processing loop (and within a SndThread object). The FFT size determines the size of the output vector and the hopsize the time interval (in samples) between successive FFT frames.

Construction

PVA()
PVA(**Table*** window, **SndObj*** input, **float** scale=1.f, **int** fftsize=DEF_FFTSIZE, **int** hopsize=DEF_VECSIZE, **float** m_sr=DEF_SR)

Public Methods

float Outphases(**int** pos)

Details

construction

PVA()
PVA(**Table*** window, **SndObj*** input, **float** scale=1.f, **int** fftsize=DEF_FFTSIZE, **int** hopsize=DEF_VECSIZE, **float** m_sr=DEF_SR)

These methods construct a PVA object. Its parameters are:

Table* window: pointer to a Table-derived object containing a window shape to be used in the analysis.
SndObj* input: input signal object (SndObj-derived). Its vector size should match the hopsize set for this PVA object.
float scale: scaling factor. The overall scaling, after transformation is *scale/N*, where *N* is the FFT size.
int fftsize: the FFT size, the number of frequency points in the analysis, which will also determine the output vector size. Defaults to DEF_FFTSIZE (1024).
int hopsize: the hopsize, or decimation, which determines the number of samples in between successive FFT analysis frames. Defaults to DEF_VECSIZE (256).
float sr: the sampling rate for this object. Defaults to DEF_SR (44100.f).

public methods

float Outphases(**int** pos)

This method provides access to the phase value for each bin, indexed from the 0 Hz bin up to, but not including, the Nyquist. Since the 0 Hz and Nyquist frequencies are always purely real, their phase information is irrelevant. Also note that the phase value output by this method is the one used to obtain the phase difference used to estimate the bin frequency value. As such, as a rotation process is applied to the analysed signal prior to the transform,

this value will be different to that obtained from a straightforward FFT followed by a cartesian-to-polar conversion. The single argument to the method is the bin number, which will correspond to the desired analysis frequency band.

Examples

PVA objects are used to transform a time-domain signal into a frequency-domain signal. The resulting output is a series of spectral frames generated every hopsize/SR seconds (after a call to PVA::DoProcess()). These frames will contain fftsize/2 + 1 frequency points between 0 and SR/2 (inclusive). With the exception of 0 and SR/2 Hz, each frequency point is a pair of values containg the amplitude and frequecny values for that frequency band. In order to fit all points in a single fftsize vector, the real parts of the 0 and SR/2 Hz points are packed together in the first two positions of the array.

PVA analysis(&winobj, &inobj);

Its output can be further transformed by a spectral processing object and the result can be then transformed back into the time-domain (using **IFFT)**. The example below multiplies two spectra:

```
PVA spec1(&winobj, &inobj1);
PVA spec2(&winobj, &inobj2);
PVMorph mph(0.5f, 0.5f,&spec1, &spec2);
PVS tdsig(&winobj, &mph);
```

This transforms two inputs (from *inobj1* and *inobj2*), morphs their frequency/amplitude spectra at 50% and resynthesises it.

```
while(processing_on) {

inobj1.DoProcess();
inobj2.DoProcess();
spec1.DoProcess();
spec2.DoProcess();
mph.DoProcess();
tdsig.DoProcess();
output.Write();

}
```

See the FFT class for more information on spectral analysis classes.

Class PVBlur

Description

This class implements time blurring of PV data. It takes an input from a PV-data generating SndObj and blurs its amplitude/frequency values on a channel-per-channel basis. The main parameter is the blurring time or period, in seconds. Because blurring depends on accessing successive PV frames, there is a time delay involved, which is equivalent to the blurring period.

Construction

PVBlur()
PVBlur(**SndObj*** input, **float** blurtime, **int** hopsize=DEF_VECSIZE, **int** vecsize=DEF_FFTSIZE, **float** sr=DEF_SR)

Details

construction

PVBlur()
PVBlur(**SndObj*** input, **float** blurtime, **int** hopsize=DEF_VECSIZE, **int** vecsize=DEF_FFTSIZE, **float** sr=DEF_SR)

These methods construct a PVBlur object:

SndObj* input: spectral SndObj, generating a PV-format signal.
float blurtime: the blurring period in seconds.
int hopsize: the original analysis hopsize, used to calculate the number of blurred frames.
int vecsize: vector size in samples. Size of the internal DSP buffer, defaults to DEF_VECSIZE, 256.
float sr: sampling rate in HZ. Defaults to DEF_SR, 44100.f.

Examples

PVBlur can be used for sound transformation applications, its effect is that of a time-smearing of amps and freqs. In realtime operation, it is important to note that there is also an extra delay between input and output, equivalent to the blurring period. The example below implements blurring of an input signal.

```
SndRTIO input(1, SND_INPUT);
SndRTIO output(1, SND_OUTPUT);

 HammingTable window(1024, 0.54f);

 SndIn in(&input);
 PVA anal(&window, &in);
 PVBlur blur(&anal, pitch);
 PVS synth(&window, &blur);

 output.SetOutput(1, &synth);
```

Here's a processing loop for it:

```
 for(int i=0; i<end; i++){
   input.Read();
   in.DoProcess();
```

```
  anal.DoProcess();
  blur.DoProcess();
  synth.DoProcess();
  output.Write();
}
```

Class PVEnvTable

Description

The PVEnvTable object builds a function table based on a spectral magnitude envelope. The envelope is defined by a starting point and two arrays: (1) segment legnths and (2) end points of each segment. The segments can be either exponential or linear. The table will contain a spectrum consisting of magnitude, frequency pairs for each positive DFT point, except for the 0Hz and Nyquist points, which are magnitude-only. Table sizes also determine the DFT size used and generally are set to a power-of-two value. The spectal table data format is the same employed by the SndObj spectral (PV) classes: 0Hz and Nyquist points, followed by all other spectral points from 1 to N/2 –1. The frequency values are set to bin centre frequencies. The table values are not normalised.

Construction

PVEnvTable()
PVEnvTable(**long** L**, int** segments**, float** start**,**
float* points**, float*** lengths**,float** type = 0.f**,**
float phi=0.f**, float** sr=44100.f**,float** nyquistamp=0.f)

Public Methods

void SetEnvelope(**int** segments, **float** start, **float*** points, **float*** lengths,
loat type, **float** nyquistamp)
void SetSr(**float** sr)

Details

construction

PVEnvTable()
PVEnvTable(**long** L**, int** segments**, float** start**,**
float* points**, float*** lengths**,float** type = 0.f**,**
float sr=44100.f**,float** nyquistamp=0.f)

Constructs a PVEnvTable object.

long L: table length.
int segments: number of envelope segments.
float start: starting value of envelope (0 Hz magnitude).
float* points: an array of floats, containing the end values of each segment. Must match the above number of segments.
float* lengths: an array of floats, containing the lengths of each segment. Must match the above number of segments. Segment lengths are normalised to the table size (added up and then each one is divided by that total and multiplied by the table size).
float type: type of curve. Linear = 0, inverse exponential < 0 < exponential.
float sr: sampling rate.
float nyquistamp: Nyquist magnitude.

public methods

void SetEnvelope(**int** segments, **float** start, **float*** points, **float*** lengths,
loat type, **float** nyquistamp)

This method sets the envelope parameters. MakeTable() is invoked by this method.

void SetSr(**float** sr)

This method sets the sampling rate and writes the frequency values of each spectral point.

Class PVMask

Description

PVMask takes a PV input and alter its amplitudes, on a per-channel basis, depending on a masking magnitude spectrum of another input or a table. The masking operation compares the amplitudes of the input with that of the masking spectral signal, if the input falls below the spectral signal, then the amplitude is scaled by a masking gain value. When the scaling value is 0, the masking is complete and all channels with less energy than the masking signal channels are eliminated. When the scaling is above 1, then all components of the mask that are stronger than the input signal will impose their amplitudes on that signal. The scaling value can be modulated by an input signal. The masking spectrum can be either fixed (stored in an fftsize-sized table in the PV format) or time-varying (from an input SndObj). Input frequencies are untouched.

Construction

PVMask()
PVMask(**float** maskgain, **SndObj*** input, **SndObj*** mask,
SndObj* inmaskgobj=0, **int** vecsize=DEF_FFTSIZE,
float sr=DEF_SR)
PVMask(**float** maskgain, **Table*** masktable, **SndObj*** input,
SndObj* inmaskgobj=0, **int** vecsize=DEF_FFTSIZE,
float sr=DEF_SR)

Public Methods

void SetMaskInput(**SndObj*** mask)
void SetMaskTable(**Table** *mask)
void SetMaskGain(**float** maskgain, **SndObj*** inmaskg=0)

Messages

[set, connect] **“mask gain”**
[connect] **“mask input”**
[connect] **“mask table”**

Details

construction

PVMask()
PVMask(**float** maskgain, **SndObj*** input, **SndObj*** mask,
SndObj* inmaskgobj=0, **int** vecsize=DEF_FFTSIZE,
float sr=DEF_SR)
PVMask(**float** maskgain, **Table*** masktable, **SndObj*** input,
SndObj* inmaskgobj=0, **int** vecsize=DEF_FFTSIZE,
float sr=DEF_SR)

These methods construct a PVMask object:

float maskgain: gain value offset, used to scale the amplitudes of input channels which are lower than the mask signal amplitudes.
SndObj* input: input signal object, whose signal is in the PV (amp, freq) spectral format.
SndObj* mask: masking signal object, with an output in the same format as above.

Table* masktable: masking table object, containing a table of *vecsize* length, with a single spectral frame in the PV format.
SndObj* inmaskgobj: mask gain signal input object, whose output will be used (in addition to maskgain) as the masking scale value.
int vecsize: signal vector size, equivalent to the FFT window size of the spectral signal.
float sr: sampling rate in Hz.

public methods

void SetMaskInput(**SndObj*** mask)
void SetMaskTable(**Table** *mask)
void SetMaskGain(**float** maskgain, **SndObj*** inmaskg=0)

These methods set the various parameters of the PVMask object. The object will take its masking input either from a Table object, if one is set or from an input SndObj. The latest call to either setting method (or Connect() with the appropriate message) will determine the masking input source.

Examples

PVMask can be used for all sorts of masking applications. One typical use is to have a noise mask source and then use PVMask to extract that noise from the input signal. For instance:

```
PVA  spec(&window, &inobj);
PVMask  clean(0.f, &noisetable, &spec);
PVS   synth(&window, &clean);
```

The above SndObj connections will take an input signal and apply a noisemask (stored in noisetable). The masking can also be time-varying, so if we use an input spectral object, we can perform such operations. The scaling gain can also be used to boost the masking amplitudes found in the input signal, if it is set above 1.

Class PVMix

Description

This class implements seamless mixing of PV data. It takes two inputs from a PV-data generating SndObjs and mixes the two by taking only the loudest channels of the two inputs.

Construction

PVMix()
PVMix(**SndObj*** input, **SndObj*** input2, **int** vecsize=DEF_FFTSIZE, **float** sr=DEF_SR)

Details

construction

PVMix()
PVMix(**SndObj*** input, **SndObj*** input2**nt** vecsize=DEF_FFTSIZE, **float** sr=DEF_SR)

These methods construct a PVMix object:

SndObj* input, input2: spectral SndObjs, generating a PV-format signal.
int vecsize: vector size in samples. Size of the internal DSP buffer, defaults to DEF_VECSIZE, 256.
float sr: sampling rate in HZ. Defaults to DEF_SR, 44100.f.

Examples

PVMix can be used for mixing applications, where the output mix will contain only the louder of the two signals for each input PV channel. The example below implements a harmoniser, mixing a transposed signal with its source.

```
SndRTIO input(1, SND_INPUT);
SndRTIO output(1, SND_OUTPUT);

 HammingTable window(1024, 0.54f);

 SndIn in(&input);
 PVA anal(&window, &in);
 PVTrans trans(&anal, pitch);
 PVMix mix(&trans, &anal);
 PVS synth(&window, &mix);

 output.SetOutput(1, &synth);
```

The above chain of SndObjs sets up a transposer whose output is mixed to its input (dry) signal. Here's a processing loop for it:

```
 for(int i=0; i<end; i++){
   input.Read();
   in.DoProcess();
   anal.DoProcess();
   trans.DoProcess();
   mix.DoProcess();
   synth.DoProcess();
   output.Write();
}
```

Class PVMorph

Description

PVMorph implements phase vocoder data interpolation. It takes two inputs from SndObj objects that output PV data (in the same format as PVA or IFGram) and interpolates frequencies and amplitudes individually. The realism of the actual morphing effect will depend very much on the qualities of the input spectra.

Construction

PVMorph()
PVMorph(**float** morphfr, **float** morpha, **SndObj*** input1, **SndObj*** input2,
SndObj* inmorphfr=0, **SndObj*** inmorpha=0, **int** vecsize=DEF_FFTSIZE,
float sr=DEF_SR)

Public Methods

void SetFreqMorph(**float** morphfr, **SndObj*** inmorphfr=0)
void SetAmpMorph(**float** morpha, **SndObj*** inmorpha=0)

Messages

[set, connect] **“amplitude morph”**
[set, connect] **“frequency morph”**

Details

construction

PVMorph()
PVMorph(**float** morphfr, **float** morpha, **SndObj*** input1, **SndObj*** input2,
SndObj* inmorphfr=0, **SndObj*** inmorpha=0, **int** vecsize=DEF_FFTSIZE,
float sr=DEF_SR)

These two methods construct a PVMorph object. Construction parameters are:

float morphfr: frequency interpolation offset, 0 takes all frequency values from input1 and 1 takes all frequency values from input2. Values in between determine the amount of frequency interpolation of the two sources.
float morpha: as above, but affecting the interpolation of amplitudes.
SndObj* input1**:** first input spectral object. Its output should be in the PVA/IFGram format (amplitude and frequency pairs, with the first pair holding the amplitude only of the 0Hz and Nyquist components).
SndObj* input2: second input spectral object, as above.
SndObj* inmorphfr: SndObj-derived object whose output will control the frequency morphing amount (offset by morphfr). Values above 1 or below 0 will be clipped.
SndObj* inmorphamp: SndObj-derived object whose output will control the ampltiude morphing amount, similarly to above.
int vecsize**:** signal vectorsize, effectively the fft window length (defaults to 1024).
float sr: sampling rate in Hz (defaults to 44100).

public methods

void SetFreqMorph(**float** morphfr, **SndObj*** inmorphfr=0)
void SetAmpMorph(**float** morpha, **SndObj*** inmorpha=0)

These two methods are used to set the frequency/amplitude interpolation parameters. These parameters can also be accessed through Set()/Connect() using the messages listed above.

Examples

The code examples below shows how a simple morphing unit generator can be built for Pure Data (the complete file is distributed in the /src/examples directory).

The constructor for the PD class would include the following SndObj code:

```
void *morph_tilde_new(t_symbol *s, int argc, t_atom *argv)
{
(...)
x->window = new HammingTable(1024, 0.5);
x->inobj1 = new SndObj(0, DEF_VECSIZE, sr);
x->inobj2 = new SndObj(0, DEF_VECSIZE, sr);
x->spec1  = new PVA(x->window, x->inobj1, 1.f, DEF_FFTSIZE,
              DEF_VECSIZE, sr);
x->spec2  = new PVA(x->window, x->inobj2, 1.f, DEF_FFTSIZE,
              DEF_VECSIZE, sr);
x->morph  = new PVMorph(morphfr, morpha, x->spec1, x->spec2,
                 0,0,DEF_FFTSIZE, sr);
x->synth  = new PVS(x->window, x->morph, DEF_FFTSIZE,
              DEF_VECSIZE, sr);
(...)
}
```

The PD class perform method would look like this:

```
  t_int  *morph_tilde_perform(int *w){
      t_sample *in1 = (t_sample*) w[1];
      t_sample *in2 = (t_sample*) w[2];
      t_sample *out = (t_sample*) w[3];
      t_int  size =   (t_int)     w[4];
      t_morph_tilde *x =
                (t_morph_tilde*)w[5];

      int pos = x->inobj1->PushIn(in1, size);
      x->inobj2->PushIn(in2, size);
      x->synth->PopOut(out, size);

      if(pos == DEF_VECSIZE){
                    x->spec1->DoProcess();
                    x->spec2->DoProcess();
                    x->morph->DoProcess();
                    x->synth->DoProcess();
          }
      return (w+6);
  }
```

Class PVRead

Description

This class reads an input PVOCEX-format amplitudet/frequency file at any specified rate, producing a time-domain signal. For multichannel files it provides a multichannel output as well as a mono sum of all channels. Separate channels can be accessed via output SndObj objects (one per channel).

Construction

PVRead();
PVRead(**char*** name, **float** timescale=1.0,**int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Public Methods

SndObj* Outchannel(**int** channel)
void SetInput(**char*** name)
void SetTimescale(**float** timescale)

Messages

[set] **“timescale”**

Details

construction

PVRead();
PVRead(**char*** name, **float** timescale=1.0,**int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Constructs a PVRead() object:

char* name: filename of a PVOCEX-format amplitude/frequency file.
float timescale: timescale control, 1 for normal play, < 1 for time stretching and > 1 for time compression.
int vecsize: synthesis vector size, defaults to 256.
int sr: sampling rate, defaults to 44100.

public methods

SndObj* Outchannel(**int** channel)

For multichannel files, this method returns the SndObj pointer associated with a particular channel. The object pointer can then be used to obtain the individual output of each channel. For instance,

```
SndObj* channel2 = specread.Outchannel(2);
output.SetOutput(2, channel2);
```

void SetInput(**char*** name)

This sets the output filename from which a PVRead object will read.

void SetTimescale(**float** timescale)

Sets the timescale for resynthesis. This parameter can be set with the message **“timescale”** and Set().

Examples

A PVRead object includes an internal SndPVOCEX object which is used to read from PVOCEX files. The DoProcess() method manipulates the readout so that any time-scale modification can be achieved. A PVRead object can be instatiated by passing it a filename to be read and, optionally, a timescale value.

```
PVRead  pvfile(“spec.pvx”, 1.2);
output.SetOutput(1, &pvfile);
```

If the file is multichannel, PVRead holds the mono sum of all channels. Individual channels can be accessed as discussed above. A processing loop for this example would look like this:

```
while(processing_on){
 pvfile.DoProcess();
 output.Write();
}
```

Class PVTransp

Description

This class implements pitch transposition of PV data. It takes an input from a PV-data generating SndObj and changes its pitch according to a fixed value or a time-vaying signal (or a combination of both). It also includes an optional operation mode whereby there is an attempt to preserve sound formants, whose success depends very much on the input spectra. This option is turned off by default.

Construction

PVTransp()
PVTransp(**SndObj*** input, **float** pitch, **int** mode=NORMAL_TRANSP,
SndObj* inpitch=0, **int** vecsize=DEF_FFTSIZE, **float** sr=DEF_SR)

Public Methods

void SetPitch(**float** pitch, **SndObj*** inpitch=0)
void SetMode(**int** mode)

Messages

[set,connect] **"pitch"**
[set] **"mode"**

Details

construction

PVTransp()
PVTransp(**SndObj*** input, **float** pitch, **int** mode=NORMAL_TRANSP,
SndObj* inpitch=0, **int** vecsize=DEF_FFTSIZE, **float** sr=DEF_SR)

These methods construct a PVTransp object:

SndObj* input: input spectral SndObj, generating a PV-format signal.
float pitch: pitch transposition factor (multiplier), or transposition interval, offset.
int mode: transposition mode, either NORMAL_TRANSP or KEEP_FORMANT.
SndObj* inpich: pitch transposition modulation input; final transposition will be a sum of this signal input with the value of the pitch offset.
int vecsize: vector size in samples. Size of the internal DSP buffer, defaults to DEF_VECSIZE, 256.
float sr: sampling rate in HZ. Defaults to DEF_SR, 44100.f.

public methods

void SetPitch(**float** pitch, **SndObj*** inpitch=0)
void SetMode(**int** mode)

These two methods set the basic PVTransp parameters, pitch transposition and mode.

Examples

PVTransp can be used for all sorts of pitch modification applications. The following example shows a simple input signal transposer:

```
SndRTIO input(1, SND_INPUT);
SndRTIO output(1, SND_OUTPUT);
```

```
 HammingTable window(1024, 0.54f);

 SndIn in(&input);
 PVA anal(&window, &in);
 PVTransp trans(&anal, pitch,formants);
 PVS synth(&window, &trans);

 output.SetOutput(1, &synth);
```

The above chain of SndObjs sets up a basic transposer. Its output can be added to the input signal for a harmoniser-like effect. Please note that due to the FFT process, there is an inherent delay of 1024 (DEF_FFTSIZE) to this process. A typical processing loop will look like this (not needed if using SndThread):

```
 for(int i=0; i<end; i++){
   input.Read();
   in.DoProcess();
   anal.DoProcess();
   trans.DoProcess();
   synth.DoProcess();
   output.Write();
}
```

Class PVS

Description

The PVS class implements an overlap-add, IFFT-based, phase vocoder resynthesis. It takes input PV data (in the PVA/IFGram format) from an input SndObj and outputs a time-domain signal.

Construction

PVS()
PVS(**Table*** window, **SndObj*** input, **int** fftsize=DEF_FFTSIZE, **int** hopsize=DEF_VECSIZE,
float sr=DEF_SR)

Details

construction

PVS()
PVS(**Table*** window, **SndObj*** input, **int** fftsize=DEF_FFTSIZE, **int** hopsize=DEF_VECSIZE,
float sr=DEF_SR)

Table* window: pointer to a Table-derived object implementing a suitable resynthesis window.
SndObj* input: pointer to a SndObj-derived object whose output, in the PVA/IFGram format, will be resynthesised by PVS.
int fftsize: the FFT size, the number of frequency points in the analysis, which will also determine the expected input vector size. Defaults to DEF_FFTSIZE (1024).
int hopsize: the hopsize, or decimation, which determines the number of samples in between successive FFT analysis frames and the time-domain output vector size. Defaults to DEF_VECSIZE (256).
float sr: the sampling rate for this object. Defaults to DEF_SR (44100.f).

Examples

PVS basically resynthesises PV data. So if we take a sound, use PVA to analyse it, PVS will reconstruct it:

```
PVA spec1(&winobj, &inobj1);
PVS tdsig(&winobj, &spec1);
```

Of course, for practical purposes, we would transform the PV data in some way before the resynthesis.

Class PVTable

Description

The PVTable object builds a spectral function table based on the phase vocoder analysis of a soundfile. It takes SndFIO-derived object linked to an open file (in READ mode) and analyses a portion of it. The table is built on an averaging of the spectral analysis over the specified time. The spectal table data format is the same employed by the SndObj spectral (PV) classes: 0Hz and Nyquist points, followed by all other spectral points from 1 to N/2 –1. The frequency values are set to bin centre frequencies. The table values are not normalised.

Construction

PVTable()
PVTable(**long** L, **SndFIO*** input, **Table*** window**,**
float start**, float** end)

Public Methods

void SetTable(SndFIO***** soundfile**, Table*** window**,**
float start**, float** end)

Details

construction

PVTable()
PVTable(**long** L, **SndFIO*** input, **Table*** window**,**
float start**, float** end)

Constructs a PVTable object.

long L: table length.
SndFIO* input: input sound, pointer to the location of a SndFIO-derived object (soundfile input).
Table* window: analysis window, a Table-derived object of the same length as this table.
float start: start position in seconds of PV analysis.
float end: end position in seconds (from start of file) of PV analysis. If beyond the end-of-file, then the soundfile will be analysed to its end.

public methods

void SetTable(SndFIO***** soundfile**, Table*** window**,**
float start**, float** end)

This method sets the function table parameters.

Class Rand

Description

The Rand class is a simple random signal (white noise) generator.

Construction

Rand()
Rand(**float** amp, **SndObj*** InAmpObj = 0, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Public Methods

short SetAmp(**float** amp, **SndObj*** InAmpObj=0)

Messages

[set,connect] **“amplitude”**

Details

construction

Rand()
Rand(**float** amp, **SndObj*** InAmpObj = 0, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

These methods construct an object of the Rand class. Construction parameters are:

float amp: amplitude offset. Initialised to 1.f .
SndObj* InAmpObj: amplitude control input, pointer to the location of a SndObj-derived object. Defaults to 0, which means *no amplitude input object*, so the amplitude is fixed to the amplitude offset value. This is added to the amplitude control input signal when a SndObj-derived object is patched in.
int vecsize: vector size in samples. Size of the internal DSP buffer, defaults to DEF_VECSIZE, 256.
float sr: sampling rate in HZ. Defaults to DEF_SR, 44100.f.

public methods

short SetAmp(**float** amp, **SndObj*** InAmpObj=0)

Sets the amplitude, returning 1 if succesfull and 0, in case of an error: Also, it is possible to change the amplitude parameters with the message “amplitude” sent to the method Set().

Examples

A Rand object generates white noise:

```
Rand noise(10000.f);
output.SetOutput(1, &noise);
```

This would generate a noise signal which is output on channel 1 of a SndIO-derived output object:

```
while(processing_on){
noise.DoProcess();
output.Write();
}
```

Class Randh

Description

The Randh object is a band-limited random signal generator. A frequency input controls how many times per second a new number is drawn from a pseudo-random routine. The drawn number is held at the output for 1/freq seconds.

Construction

Randh()
Rand(**float** freq, float amp, **SndObj*** InFrObj = 0, **SndObj*** InAmpObj = 0, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Public Methods

short SetFreq(**float** freq, **SndObj*** InFreqObj=0)

Messages

[set,connect] **“frequency”**

Details

construction

Randh()
Rand(**float** freq, float amp, **SndObj*** InFrObj = 0, **SndObj*** InAmpObj = 0, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

These methods construct an object of the Rand class. Construction parameters are:

float fr: frequency offset, in Hz. Initialised to 44100 by the default constructor.
float amp: amplitude offset. Initialised to 1.f .
SndObj* InFrObj: frequency control input, pointer to the location of a SndObj-derived object. The frequency can be controlled by a time-varying signal from another SndObj-derived object. A signal is fed from the input object and added to the frequency offset value. Defaults to 0, *no frequency input object*
SndObj* InAmpObj: amplitude control input, pointer to the location of a SndObj-derived object. Defaults to 0, which means *no amplitude input object*, so the amplitude is fixed to the amplitude offset value. This is added to the amplitude control input signal when a SndObj-derived object is patched in.
int vecsize: vector size in samples. Size of the internal DSP buffer, defaults to DEF_VECSIZE, 256.
float sr: sampling rate in HZ. Defaults to DEF_SR, 44100.f.

public methods

short SetFreq(**float** freq, **SndObj*** InFreqObj=0)

Sets the frequency, returning 1 if succesfull and 0, in case of an error: Also, it is possible to change the frequency parameters with the message “frequency” sent to the method Set().

Examples

A Randh object generates band-limited noise:

```
Rand noise(100.f, 10000.f);
output.SetOutput(1, &noise);
```

This would generate a noise signal, with most of its energy concentrated below 100 Hz. This is output on channel 1 of a SndIO-derived output object:

```
while(processing_on){
noise.DoProcess();
output.Write();
}
```

Class Randi

Description

The Randi object is a band-limited random signal generator. A frequency input controls how many times per second a new number is drawn from a pseudo-random routine. The output will interpolate linearly between successive random values. Randi produces a narrower bandwidth signal than its superclass, Randh.

Construction

Randi()
Rand(**float** freq, float amp, **SndObj*** InFrObj = 0, **SndObj*** InAmpObj = 0, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Details

construction

Randi()
Rand(**float** freq, float amp, **SndObj*** InFrObj = 0, **SndObj*** InAmpObj = 0, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

These methods construct an object of the Rand class. Construction parameters are:

float fr: frequency offset, in Hz. Initialised to 44100 by the default constructor.
float amp: amplitude offset. Initialised to 1.f .
SndObj* InFrObj: frequency control input, pointer to the location of a SndObj-derived object. The frequency can be controlled by a time-varying signal from another SndObj-derived object. A signal is fed from the input object and added to the frequency offset value. Defaults to 0, *no frequency input object*
SndObj* InAmpObj: amplitude control input, pointer to the location of a SndObj-derived object. Defaults to 0, which means *no amplitude input object*, so the amplitude is fixed to the amplitude offset value. This is added to the amplitude control input signal when a SndObj-derived object is patched in.
int vecsize: vector size in samples. Size of the internal DSP buffer, defaults to DEF_VECSIZE, 256.
float sr: sampling rate in HZ. Defaults to DEF_SR, 44100.f.

Examples

A Randi object generates band-limited noise:

```
Rand noise(100.f, 10000.f);
output.SetOutput(1, &noise);
```

This would generate a noise signal, with most of its energy concentrated below 100 Hz. This is output on channel 1 of a SndIO-derived output object:

```
while(processing_on){
noise.DoProcess();
output.Write();
}
```

Class ReSyn

Description

The ReSyn class implements sinusoidal additive resynthesis, based on a cubic interpolation algorithm. The class takes an input from a SinAnal-type class, which consists of a series of tracks containing amplitude, frequency and phase information. Tracks are identified by IDs given by the input object, which are then used to match them between hop periods. ReSyn objects can resynthesise any number of tracks up to the maximum tracks found at their input. ReSyn can modify the timescale of the resynthesis by altering the hopsize between frames and modify the pitch by scaling the frequencies on each hopsize

Construction

ReSyn()
ReSyn(**SinAnal*** input, **int** maxtracks, **Table*** table, **float** pitch=1.f, **float** scale=1.f,
float tscale=1.f, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Public Methods

void SetTimeScale(**float** scale)
void SetPitch(**float** pitch)

Messages

[set] **"timescale"**
[set] **"pitch"**

Details

construction

ReSyn()
ReSyn(**SinAnal*** input, **int** maxtracks, **Table*** table, **float** pitch=1.f, **float** scale=1.f,
float tscale=1.f, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

These methods construct a ReSyn object:

SinAnal* input: input object object, of the SinAnal type, from which the tracks to be resynthesised will be read.
int maxtracks: max resynthesis tracks, should be <= tracks generated by the input object.
Table* table: table object containing a wavetable to be used by each oscillator in the resynthesis, typically a cosine wave.
float pitch: pitch scaling of output (transposition ratio).
float scale: amplitude scaling of output.
float tscale: timescaling factor, the interpolation(synthesis hopsize):decimation ratio(analysis hopsize)
int vecsize: object vector size, also determines the synthesis hopsize between analysis frames (defaults to 256).
float sr: sampling rate in Hz (defaults to 44100).

public methods

void SetTimeScale(**float** scale)
void SetPitch(**float** pitch)

These methods set the different object parameters, such as the timescale and pitch factor. Note that the timescale factor only needs to be reset in case of change of the vector size (synthesis hopsize). These can alternatively be set using the messages listed above.

Examples

The following connections are a simple example of the use of ReSyn to resynthesise track data generated by SinAnal:

```
HarmTable table(4000, 1, 1, 0.75);    // cosine wave
HammingTable window(fftsize, 0.5); // hanning window

// input sound
SndWave input(infile,READ,1,16,0,0.f,decimation);
SndIn   insound(&input, 1, decimation);

// IFD analysis
IFGram ifgram(&window,&insound,1.f,fftsize,decimation);
// Sinusoidal analysis
SinAnal sinus(&ifgram,thresh,intracks);
// Sinusoidal resynthesis
ReSyn  synth(&sinus,outracks,&table,pitch,scale,(float) interpolation/decimation,
                    interpolation);

// output sound
SndWave output(outfile, OVERWRITE,1,16,0,0.f,interpolation);
output.SetOutput(1, &synth);
```

This code takes an input sound, from a file and passes it through the analysis process and then the data is resynthesised. The timescale change is determined by the decimation:interpolation ratio. In order to implement processing, the programmer either needs to write a loop and call the reading/writing and processing methods, or use a SndThread object, passing these objects to it. For the full program code, see src/examples/sinus.cpp.

Class Ring

Description

The Ring object is a multi-purpose ring modulator. It multiplies two inputs together and it can be used both as standard ring modulator or as a variable gain control.

Construction

Ring()
Ring(**SndObj*** InObj1, **SndObj*** InObj2, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Public Methods

short SetInput1(**SndObj*** InObj)
short SetInput2(**SndObj*** InObj)

Messages

[connect] **“input 2”**

Details

construction

Ring()
Ring(**SndObj*** InObj1, **SndObj*** InObj2, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

These methods construct an object of the Ring class. The default constructor sets the inputs to 0, so they need to be set before use. If one of the inputs is not set, the output will be silent.

SndObj* InObj1, InObj2: the two inputs, pointers to locations of SndObj-derived objects.
int vecsize: vector size in samples. Size of the internal DSP buffer, defaults to DEF_VECSIZE, 256.
float sr: sampling rate in HZ. Defaults to DEF_SR, 44100.f.

public methods

short SetInput1(**SndObj*** InObj)
short SetInput2(**SndObj*** InObj)

These two methods connect the two inputs. They can also be connected using the messages **“input”** and **“input 2”** and the Connect() method.

Examples

Ring can be used as a ring modulator with two audio-range signals or as an amplitude modulator/shaper with one of the inputs being a low-frequency / envelope signal. In a ring modulator we could have the following connections:

```
Oscili  sigobj1(&sinetable, 250.f, 10000.f);
Oscili  sigobj2(&sinetable, 300.f, 10000.f);
Ring    ringmod(&sigobj1, &sigobj2);
```

In an amplitude shaper, we could have:

```
Oscili  sigobj(&sinetable, 250.f, 10000.f);
Interp  envobj(0.f, 1.f, 2.f);
Ring    gain(&sigobj, &envobj);
```

This last example would create a linearly rising amplitude signal which would reach its maximum in 2 seconds:

```
while(processing_on){

sigobj.DoProcess();
envobj.DoProcess();
gain.DoProcess();
output.Write();

}
```

Class SinAnal

Description

The SinAnal class implements sinusoidal analysis of a spectral input. The class takes a frequency, amplitude and (unwrapped) phase input from a spectral analysis object (typically an IFGram object, but if the phase information is not relevant, also a PVA object can be used). It estimates the spectral peaks present in successive analysis according to a certain threshold. In case of permanent peaks, the object outputs 'tracks' corresponding to each peak. The tracks contain the amplitudes, frequencies and phases at each hop-period. Tracks are ordered by increasing time origin and then ascending frequency. At each hop-period, a number of tracks is output, and this number can be obtained from the object. The object output will consist of that number of tracks, each containing the three spectral parameters. Tracks are also given a unique ID to identify them between periods, for resynthesis and other purposes. IDs can be retrieved from the object with the appropriate method.

Construction

SinAnal();
SinAnal(**SndObj*** input, **float** threshold, **int** maxtracks, **int** minpoints=1,
int maxgap=3, **float** sr=DEF_SR)

Public Methods

int GetTrackID(**int** track)
int GetTracks()
void SetThreshold(**float** threshold)
void SetIFGram(**SndObj*** input);
void SetMaxTracks(**int** maxtracks);

Messages

[set] **"max tracks"**
[set] **"threshold"**

Details

construction

SinAnal();
SinAnal(**SndObj*** input, **float** threshold, **int** maxtracks, **int** minpoints=1,
int maxgap=3, **float** sr=DEF_SR)

These methods construct a SinAnal object:

SndObj* input: an input SndObj-object derived spectral analysis object, usually an IFGram object, but in certain cases, other spectral-type objects can also be used.
float threshold: analysis threshold, between 0 and 1. This determines which peaks to look for, excluding the ones with magnitude below the threshold. The threshold is set against the highest magnitude, so it cuts peaks whose amplitude is below threshold*mag_max.
int maxtracks: maximum number of output tracks at each hop period.
int minpoints: minimum number of past time points (hop periods) needed for a peak to make it into a track. Higher values will exclude short-lived peaks from the analysis. This will also imply an extra delay between input and output, because of the need to wait a number of hop-periods before peaks are output.
int maxgap: the maximum gap between time-points that peaks can have before a track is considered dead.
float **sr:** sampling rate, defaults to 44100.f.

public methods

int GetTrackID(**int** track)

This method returns the ID associated with the output track index (the argument to it). Its function is to match tracks between hop-periods.

int GetTracks()

This method returns the number of tracks at the output.

void SetThreshold(**float** threshold)

This method sets the analysis threshold (also set by the message **"threshold"**).

void SetIFGram(**SndObj*** input)

This connects the input SndObj object, which usually is an IFGram object.

void SetMaxTracks(**int** maxtracks)

This sets the maximum number of input tracks. The message **"max tracks"** has the same effect.

Examples

The following is taken from the example program **sinus** (src/examples/sinus.cpp):

```
// IFD analysis
IFGram ifgram(&window,&insound,1.f,fftsize,decimation);
// Sinusoidal analysis
SinAnal sinus(&ifgram,thresh,intracks, 1, 3);
// Sinusoidal resynthesis
SinSyn  synth(&sinus,outracks,&table,scale,interpolation);
```

This program takes an input sound analyses it and outputs the resynthesised sound, with a possible time-scale modification. It was noted that some sounds will have a more successful analysis than others. Low-frequency sounds are often problematic.

Class SinSyn

Description

The SinSyn class implements sinusoidal additive resynthesis, based on a cubic interpolation algorithm. The class takes an input from a SinAnal-type class, which consists of a series of tracks containing amplitude, frequency and phase information. Tracks are identified by IDs given by the input object, which are then used to match them between hop periods. SinSyn objects can resynthesise any number of tracks up to the maximum tracks found at their input.

Construction

SinSyn()
SinSyn(**SinAnal*** input, **int** maxtracks, **Table*** table, **float** scale=1.f,
int vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Public Methods

void SetTable(**Table*** table)
void SetMaxTracks(**int** maxtracks)
void SetScale(**float** scale)

Message

[set] **“max tracks”**
[set] **“scale”**
[connect] **“table”**

Details

construction

SinSyn()
SinSyn(**SinAnal*** input, **int** maxtracks, **Table*** table, **float** scale=1.f,
int vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

These methods construct a SinSyn object:

SinAnal* input: input object object, of the SinAnal type, from which the tracks to be resynthesised will be read.
int maxtracks: max resynthesis tracks, should be <= tracks generated by the input object.
Table* table: table object containing a wavetable to be used by each oscillator in the resynthesis, typically a cosine wave.
float scale: amplitude scaling of output.
int vecsize: object vector size, also determines the synthesis hopsize between analysis frames (defaults to 256). This needs to be the same value of the analysis hopsize for proper resynthesis
float sr: sampling rate in Hz (defaults to 44100).

public methods

void SetTable(**Table*** table)
void SetMaxTracks(**int** maxtracks)
void SetScale(**float** scale)

These methods set the different object parameters, such as the resynthesis wavetable, maximum number of tracks and scaling factor. These can alternatively be set/connected using the messages listed above.

Examples

The following connections are a simple example of the use of SinSyn to resynthesise track data generated by SinAnal:

```
HarmTable table(4000, 1, 1, 0.75);    // cosine wave
HammingTable window(fftsize, 0.5); // hanning window

// input sound
SndWave input(infile,READ,1,16,0,0.f);
SndIn   insound(&input, 1, decimation);

// IFD analysis
IFGram ifgram(&window,&insound,1.f,fftsize);
// Sinusoidal analysis
SinAnal sinus(&ifgram,thresh,intracks);
// Sinusoidal resynthesis
SinSyn  synth(&sinus,outracks,&table,scale);

// output sound
SndWave output(outfile, OVERWRITE,1,16,0,0.f);
output.SetOutput(1, &synth);
```

This code takes an input sound, from a file and passes it through the analysis process and then the data is resynthesised. In order to implement processing, the programmer either needs to write a loop and call the reading/writing and processing methods, or use a SndThread object, passing these objects to it.

Class SndAiff

Description

The SndAiff class implements AIFF file input/output. It currently support PCM formats only.

Construction

SndAiff(**char*** name, **short** mode, **short** channels=1, **short** bits=16, **SndObj**** inputlist=0,
float spos= 0.f, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Details

construction

SndAiff(**char*** name, **short** mode, **short** channels=1, **short** bits=16, **SndObj**** inputlist=0,
float spos= 0.f, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

This method constructs an object of the SndAiff class. Construction parameters are:

char* name: input/output AIFF soundfile name.
short mode: file open mode. One of the four options: OVERWRITE, APPEND,INSERT or READ. The first three open the file for writing (output), the last one opens it for reading (input).
short channels: number of output channels.
short bits: number of bits per sample (sample size). Since this class implements input from a self-describing format, in the READ mode the sample precision will be obtained from the soundfile header (8, 16, 24 and 32-bit formats are supported).
SndObj** inputlist: array of pointers to locations of SndObj-derived objects, which will be patched to the output channels.
float spos: start position of the read/write pointer, in seconds from the beginning of the sound data.
int vecsize: vector size in samples. Size of the internal read/write buffer, defaults to DEF_VECSIZE, 256.
float sr: sampling rate in HZ. Defaults to DEF_SR, 44100.f. Since this class implements input from a self-describing format, in the READ mode the sampling rate will be obtained from the soundfile header.

Examples

SndAIff is a specialisation of the SndIO class to deal with AIFF-format files. As such it takes the signal format information from the soundfile header and it also writes a header with full details. The output header writing is only complete when an object is destroyed. If the program fails to kill the object (for instance when a dynamic type is not deleted or when the program has exited early or crashed), the header will not be properly updated.

SndAiff outfile(“output.aif”, OVERWRITE);

The following object can write to an aiff file. A simple raw-format to aiff conversion program using that object can be written like this:

```
SndFIO  infile(“input.raw”, READ);
SndIn     input(&infile, 1);
outfile.SetOutput(1, &input);

while(!infile.Eof()){

infile.Read();
```

```
SndIn.DoProcess();
SndAiff.DoProcess();

}
```

Class SndASIO

Description

The SndASIO class implements ASIO driver input/output, as an option for realtime IO on Windows systems where such drivers are present. The main operational difference with the SndRTIO class is that one, and only one, object is required for both input and output (full duplex operation), as opposed to separate objects for input and output.

This class is based on the Steinberg ASIO API, and as such cannot be compiled under cygwin and is not present in that version. It has been compiled, tested and is available for the MS-Visual C++ compiler (it is posssible that it can be compiled with Borland C++ as well).

It is expected that a number of synchronisation and timing features will be added to this class in the future. This method of sound IO is probably the most efficient and with the best latency.

Construction

SndASIO(**int** channels, **int** mode = SND_IO, **char*** driver = "ASIO Multimedia Driver", **int** numbuffs=4, **SndObj**** inputs = 0, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Utilities

void DriverList()
char* DriverName(**int** dev, **char*** name)

Details

construction

SndASIO(**int** channels, **int** mode = IO, **char*** driver = "ASIO Multimedia Driver", **SndObj**** inputs = 0, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

This method constructs an object of the SndASIO class. Construction parameters are:

int channels: number of requested audio channels (the number audio channelsthat are opened will actually depend on the device driver and audio card). Some audio cards will only work properly with a minimum of two channels, it is advisable to open the device in stereo by default.
int mode: either SND_IO, for simultaneous input and output, SND_INPUT, for input only, or SND_OUTPUT, for output only.
char * driver: string containing the name of the device to be open. It must be present in the system. Use the utility functions DriverList() and DriverNames to obtain information about drivers available in the system.
int numbuffs**:** number of audio buffers, at least two, but usually 4 (the default).
SndObj** inputs: array of [channels] pointers to input SndObj objects, one per channel.
int vecsize: vector size in samples. Size of the internal DSP buffer, defaults to DEF_VECSIZE, 256.
float sr: sampling rate in HZ. Defaults to DEF_SR, 44100.f.

Utilities

void DriverList()

lists all present drivers on the standard output.

char* DriverName(**int** dev, **char*** name)

retrieves the name of a driver with driver ID **dev**, placing it into the string **name**. This string

should contain 32 characters, at least. It returns the string contatining device name. A NULL pointer (0) is returned if the device ID is not valid

Examples

A SndASIO object can be used to read or write to an audio device:

```
SndASIO io(2);
```

The object behaves like any other SndIO object, calls to Read() and Write() are used to effect the IO operations. The output object is set like this:

```
io.SetOutput(1, &outobj);
```

The sampling rate, buffersizes and latencies are adjusted externally to the client. If these are changed on-the-fly the object will possibly need to be deleted and re-initialised.

Class SndBuffer

Description

This class implements buffer input/output. It reads/writes samples to a buffer, making it possible to send buffered signals from one thread or process to another. The Read() and Write() functions will block if the buffer is empty or full. This class can be used to create buffering objects when synchronisation of signals from multiple threads is necessary.

Construction

SndBuffer(**short** channels, **int** buffsize=DEF_VECSIZE, **SndObj**** inputlist=0,
int vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Details

construction

SndBuffer(**short** channels, **int** buffsize=DEF_VECSIZE, **SndObj**** inputlist=0,
int vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

This method constructs an object of the SndBuffer class. Construction parameters are:
short channels: number of input/output channels
int buffsize: size of memory buffer, set by default to the same size of the DSP vector, but does not necessarily need to be of that size (but it will help if it is an integer multiple of the vector size), in sample frames.
SndObj** inputlist: list of input SndObj-derived objects, one for each output channel.
int vecsize: vector size in samples. Size of the internal DSP buffer, defaults to DEF_VECSIZE, 256.
float sr: sampling rate in HZ. Defaults to DEF_SR, 44100.f.

Examples

Read() reads the buffer and outputs a new signal vector to the processing chain every time it is called. Should be used in conjunction with at least one SndIn object. Calls to Read() will block if no samples are currently present in the buffer. Successive calls to Write() will fill the buffer with new samples.

Write() writes a new signal vector to the buffer every time it is called. Calls to Write() will block if the buffer is full. Successive calls to Read() will empty the buffer.

SndBuffer behaves like any other SndIO object, except that it writes to memory, instead of a particular device. It can be used to buffer signals between different threads.

Class SndCoreAudio

Description

The SndCoreAudio class implements dedicated audio IO to CoreAudio drivers in Mac OSX. A single object of this class is used on an application as it opens the driver for both input and output. To access inputs, programs should issue calls to Read() and to send signals to the outputs, the Write() method is used. See also the description of the SndRTIO class.

Construction

SndCoreAudio(**int** channels=2,**int** buffframes=512, **int** buffnos=4, **float** norm=32767.f,
SndObj** inObjs=0, **AudioDeviceID** dev=DEV_DEFAULT,
int vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Details

construction

SndCoreAudio(**int** channels=2,**int** buffframes=512, **int** buffnos=4, **float** norm=32767.f,
SndObj** inObjs=0, **AudioDeviceID** dev=DEV_DEFAULT,
int vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

int channels: number of output channels.
int buffframes: size of the read/write buffer implemented to optimise the operation (in frames). This is independent from the object output vector.
int buffnos: number of software buffers.
float norm: scaling value which will be applied to the input (actually its reciprocal). Since CoreAudio takes signals in the –1.0 to + 1.0 range, this is used to normalise any signals that might possibly be in the 16,24 or 32-bit ranges. This is mostly used for portability reasons. If the input is in the right amplitude range, this can be set to 1. Defaults to 16-bit normalisation (32767).
SndObj** inputs: array of pointers to locations of SndObj-derived objects, which will be patched to the output channels.
AudioDeviceID dev: the CoreAudio device ID, usually an integer, zero being the first device. It defaults to default output device (set in system preferences).
int vecsize: vector size in samples. Size of the internal DSP buffer, relevant only to INPUT mode, defaults to DEF_VECSIZE, 256.
float sr: sampling rate in HZ. Defaults to DEF_SR, 44100.f.

Examples

A SndCoreAudio object can be used to read/write to an audio device:

```
SndCoreAudio  soundcard;
```

This opens the default device (set in System Preferences) for input and output of stereo signals. The object behaves like any other SndIO object, calls to Read() and Write() are used to effect the IO operations. The output SndObjs are set like this:

```
soundcard.SetOutput(1, &outleft_obj);
soundcard.SetOutput(2, &outright_obj);

while(processing_on) {
    soundcard.Read();
     (...)
    soundcard.Write();
}
```

Class SndFIO

Description

The SndFIO class implements binary file input/output. It can be used for headerless ("raw" format) soundfiles, or for reading/storing any information on binary files.

Construction

SndFIO(**char*** name, **short** mode, **short** channels=1, **short** bits=16, **SndObj**** inputlist=0,
float spos= 0.f, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Public Methods

file information and status:
short GetMode()
long GetDataFrames()
float GetPos()
short GetStatus()
int Eof()

setting reading position:
void SetPos(**float** pos)
void SetPos(**long** pos)

Details

construction

SndFIO(**char*** name, **short** mode, **short** channels=1, **short** bits=16, **SndObj**** inputlist=0,
float spos= 0.f, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

This method constructs an object of the SndFIO class. Construction parameters are:

char* name: input/output file name.
short mode: file open mode. One of the four options: OVERWRITE, APPEND,INSERT or READ. The first three open the file for writing (output), the last one opens it for reading (input).
short channels: number of output channels.
short bits: number of bits per sample (sample size: 8, 16, 24 and 32-bit formats are supported).
SndObj** inputlist: array of pointers to locations of SndObj-derived objects, which will be patched to the output channels.
float spos: start position of the read/write pointer, in seconds from the beginning of the file.
int vecsize: vector size in samples. Size of the internal read/write buffer, defaults to DEF_VECSIZE, 256. float sr: sampling rate in HZ. Defaults to DEF_SR, 44100.f.

public methods

short GetMode()
long GetDataFrames()
float GetPos()
short GetStatus()

These methods retrieve the various class attributtes, respectively: read/write mode, number of data frames to be read in the file, start position (in secs) of read/write operation and file status (WAITOPEN, SFOPEN or SFERROR).

int Eof()

This method is used to check whether a reading operation has reached the end of the file. It returns 1 if so, and 0 if not.

void SetPos(**float** pos)
void SetPos(**long** pos)

These methods set the read/write pointer position, in secs and in bytes from the beginning of the file(or from the beginning of sound data, after the header, etc.), respectively. They are especially useful for reading information from different positions along the file.

Examples

The SndFIO class serves two main purposes: to provide raw soundfile IO and to set the interface for all soundfile IO classes. Raw soundfile input and output is usually disliked as it has the associated problems of portability and intelligibility.

SndFIO input(“soundfile.raw”, READ);

creates an object named input, which will read from “soundfile.raw”. It will take the samples to be 16-bit long and the sample frame to contain 1 channel only. Data will be expected to be at the 44100 Hz SR. The samples are taken to be in native byte ordering.

SndFIO output(“outfile.raw”, OVERWRITE);

can be used then to write to a raw soundfile with the same attributes as above. The signal to be written to file is taken from a SndObj object, which is connected to the output as in:

output.SetOutput(1, &outobj);

The objects then are manipulated in a loop to provide the processing. Notice how **SndFIO::Eof()** can be used to test if we reached the end of the file:

```
while(!input.Eof()){
input.Read();
(...)
output.Write();
}
```

Class SndIn

Description

This object receives one channel from a SndIO-derived object and outputs it in the SndObj processing chain.

Construction

SndIn()
SndIn(**SndIO*** input, **short** channel=1, **int** vecsize=DEF_VECSIZE **float** sr=DEF_SR)

Public Methods

short SetInput(**SndObj*** input, **short** channel)

Messages

[set] **“channel”**
[connect] **“input”**

Details

construction

SndIn()
SndIn(**SndIO*** input, **short** channel=1, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

These methods construct an object of the SndIn class. Construction parameters are:

SndIO* input: pointer to the location of a SndIO-derived object, such as, a soundfile input object, a realtime input object, etc....
short channel: audio channel from which a monophonic stream will be read. Defaults to channel 1.
int vecsize: the output vectorsize, defaults to 256.
float sr: sampling rate, defaults to 44100.

public methods

short SetInput(**SndObj*** input, **short** channel)

This method can be used to set the input object SndIn will be reading from. The messages **“channel”** and **“input”** can be used to set a channel and an input to read from.

Examples

SndIn objects are utilities used to interface between SndIO and SndObj objects. Most SndObj classes cannot take a direct input from a file or a device, so SndIO classes are there to implement these actions. SndObj objects also cannot connect to SndIO objects, so the SndIn class is there to provide this connection:

```
SndWave input(“soundfile.wav”,READ);
SndIn   insound(&input, 1);
```

In the above example, we are reading the first channel of an input RIFF-Wave file. A processing loop would look like this:

```
while(processing_on){
input.Read();
```

```
insound.DoProcess();
output.Write();
}
```

In the case of multichannel input, we would use multiple SndIn objects:

```
SndRTIO  adc(2, SND_INPUT);
SndIn      left_channel(&adc, 1);
SndIn      right_channel(&adc, 2);
```

Each SndIn then would output the data from each respective input channel.

Class SndIO

Description

The SndIO class is the base class for all IO operations: file, RT audio, midi, text/console and memory. This class implements input/output to the stdio. It reads/writes samples as text (floats) to the standard IO, making possible the use of UNIX-like pipes and redirection.

Construction

SndIO (**short** channels=1, **short** bits=16,**SndObj**** inputlist=0, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Protected Members

SndObj** m_IOobjs
float* m_output
float m_sr
short m_channels
short m_bits
int m_vecsize
int m_vecpos
int m_error

Public Methods

float GetSr()
int GetVectorSize()
short GetChannels()
short GetSize()
float Output(int pos, int channel)
short SetOutput(short channel, SndObj* input)

virtual short Read()
virtual short Write()
virtual char* ErrorMessage()

Details

construction

This method constructs an object of the SndIO class. Construction parameters are:

short channels: number of input/output channels
short bits: precision in bits (not directly applicable to this class, since it outputs floats, whatever precision is specified). Defaults to 16 bits (shorts).
SndObj** inputlist: list of input SndObj /SndObj -derived objects, one for each output channel.
int vecsize: vector size in samples. Size of the internal DSP buffer, defaults to DEF_VECSIZE, 256.
float sr: sampling rate in HZ. Defaults to DEF_SR, 44100.f.

protected members

SndObj** m_IOobjs: input object list, one item per channel. Allocated by the SndIO constructor.
float* m_output: signal vector, used by input objects. The method Output() is used to access the samples in this buffer. It works in an analoguous way to SndObj-derived classes. Note that this is only relevant four signal input.
float m_sr: sampling rate in Hz. Set ONLY at construction.

short m_channels: number of channels.
short m_bits: precision in bits.
int m_vecsize: size of signal vector (in items).
int m_vecpos: counter to use when accessing vectors of other objects.
int m_error: error code
int m_samples: number of samples in the signal vector (vecsize*channels).

public methods
float GetSr()

Returns the current sampling rate in Hz of the object.

int GetVectorSize()

Returns the current output vector size of the object.

short GetSize()

Returns the size (in bits) of the input/output sample.

short GetChannels()

Returns the number of input/output channels.

short SetOutput(short channel, SndObj* input)

Sets the output channels of a SndIO object. Returns 1 if successfully patched. This method is very useful as it allows patching of outputs after construction, which is the usual way of doing things.

virtual short Read ()

Reads one buffer of input signal. Returns 1 if successful. This function is normally placed in a loop with other processing functions such as DoProcess().

virtual short Write ()

Writes one buffer of output signal. Returns 1 if successful. This function is normally placed in a loop with other processing functions such as DoProcess().

virtual char* ErrorMessage ()

This method retrieves an error string from a SndObj class object. It is used for simple debugging.

Examples

The SndIO class is the base class for all input/output in the library. As such, its function is rather like SndObj, in that it provides the common interface for all IO operations. Its functionality in terms of processing is small. It can input/output signals as text characters to the standard IO. Simple piping of data can be then established with a SndIO object.

SndIO stdio(1);

Creates a SndIO object that will can from the standard input and write to the standard output. This object will handle one channel of data. Multichannel data is expected to be interleaved, so a 2 channel object will separate every other data item to each IO channel. With a SndObj-derived SndIn, we can get the data into a SndObj chain:

```
SndIn  input(&stdio, 1);
```

The SndIn **input** object can be patched into any other SndObj object. Even back into **stdio**:

```
stdio.SetOutput(1, &input);
```

Such patching will send the input straight out of the program, through the standard output. it will work rather like ‘echo’. In order to effect the processing we have to write a loop (or use SndThread):

```
while(processing_on){

stdio.Read();
input.DoProcess();
stdio.Write();

}
```

This is a rather silly program, but it shows how to connect the SndIO objects into a SndObj chain. See the reference page on the class SndObj, for other ways of connecting the two types of objects together.

Class SndJackIO

Description

The SndJackIO class implements audio input and output to a Jack Connection Kit server. The server needs to be already running in the machine. If it is not, the object will fail to construct. Any number of objects can be instantiated, each one will provide a input, output (or both) port for each channel. For simple applications, a single instance of the class can be used for both input and output. The ports are always connected by default to the hardware ports of the Jack server. They of course can be disconnected (programmatically or manually). The Jack signal standard defines 0dB amplitude as 1.0. Signals are expected to lie within that range.

Construction

SndJackIO(**char*** name, **int** channels=2, **int** mode= SND_IO, **int** buffnos=4,
SndObj** inObjs=0, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Public Methods

int ConnectIn(int channel, char* port)
int ConnectOut(int channel, char* port)
int DisconnectIn(int channel, char* port)
int DisconnectOut(int channel, char* port)

**jack_client_t *****GetClient()
jack_port_t *GetinPort(int channel)
jack_port_t *GetOutPort(int channel)

Details

construction

SndJackIO(**char*** name, **int** channels=2, **int** mode= SND_IO, **int** buffnos=4,
SndObj** inObjs=0, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

char *name: string with the name of the client, which will be used to identify the ports in the server. Ports will be named *name*_[in/out]*channel*.
int channels: number of output channels.
int buffnos: number of software buffers.
SndObj** inputs: array of pointers to locations of SndObj-derived objects, which will be patched to the output channels.
int vecsize: vector size in samples. Size of the internal DSP buffer, relevant only to INPUT mode, defaults to DEF_VECSIZE, 256.
float sr: sampling rate in HZ. Defaults to DEF_SR, 44100.f.

public methods

int ConnectIn(int channel, char* port)
int ConnectOut(int channel, char* port)

These methods connect the object input or output ports to specific ports, referred to by their name strings. if the port cannot be connected the methods return 0.

**jack_client_t *****GetClient()

This method returns the client handle associated with this object. It can be used to call other Jack API functions not defined in this class.

jack_port_t *GetinPort(int channel)
jack_port_t *GetOutPort(int channel)

These methods retrieve the port IDs associated with specific input or output channels.

Examples

A SndJackIO object can be used to read/write to a Jack port running on a server:

```
SndJackIO  jack("sndobjapp");
```

This creates a Jack client on a server with two input ports and two output ports. The ports will be named "sndobjapp_in1", "sndobjapp_in2", "sndobjapp_out1" and "sndobjapp_out2".
The output SndObjs are set like this:

```
jack.SetOutput(1, &outleft_obj);
jack.SetOutput(2, &outright_obj);
```

Signals can be input and output to the Jack ports with calls to Read() and Write():

```
while(processing_on) {
     jack.Read();
      (...)
     jack.Write();
}
```

Class SndLoop

Description

This object samples and loops a portion of a signal, with user-defined crossfade and loop time. The playback pitch can also be controlled, by the pitch multiplier parameter. A signal starts to be sampled when the DoProcess() method is called for the first time, but it can be resampled at any time.

Construction

SndLoop()
SndLoop(**float** xfadetime, **float** looptime, **SndObj*** InObj, **float** pitch = 1.f,
int vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Public Methods

void SetXFade **float** xfadetime)
void SetPitch(**float** pitch)
void ReSample()

Messages

[set] **"crossfade"**
[set] **"pitch"**
[set] **"resample"**

Details

construction

SndLoop()
SndLoop(**float** xfadetime, **float** looptime, **SndObj*** InObj, **float** pitch = 1.f,
int vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

These methods construct an object of the SndLoop class. Construction parameters are:

float xfadetime: cross-fade time between the end and start points of the signal loop in secs.
float looptime: total loop time in secs.
SndObj* InObj: input object, pointer to the location of a SndObj-derived object. The output signal from this object is added to the offset value and then used as the index on the lookup process.
float pitch: pitch ratio affecting the playback rate of the loop. Values > 1 increase the original pitch and < 1 decrease it,
also affecting the playback speed.
int vecsize: vector size in samples. Size of the internal DSP buffer, defaults to DEF_VECSIZE, 256.
float sr: sampling rate in HZ. Defaults to DEF_SR, 44100.f.

public methods

void SetXFade **float** xfadetime)
void SetPitch(**float** pitch)
void ReSample()

These methods set the different parameters of a SndLoop object, as described above. These can also be set using the messages **"crossfade"**, **"pitch"** and **"resample"** (this one with any value, which is ignored).

Examples

The following example shows a SndLoop object which takes a signal from the input (file or device), and samples it for 5 seconds, playing it back in a loop afterwards:

```
SndIn inobj(&input, 1);
SndLoop sampler(0.05f, 5.f, &inobj);

(...)

while(processing_on){

input.Read();
inobj.DoProcess();
sampler.DoProcess();
output.Write();

}
```

Class SndMidiIn [/SndMidi]

Description

The SndMidiIn class implements MIDI input on selected platforms (OSS, Irix and Windows MME). This class is derived from the SndIO SndMidi class, which is never used directly. The SndMidi class provides the basic mechanisms for realtime MIDI IO.

Construction

All Platforms:
SndMidiIn()

OSS/Irix:
SndMidiIn(**char*** port, **int** buffsize=64)

Windows MME:
SndMidiIn(**int** port, **int** buffsize=64)

Public Methods

[implemented in SndMidi]
short NoteOn()
short NoteOff()
char LastNote()
char Velocity(**char** note)
char LastNoteVelocity()
char Aftertouch(**char** note)
char LastNoteAftertouch()
short GetMessage(**short** channel)

Details

construction

SndMidiIn()
SndMidiIn(int port, int buffsize=64)
SndMidiIn(char* port, int buffsize=64)

These methods construct an object of the SndMidiIn class. Construction parameters are:

char port: midi input device. Defaults to the first input device on all platforms: "Serial Port 2" on Irix and "/dev/midi/" on OSS.
int port: midi input device, defaults to 0 (first input device).
int buffsize: size of input buffer.

On Windows, the number of devices and their names can be retrieved using three utility functions provided by this library:

void **MidiDeviceList**(): lists all present devices on the standard output.
char* **MidiInputDeviceName**(int dev, char* name): retrieves the name of a MIDI device with device ID **dev**, placing it into the string **name**. This string should be of MAXPNAMELEN size, at least. Both functions return the device name. A NULL pointer (0) is returned if the device ID is not valid. The sample code below will list all input devices on the standard output:

```
char name[MAXPNAMELEN];
int j = 0;
```

```
while(MidiInputDeviceName(j, name)){
cout << name << "\n";
j++;
}
```

public methods

short**NoteOn**()
short **NoteOff**()

These methods can be used to check for NOTE messages. Whenever a new noteon/off message is received, these methods will return the note number. If no new NOTE message was received, they return -1. They will keep returning -1 until a new noteon/off message was received.

char **LastNote**()
char **LastNoteVelocity**()
char **Velocity**(char note)

These methods retrieve, respectively, the last note number received, its velocity and the last velocity value for a particular note number.

char **Aftertouch**(char note)
char **LastNoteAftertouch**()
These methods retrieve, respectively, the last aftertouch value for a particular note and the aftertouch value for the last note received.

short **GetMessage**(short channel)

This method retrieves a code reflecting the current MIDI channel message number received. The usual set codes are:

NOTE_MESSAGE, note on
PBEND_MESSAGE, pitchbend or controller 0
MOD_MESSAGE, modulation wheel (controller 1)
BREATH_MESSAGE, breath control (controller 2)
FOOT_MESSAGE, breath control (controller 4)
PORT_MESSAGE, portamento (controller 5)
VOL_MESSAGE, volume (controller 6)
BAL_MESSAGE, balance (controller 7)
PAN_MESSAGE , pan (controller 9)
EXPR_MESSAGE, expression (controller 10)
AFTOUCH_MESSAGE, monophonic aftertouch or channel pressure
POLYAFTOUCH_MESSAGE, polyphonic aftertouch
PROGRAM_MESSAGE, program change
VELOCITY_MESSAGE, velocity (from a note message)
NOTEOFF_MESSAGE, note off

float Output(**int** channel)

The SndIO::Output() method can be used to retrieve the current output value for a particular MIDI message on a particular MIDI channel. The MIDI byte number returned (as a float) will depend on the type of message (for instance, for NOTE messages, the value returned is MIDI byte 2, note number).

Examples

Read() Outputs the current MIDI input message, every time it is called. Used in conjuction with MidiIn or MidiIn-derived classes. Returns the number of messages read (on Windows this means 1, on SGI it can be > 1) and 0 if no new messages were received.

```
SndMidiIn input(1); // Port 1 (on Windows)
(...)

while(processing_on){

(...)
input.Read(); // reads MIDI messages from the interface
(...)

}
```

Class SndObj

Description

The SndObj class is the base for the processing classes in the Sound Object Library. It defines the basic elements that make up a Sound Processing Object and provides basic methods for manipulating them. This class provides very little processing, it merely copies its input signal to the output (signal is read from an input object and copied to the output buffer). Its importance is that it establishes the basic mechanisms on which the library is based. Nevertheless,SndObj objects can be useful for simple applications such as retrieving signals from vectors, scaling, offset, mixing, etc..

Construction

SndObj()
SndObj(**SndObj&** obj)
SndObj(**SndObj ***input, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Protected Member Variables

**float ***m_output
**SndObj ***m_input
float m_sr
int m_vecsize
int m_vecpos
int m_error
short m_enable
**msg_link ***m_msgtable

Protected Methods

void AddMsg(**char ***mess, **int** ID)
int FindMsg(**char ***mess)

Public Methods

processing bypass:
void Enable()
void Disable()

output signal access:
float Output**(int** pos**)**
int PopOut**(float ***vector**, int** size**)**
int AddOut(**float*** vec, **int** size)

signal input:
int PushIn(**float** *vector, **int** size)

parameter/state access:
bool IsProcessing()
int GetVectorSize()
int GetError();
float GetSr()
SndObj* GetInput()
void GetMsgList(**string*** list)

parameter/state setting:
virtual void SetSr(**float** sr)

void SetInput(**SndObj *****input)
void SetVectorSize(**int** vecsize)

message passing:
virtual int Set(**char *****mess, **float** value)
virtual int Connect(char *mess, void *input)

operators:
SndObj operator=(**SndObj** obj)
SndObj operator+(**SndObj&** obj)
SndObj operator-(**SndObj&** obj)
SndObj operator*(**SndObj&** obj)
SndObj& operator+=(**SndObj&** obj)
SndObj& operator-=(**SndObj&** obj)
SndObj& operator*=(**SndObj&** obj)
SndObj operator+(**float** val)
SndObj operator-(**float** val)
SndObj operator*(**float** val)
SndObj& operator+=(**float** val)
SndObj& operator-=(**float** val)
SndObj& operator*=(**float** val)
void operator>>(SndIO& obj)
void operator<<(SndIO& obj)
void operator<<(**float** val)
void operator<<(**float *****vector)

signal processing:
virtual short DoProcess()

error messages:
virtual char* ErrorMessage()

Messages

[Set] “SR”
[Set] “vector size”
[Connect] “input”

Details

construction

SndObj()
SndObj(**SndObj&** obj)
SndObj(**SndObj *****input, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

These methods construct an object of the SndObj class. The default constructor **SndObj()** creates a basic object with the default sampling rate DEF_SR (44100), default vector size DEF_VECSIZE (256) and no input signal object (NULL pointer). The copy constructor **SndObj(SndObj& obj)** creates a new object based on an existing one, with the same parameters. The full constructor takes the following arguments:

SndObj* input: an input signal object, whose output will be processed by this object.
int vecsize: object vector size. This is the size of the m_output signal buffer, which holds the output signal for this object. Defaults to DEF_VECSIZE (256).
float sr: the sampling rate for this object. Defaults to DEF_SR (44100.f).

protected member variables

These variables can only be accessed by methods of this class or other derived classes.

float *m_output: this is the output signal buffer, an array of *m_vecsize* floats.
SndObj *m_input: this holds the pointer to the input signal object.
float m_sr: the sampling rate.
int m_vecsize: the size of the output signal buffer *m_output*.
int m_vecpos: an index into the *m_output* array, which can be used by class methods to access individual signal samples. For instance, it is used by *operator<<(float val),* to keep track of vector position and *DoProcess()* to step through the array. If m_vecpos is not 0 or *m_vecsize*, it is likely that the object is currently processing audio at that moment.
int m_error: error code, which can be used by error checking routines, such as *ErrorMessage()*.
short m_enable: on/off switch variable. if 0, the processing method is bypassed and the object outputs 0s (silence).
msg_link *m_msgtable**:** a pointer to the last item of a linked list containing string messages and integer IDs, used by the message-passing methods **Set()** and **Connect()** to process messages to objects.

protected methods

These methods can only be invoked by this class or other derived classes.

void AddMsg(**char ***mess, **int** ID)

Adds a message string to the object message list, assigning an ID to the message. The null-terminated (C-type) string *mess* containins a message to be added to the list and the integer *ID* is an arbitrary integer which will be associated with the message.

int FindMsg(**char ***mess)

This method returns an ID associated with a message string *mess*, or 0 if it does not find the message in the message list.

public methods

void Enable()
void Disable()

These two methods are used to bypass (**Disable()**) or to switch on (**Enable()**) the processing. They act on the main perform method, **DoProcess()**. If the processing is disabled, all calculations are bypassed and the object will output silence (0s). The class is constructed with the processing enabled.

float Output**(int** pos**)**

This method can be used to access the object output samples directly. The argument ***pos*** is an integral index into the output vector, so it can be anything between 0 and the vector size.

int PopOut**(float ***vector**, int** size**)**

This method retrieves a vector of size size, from the object vector, continuoulsy. Returns an index to the next vector position immediately after the extracted block.

int AddOut(**float*** vec, **int** size)

Same as PopOut(), but adds to the output vector (accumulates), instead of replacing its samples.

int PushIn(**float** *vector, **int** size)

this method pushes a vector of samples (of size size) into the object vector. Returns an index pointing to the next 'insert' position in the vector.

bool IsProcessing()
int GetVectorSize()
int GetError();
float GetSr()
SndObj* GetInput()
void GetMsgList(**string*** list)

These methods retrieve the different parameters that make the object state. **GetVectorSize()** and **GetSr()** return the size of the output signal vector and the sampling rate respectively. **GetInput()** returns a pointer to the input signal object and **GetError()** retrieves an error code, or 0 if no error has occurred. **GetMsgList()** takes a pointer to an empty **string** object and fills it with all the string messages defined for this object, separated by newline characters ('\n'). **IsProcessing()** returns true if the object is processing audio at that instant, that is, **DoProcess()** is at work, or false if the object is idle.

virtual void SetSr(**float** sr)
void SetInput(**SndObj ***input)
void SetVectorSize(**int** vecsize)

These methods are used to set the parameters that make the object state. **SetSr()** and **SetVectorSize()** change the sampling rate and output vector size, respectively. They should be used carefully since they might affect the way the signal is processed. It is important to note that **SetVectorSize()** destroys the output array in the process of re-sizing it, so it should not be used during performance. **SetInput()** sets the input signal object. It is safe to use in most situations. **SetSr()** can be overriden, if the derived class state is dependent on the values of the sampling rate.

virtual int Set(**char ***mess, **float** value)
virtual int Connect(**char** *mess, **void** *input)

These two methods are used to process *Set* and *Connect* messages sent to the object. Please refer to list given above (under "Messages") for valid messages. Set() sets the parameter associated to the message to **value**, whereas **Connect()** connects the input associated with the message to the object pointed at by **input**. Both methods return 0 if the message was not understood.

SndObj operator=(**SndObj** obj)
SndObj operator+(**SndObj&** obj)
SndObj operator-(**SndObj&** obj)
SndObj operator*(**SndObj&** obj)
SndObj& operator+=(**SndObj&** obj)
SndObj& operator-=(**SndObj&** obj)
SndObj& operator*=(**SndObj&** obj)
SndObj operator+(**float** val)
SndObj operator-(**float** val)
SndObj operator*(**float** val)
SndObj& operator+=(**float** val)
SndObj& operator-=(**float** val)
SndObj& operator*=(**float** val)
void operator>>(**SndIO&** obj)
void operator<<(**SndIO&** obj)
void operator<<(**float** val)
void operator<<(**float ***vector)

These operators define simple operations on objects such as assignment, sum, difference, multiplication and shift (<<, >>). Objects can be added, subtracted and multiplied together. Especially useful are the operation-and-assigment (+=, -= and *=) operators, which work on the contents of the object output buffer, adding,subtracting or multiplying them by the contents the output buffer of another object (or by a floating-point scalar value). The shift operators provide a way of sending/retrieving a signal to/from a SndIO (sound input/output) object. A shift operator is also provided for filling the output buffer with the contents of a floating-point vector and for shifting in a floating-point scalar into the buffer. This last operator uses an internal counter to keep track of the current position and the buffer is treated as a circular buffer (the counter is incremented *modulo* vector size).

virtual short DoProcess()

This is the main processing method for the class, also known as the *perform* method.It produces a new vector full of samples every time it is invoked (if processing is enabled and the constructor did not accuse any errors). In the case of this class, the processing is limited to copying the input samples from an input object to the output. The method returns 1 if successful and 0 if not. Derived classes typically override this method, implementing it according to the way they process the audio signals. The only restriction to their implementation is that the class should produce an output vector of *m_vecsize* samples and should return 0 if unsuccesful. Other return values can be defined for special uses.

virtual char* ErrorMessage()

This method is a simple error-checking method which can be useful for debugging purposes. It returns a string message which relates to an error code issued by one of the class methods (usually the constructor), in case of an error. So it can be invoked if a class has found errors. Derived classes can override this method to add error messages if necessary.

Examples

A SndObj object is created using one of its constructors. SndObjs (and derived-type objects) can be created as static-memory objects:

SndObj sigobj;

or as dynamic-memory objects:

SndObj *sigobj = new SndObj;

The two examples above used the default constructor, but we will recommend the use of the full constructor whenever possible:

SndObj sigobj(&inobj);

Please note that the parameter **inobj** needs to be either of type SndObj or of the derived types. Because the constructor takes a pointer to an object, the address-of operator (&) is used. In case of pointers, the variable name will suffice (here we call it pinobj):

SndObj sigobj(pinobj);

Messages can be sent to an object to set one of its parameters:

sigobj.Set(“SR”, 48000.f);

or to connect an object into it:

```
sigobj.Connect("input", &inobj);
```

Once an object has been constructed and its parameters set, we can use it to process sound. The main *perform* method is used to produce a vector full of output samples. This is usually placed in a loop, as in:

```
while(processing_on) {

sigobj.DoProcess();

}
```

Here the object will continually produce signal vectors, which can be further processed or sent to an object of an output class. When using the class SndThread to manage the processing, there is no need to invoke the **DoProcess()** method (see the Class SndThread manual page for details).

The object output can be accessed by using the **Output()** method. This is the standard way for derived classes to get the input signal. A pointer to the input object is held by the m_input variable, so the following code will access the signal input (and copy it to the output) :

```
for(m_vecpos=0; m_vecpos< m_vecsize; m_vecpos++)
                        m_output[m_vecpos] = m_input->Output(m_vecpos);
```

There are also other ways of manipulating the object output, using the defined overloaded operators. For instance, the operators **+** and **=** can be used to mix objects:

```
while(processing_on) {

sigobj1.DoProcess;
sigobj2.DoProcess;
sigobj3.DoProcess;

sumobj = sigobj1 + sigobj2 + sigobj3;

}
```

A simple way of combing objects is found using the ***=**, **+=** and **-=** operators:

```
while(processing_on) {

sigobj1.DoProcess;
sigobj2 *= sigobj1;

}
```

The shift operators can be used to get and send signals:

```
float a[DEF_VECSIZE];
SndObj sigobj;
SndIO  outobj;

(…)

while(processing_on){

sigobj << a;
sigobj >> outobj;
```

}

The shift operator defined for a scalar can be used to shift in single values:

for(int n=0, float a=1.f; n < sigobj.GetVectorSize(); n++) sigobj << a*n;

Most operators will work on derived classes as well, except for **=**, **+**, **-** and ***** (but **+=** etc will do).

Class SndPVOCEX

Description

This class implements file IO to a PVOCEX-format file. This file format is designed for phase vocoder and spectral data IO.

Construction

SndPVOCEX(**char*** name, **short** mode = OVERWRITE,
int analformat=PVOC_AMP_FREQ, **int** windowtype=HANNING,
short channels=1, **int** channelmask=0, **short** bits=32,
int format=PCM, **SndObj**** inputlist=0,
float framepos= 0.f, **int** hopsize = DEF_VECSIZE,
int fftsize = DEF_FFTSIZE, **float** sr = DEF_SR)

Public Methods

int GetFFTSize()
int GetHopSize()
int GetWindowType()
int GetWindowLength()
void GetHeader(**WAVEFORMATPVOCEX*** pheader);
void SetTimePos(**float** pos);
bool IsPvocex()

Details

construction

SndPVOCEX(**char*** name, **short** mode = OVERWRITE,
int analformat=PVOC_AMP_FREQ, **int** windowtype=HANNING,
short channels=1, **int** channelmask=0, **short** bits=32,
int format=PCM, **SndObj**** inputlist=0,
float framepos= 0.f, **int** hopsize = DEF_VECSIZE,
int fftsize = DEF_FFTSIZE, **float** sr = DEF_SR)

char* name: input/output file name.
short mode: file open mode. One of the four options: OVERWRITE, APPEND,INSERT or READ. The first three open the file for writing (output), the last one opens it for reading (input).
int analformat: analysis format type, either PVOC_AMP_FREQ (amplitude and frequency) PVOC_AMP_PHASE (polar format) or PVOC_COMPLEX (rectangular)
int windowtype: analysis window shape, HANNING, HAMMING, KAISER, CUSTOM or RECTANGULAR.
short channels: number of output channels.
int channelmask: channel mask identifying the multichannel format used (for multichannel output). This is a series of descriptors for the channels used in the multichannel encoding, which are or'ed (|) together (any combination from this set of 16):

SPEAKER_FRONT_LEFT
SPEAKER_FRONT_RIGHT
SPEAKER_FRONT_CENTER
SPEAKER_LOW_FREQUENCY
SPEAKER_BACK_LEFT
SPEAKER_BACK_RIGHT
SPEAKER_FRONT_LEFT_OF_CENTER
SPEAKER_FRONT_RIGHT_OF_CENTER

SPEAKER_BACK_CENTER
SPEAKER_SIDE_LEFT
SPEAKER_SIDE_RIGHT
SPEAKER_TOP_CENTER
SPEAKER_TOP_FRONT_LEFT
SPEAKER_TOP_FRONT_CENTER
SPEAKER_TOP_FRONT_RIGHT
SPEAKER_TOP_BACK_LEFT
SPEAKER_TOP_BACK_CENTER
SPEAKER_TOP_BACK_RIGHT
SPEAKER_RESERVED

short bits: number of bits per sample (sample size). Since this class implements input from a self-describing format, in the READ mode the sample precision will be obtained from the soundfile header. 32 bits here actually mean float samples. 64 bits would set the output format to double precision.
int format: time-domain format, PCM is currently the only format supported.
SndObj** inputlist: array of pointers to locations of SndObj-derived objects, which will be patched to the output channels.
float framepos: start position of the read/write pointer, in seconds from the beginning of the sound data.
int fftsize: fft analysis size, also the window size (current implementation only supports windows of the same size as the spectral frame).
int hopsize: hopsize in samples between analysis windows.
float sr: sampling rate in HZ. Defaults to DEF_SR, 44100.f. Since this class implements input from a self-describing format, in the READ mode the sampling rate will be obtained from the soundfile header

public methods
int GetFFTSize()
int GetHopSize()
int GetWindowType()
int GetWindowLength()

These methods return information relating to the formatting of the spectral analysis data, fft size, hopsize, window type and length.

void GetHeader(**WAVEFORMATPVOCEX*** pheader);

This returns the spectral file header as a single structure:

```
struct pvoc_data {
 short wWordFormat;    /* IEEE_FLOAT or IEEE_DOUBLE */
 short wAnalFormat;    /*PVOC_AMP_FREQ, PVOC_AMP_PHASE, PVOC_COMPLEX */
 short wSourceFormat;    /* WAVE_FORMAT_PCM or WAVE_FORMAT_IEEE_FLOAT*/
 short wWindowType;    /* defines the standard analysis window used, or a  custom window */
 int nAnalysisBins;  /* number of analysis channels. */
 int dwWinlen;     /* analysis window length, in samples */
 int dwOverlap;     /* window overlap length in samples (decimation)  */
 int dwFrameAlign;    /* usually nAnalysisBins * 2 * sizeof(float) */
 float fAnalysisRate;            /* sample rate / Overlap */
 float fWindowParam;    /* parameter associated with some window types: default 0.0f unless
needed */
};

struct pvocex{
 int dwVersion;                          /* initial version is 1*/
 int dwDataSize;                        /*  sizeof PVOCDATA data block */
```

```
 pvoc_data data;                    /* 32 byte block */
};

struct WAVEFORMATPVOCEX {

wave_head waveformatex;
wav_ex     waveformat_ext;
pvocex    pvocformat_ext;

};
```

For a description of the wave_head and wav_ex structures, see the reference page for the SndWaveX class.

void SetTimePos(**float** pos);

This sets a position along the file for the read/write pointer, in seconds from the beginning of the file.

bool IsPvocex()

Checks if the file is of the PVOCEX format.

Class SndRead

Description

This class implement s direct file readout from either RIFF-Wave or AIFF-format files. File format is determined by the filename extension: ".wav" for Wave format and ".aif" for AIFF. SndRead objects can be used to play files at any speed.

Construction

SndRead();
SndRead(**char*** name, **float** pitch=1.f, **float** scale=1.f,
int vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Public Methods

SndObj* Outchannel(**int** channel)
void SetInput(**char*** name)
void SetScale(**float** scale)
void SetPitch(**float** pitch)

Messages

[set] **"pitch"**
[set] **"scale"**

Details

construction

SndRead();
SndRead(**char*** name, **float** pitch=1.f, **float** scale=1.f,
int vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

These methods construct an object of the SndRead class:

char* name: input filename. Formats accepted are RIFF-Wave and AIFF, which are determined by the filename extension (".wav") or (".aif").
float pitch: readout pitch (or speed).
float scale: amplitude scaling factor.
int vecsize: object vector size, defaults to 256.
float sr: sampling rate, defaults to 44100.

public methods

void SetInput(**char*** name)
void SetScale(**float** scale)
void SetPitch(**float** pitch)

These methods set the input file and object parameters. The latter can be also set with the messages listed above

SndObj* Outchannel(**int** channel)

For multichannel files, this method returns the SndObj pointer associated with a particular channel. The object pointer can then be used to obtain the individual output of each channel. For instance,

SndObj* channel2 = fileread.Outchannel(2);

output.SetOutput(2, channel2);

Examples

SndRead objects read directly from files (using internal SndIO-derived objects), as such they can easily manipulate the readout speed to provide transformations of pitch, etc.. File formats are defined by the filename extension:

SndRead aiffile(“sound.aif”, 1.25f);
SndRead wavefile(“sound.wav”, 0.75f);

In the examples above, the two SndRead objects are transposing the input files (a major third above and a perfect fifth below). The length of the soundfiles will of course be altered. If the file is multichannel, a SndRead object will hold the mono sum of all channels. Individual channels can be accessed as discussed above

```
while(processing_on){

aifile.DoProcess();
wavefile.DoProcess();
mixer.DoProcess();
output.Write();

}
```

Class SndRTIO

Description

The SndRTIO class implements realtime audio device input/output. It has been currently implemented on five platforms: OSS (Linux, etc.), ALSA (Linux), Irix (SGI), MacOSX (CoreAudio) and MS-Windows (Windows Multimedia Extensions audio).

Construction

OSS [-DOSS]:
SndRTIO(**short** channels, **int** mode, **int** buffsize = 512,
int DMAbuffsize = 512, **int** encoding = SHORTSAM_LE, **SndObj**** inputs = 0,
int vecsize = DEF_VECSIZE, **float** sr=DEF_SR, **char*** device = "/dev/dsp")

ALSA [-DALSA]:
SndRTIO(**short** channels, **int** mode, **int** buffsize = 512,
int buffno=4, **int** encoding = SHORTSAM_LE, **SndObj**** inputs = 0,
int vecsize = DEF_VECSIZE, **float** sr=DEF_SR, **char*** device = "plughw:0,0")

SGI [-DSGI]:
SndRTIO(**short** channels, **int** mode, **int** buffsize = 512,
int DACqueue = 512, **int** encoding = SHORTSAM, **SndObj**** inputs=0,
int vecsize = DEF_VECSIZE, **float** sr=DEF_SR, **int** dev=AL_DEFAULT_OUTPUT)

OSX [-DMACOSX]:
SndRTIO(**short** channels, **int** mode, **int** buffsize = 512,
int buffno = 10, int encoding = SHORTSAM, SndObj** inputs=0,
int vecsize = DEF_VECSIZE, **float** sr=DEF_SR,
AudioDeviceID dev = DEFAULT_DEV)

Windows MME (-DWIN):
SndRTIO(**short** channels, **int** mode, **int** buffsize = 128,
int buffno = 10, **int** encoding = SHORTSAM, **SndObj**** inputs=0,
int vecsize = DEF_VECSIZE, **float** sr=DEF_SR, **int** dev = WAVE_MAPPER)

Details

construction

SndRTIO(**short** channels, **int** mode, **int** buffsize = 512,
int DMAbuffsize = 512, **int** encoding = SHORTSAM_LE, **SndObj**** inputs = 0,
int vecsize = DEF_VECSIZE, **float** sr=DEF_SR, **char*** device = "/dev/dsp")
SndRTIO(**short** channels, **int** mode, **int** buffsize = 512,
int buffno=4, **int** encoding = SHORTSAM_LE, **SndObj**** inputs = 0,
int vecsize = DEF_VECSIZE, **float** sr=DEF_SR, **char*** device = "plughw:0,0")
SndRTIO(**short** channels, **int** mode, **int** buffsize = 512,
int DACqueue = 512, **int** encoding = SHORTSAM, **SndObj**** inputs=0,
int vecsize = DEF_VECSIZE, **float** sr=DEF_SR, **int** dev=AL_DEFAULT_OUTPUT)
SndRTIO(**short** channels, **int** mode, **int** buffsize = 512,
int buffno = 10, int encoding = SHORTSAM, SndObj** inputs=0,
int vecsize = DEF_VECSIZE, **float** sr=DEF_SR,
AudioDeviceID dev = DEFAULT_DEV)
SndRTIO(**short** channels, **int** mode, **int** buffsize = 128,
int buffno = 10, **int** encoding = SHORTSAM, **SndObj**** inputs=0,
int vecsize = DEF_VECSIZE, **float** sr=DEF_SR, **int** dev = WAVE_MAPPER)

These methods construct an object of the SndRTIO class on the different supported platforms. Common Construction parameters are:

short channels: number of output channels.
int encoding: sample encoding, one of the following: FLOATSAM (floats, only on Irix), BYTESAM (8-bit), SHORTSAM_LE (16-bit little-endian, also SHORTSAM on WIN/OSS), SHORTSAM_BE (16-bit big-endian, also SHORTSAM on Irix), LONGSAM (32-bit integers, only on Irix). Defaults to SHORTSAM.
int buffsize: size of the read/write buffer implemented to optimise the operation (in frames). This is independent from the object output vector.
SndObj** inputs: array of pointers to locations of SndObj-derived objects, which will be patched to the output channels.
int vecsize: vector size in samples. Size of the internal DSP buffer, relevant only to INPUT mode, defaults to DEF_VECSIZE, 256.
float sr: sampling rate in HZ. Defaults to DEF_SR, 44100.f.

Platform specific:

[OSS]
short mode: SND_INPUT or SND_OUTPUT.
int DMAbuffsize: size of the DMA buffer used in the operation.
char* device: input/output device used (normally one of the /dev/dsp or /dev/audio files listed in the /dev directory).

[ALSA]
short mode: SND_INPUT or SND_OUTPUT.
int buffno: number of hardware buffers (in alsa terminology, periods).
char* device: input/output device used (“plughw:0,0”)

[SGI]
short mode: SND_INPUT or SND_OUTPUT.
int DACqueue: size of the input/output device hardware queue.
int dev: input/output device, defaults to AL_DEFAULT_OUTPUT.

[OSX]
short mode: SND_INPUT, SND_OUTPUT or SND_IO. Some soundcards will not allow being opened for input and output separately, so a single object should be used for reading AND writing, using SND_IO. This is the case with builtin audio. Alternatively the SndCoreAudio class can be used.
int buffno: number of software buffers.
AudioDeviceID dev: the CoreAudio device ID, usually an integer, zero being the first device. It defaults to default output device (set in system preferences).

[WIN]
short mode: SND_INPUT or SND_OUTPUT.
int buffno: number of input/output buffers used.
int dev: input/output device ID, defaults to WAVE_MAPPER, the device set at the Windows multimedia control panel. The number of devices and their names can be retrieved using three utility functions provided by this library:

void **DeviceList**(): lists all present devices on the standard output.
char* **InputDeviceName**(int dev, char* name)
char* **OutputDeviceName**(int dev, char* name): retrieves the name of a device with device ID **dev**, placing it into the string **name**. This string should be of MAXPNAMELEN size, at least. Both functions return the device name. A NULL pointer (0) is returned if the device ID is not valid. The sample code below will list all input devices on the standard output:

```
char name[MAXPNAMELEN];
int j = 0;
while(InputDeviceName(j, name)){
cout << name << "\n";
j++;
}
```

Examples

A SndRTIO object can be used to read or write to an audio device:

```
SndRTIO input(1, SND_INPUT);
SndRTIO output(1, SND_OUTPUT);
```

The only exception is that on OSX, some CoreAudio devices do not like to be opened for reading and writing separately, so a single object is used:

```
SndRTIO  coreaudio(1, SND_IO);
```

The object behaves like any other SndIO object, calls to Read() and Write() are used to effect the IO operations. The output SndObj is set like this:

```
output.SetOutput(1, &outobj);
```

or

```
coreaudio.SetOutput(1, &outobj);
```

in the case of the coreaudio object shown above. The buffersizes and buffer numbers can be adjusted to avoid drop-outs, which might occur depending on the capabilities of a particular system. Larger values will of course affect the latency of the processing.

Class SndSinIO

Description

This class implements file IO to a SINUSEX-format file. This file format is designed for sinusoidal analysis track data.

Construction

SndSinIO(**char*** name, int maxtracks, float threshold=0.01f, **int** windowtype=HANNING,
short mode = OVERWRITE, **short** channels=1, **int** channelmask=0, **short** bits=32,
int format=PCM, **SndObj**** inputlist=0, **float** framepos= 0.f,
int hopsize = DEF_VECSIZE, **int** fftsize = DEF_FFTSIZE, **float** sr = DEF_SR)

Public Methods

int GetTrackID(**int** track, **int** channel)
int GetTracks(**int** channel)
int GetFFTSize()
int GetHopSize()
int GetWindowType()
int GetMaxTracks()
void GetHeader(**WAVEFORMATSINUSEX*** pheader)

void SetTimePos(**float** pos)

Details

construction

SndSinIO(**char*** name, **int** maxtracks, **float** threshold=0.01f, **int** windowtype=HANNING,
short mode = OVERWRITE, **short** channels=1, **int** channelmask=0, **short** bits=32,
int format=PCM, **SndObj**** inputlist=0, **float** framepos= 0.f,
int hopsize = DEF_VECSIZE, **int** fftsize = DEF_FFTSIZE, **float** sr = DEF_SR)

char* name: input/output file name.
int maxtracks: maximum number of tracks per frame.
float threshold: analysis threshold
int windowtype: analysis window shape, HANNING, HAMMING, KAISER, CUSTOM or RECTANGULAR.
short mode: file open mode. One of the four options: OVERWRITE, APPEND,INSERT or READ. The first three open the file for writing (output), the last one opens it for reading (input).
short channels: number of output channels.
int channelmask: channel mask identifying the multichannel format used (for multichannel output). This is a series of descriptors for the channels used in the multichannel encoding, which are or'ed (|) together (any combination from this set of 16):

SPEAKER_FRONT_LEFT
SPEAKER_FRONT_RIGHT
SPEAKER_FRONT_CENTER
SPEAKER_LOW_FREQUENCY
SPEAKER_BACK_LEFT
SPEAKER_BACK_RIGHT
SPEAKER_FRONT_LEFT_OF_CENTER
SPEAKER_FRONT_RIGHT_OF_CENTER
SPEAKER_BACK_CENTER
SPEAKER_SIDE_LEFT
SPEAKER_SIDE_RIGHT
SPEAKER_TOP_CENTER
SPEAKER_TOP_FRONT_LEFT
SPEAKER_TOP_FRONT_CENTER
SPEAKER_TOP_FRONT_RIGHT
SPEAKER_TOP_BACK_LEFT
SPEAKER_TOP_BACK_CENTER
SPEAKER_TOP_BACK_RIGHT
SPEAKER_RESERVED

short bits: number of bits per sample (sample size). Since this class implements input from a self-describing format, in the READ mode the sample precision will be obtained from the soundfile header. 32 bits here actually mean float samples. 64 bits would set the output format to double precision.
int format: time-domain format, PCM is currently the only format supported.
SndObj** inputlist: array of pointers to locations of SndObj-derived objects, which will be patched to the output channels.
float framepos: start position of the read/write pointer, in seconds from the beginning of the sound data.
int fftsize: fft analysis size, also the window size (current implementation only supports windows of the same size as the spectral frame).
int hopsize: hopsize in samples between analysis windows.

float sr: sampling rate in HZ. Defaults to DEF_SR, 44100.f. Since this class implements input from a self-describing format, in the READ mode the sampling rate will be obtained from the soundfile header

public methods

int GetTrackID(**int** track, **int** channel)
int GetTracks(**int** channel)
int GetMaxTracks()

These two methods retrieve two important sinuosoidal analysis parameters, a track ID, which is used to identify and match tracks between frames (see the reference on SinAnal, SinSyn and AdSyn) and the current number of tracks for a particular channel. The number of tracks can possibly vary from frame to frame as old tracks can die and new ones can be created. GetMaxTracks() retrieves the maximum number of tracks possible in each frame.

int GetFFTSize()
int GetHopSize()
int GetWindowType()
int GetWindowLength()

These methods return information relating to the formatting of the spectral analysis data, fft size, hopsize, window type and length.

void GetHeader(**WAVEFORMATSINUSEX*** pheader);

This returns the spectral file header as a single structure:

```
struct sinus_data {
 short wWordFormat;
 short wHopsize;
 short wWindowType;
 short wMaxtracks;
 int dwWindowSize;
 float fThreshold;
 float fAnalysisRate;
};



struct sinusex {

int dwVersion;
sinus_data data;

};

struct WAVEFORMATSINUSEX {

wave_head waveformatex;
wav_ex     waveformat_ext;
sinusex    sinusformat_ext;

};
```

For a description of the wave_head and wav_ex structures, see the reference page for the SndWaveX class.

void SetTimePos(**float** pos);

This sets a position along the file for the read/write pointer, in seconds from the beginning of the file.

Class SndTable

Description

The SndTable object builds a function table based on the output of a SndFIO-derived object. It can be used to store one channel of previously recorded sound samples on a table for synthesis/processing purposes. The sound samples are normalised as they are inserted on the table.

Construction

SndTable()
SndTable(**long** L, **SndFIO*** input, **short** channel =1)

Public Methods

void SetInput(**long** L, **SndFIO*** input, **short** channel =1)

Details

construction

SndTable()
SndTable(**long** L, **SndFIO*** input, **short** channel =1)

Constructs a SndTable object.

long L: table length.
SndFIO* input: input sound, pointer to the location of a SndFIO-derived object (soundfile input).
short channel: channel from which the sound samples will be extracted

public methods

void SetInput(**long** L, **SndFIO*** input, **short** channel =1)

This method sets the function table parameters. MakeTable() should be invoked after any parameter resetting.

Class SndThread

Description

The SndThread class implements a processing pthread-based thread for SndObj and SndIO - derived objects. The class implements three lists of objects which are used by the processing thread to generate audio signals. Two of these lists hold pointers to SndIO-derived objects (the SNDIO_IN, for input objects and the SNDIO_OUT, for output objects). The third list holds pointers to SndObj-derived objects. Methods for adding, inserting and deleting objects to these lists are provided. Processing can be turned on or off by the ProcOn() and ProcOff() methods, respectively.

Construction

SndThread()

Public Methods

int AddObj(**SndObj** *obj)
int AddObj(**SndIO** *obj, **int** iolist)
int Insert(**SndObj** *obj, **SndObj*** prev)
int DeleteObj(**SndObj** *obj)
int DeleteObj(**SndIO** *obj, **int** iolist)
int GetStatus()
int GetSndObjNo()
int GetInputNo()
int GetOutputNo()
int ProcOn()
int ProcOff()

Details

construction

SndThread()

Constructs an empty SndThread object.

public methods

int AddObj(**SndObj** *obj)

Adds a SndObj-derived object to the top of the processing list. Returns 1 if successful and 0 if not.

int AddObj(**SndIO** *obj, **int** iolist)

Adds a SndIO-derived object to the specified IO list (SNDIO_IN or SNDIO_OUT).

For example

sndthread.AddObj(&input, SNDIO_IN);

adds a SndIO-derived object input to the SNDIO_IN list.

int Insert(**SndObj** *obj, **SndObj*** prev)

Inserts a SndObj-derived object to a position in the list after the SndObj object pointed by **prev**, which will have been previously added to the list.

This is useful to insert objects in the processing list, so that the right order of processing is obtained. For instance, if obj1 receives input from obj2, then it should be placed in the listafter obj2. The following call will make sure that this is the case

sndthread.Insert(&obj1, &obj2);

provided, of course, that obj2 has already been added to the thread list. This method returns the position of the inserted object in the list, if the insertion was succesful, 0 otherwise.

int DeleteObj(**SndObj** *obj)
int DeleteObj(**SndIO** *obj, **int** iolist)

Delete objects from the respective lists. As in AddOBJ(...), the parameter **iolist** can be either SNDIO_IN or SNDIO_OUT, the input and the output lists, respectively.

int GetStatus()
int GetSndObjNo()
int GetInputNo()
int GetOutputNo()

These are various methods to get properties of a SndThread object. GetStatus() returns the processing status (ON or OFF). GetSndObjNo(), GetInputNo() and GetOutputNo() return the number of objects (actually pointers to objects) held in the processing, input and ouput lists, respectively

int ProcOn()
int ProcOff()

These methods are use to switch the processing thread on or off. They return the current processing status (ON or OFF). The ProcOn() will return an OFF status if it fails to start the processing thread.

Examples

The following example shows the use of a SndThread object to provide a processing thread for a program:

```
char command[10];
int status;

// SndObj objects set-up
HarmTable table1(1024,1,SINE);
Oscilt    mod(&table1, 2.5f, 10000.f);
Oscili    oscil(&table1, 440.f, 10000.f, 0, &mod);
SndRTIO    output(1, OUTPUT);
output.SetOutput(1, &oscil);
```

The above shows a typical SndObj-SndIO chain. This chain can be set to process sound with a user-coded processing loop. Or alternatively as this example demonstrates, with SndThread:

```
// sound thread set-up
SndThread thread;

thread.AddObj(&mod);         // adds mod to the thread
thread.Insert(&oscil, &mod); // inserts oscil after oscil2
```

```
 thread.AddObj(&output, SNDIO_OUT); // adds output to the out list
```

Now we can start processing audio. This is controlled by program user, who can start. interrupt, re-start processing or exit the program at any time:

```
 // command loop
 while(1){

  cout << "Type a command: on, off or end\n";
  cin >> command;

  if(!strcmp(command, "on"))
    status = thread.ProcOn();// processing ON

  if(!strcmp(command, "off"))
    status = thread.ProcOff();

  if(!strcmp(command, "end")){
    if(status){ // if processing still ON
    status = thread.ProcOff();
    sleep(1);
     }
    break;
  }
 }
```

Class SndWave

Description

The SndWave class implements RIFF-Wave file input/output.

Construction

SndWave(**char*** name, **short** mode, **short** channels=1, sh**o**rt bits=16, **SndObj**** inputlist=0, **float** spos= 0.f, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Details

construction

SndWave(**char*** name, **short** mode, **short** channels=1, sh**o**rt bits=16, **SndObj**** inputlist=0, **float** spos= 0.f, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

This method constructs an object of the SndWave class. Construction parameters are:

char* name: input/output Wave soundfile name.
short mode: file open mode. One of the four options: OVERWRITE, APPEND,INSERT or READ. The first three open the file for writing (output), the last one opens it for reading (input).
short channels: number of output channels.
short bits: number of bits per sample (sample size). Since this class implements input from a self-describing format, in the READ mode the sample precision will be obtained from the soundfile header (8, 16, 24 and 32-bit formats are supported).
SndObj** inputlist: array of pointers to locations of SndObj-derived objects, which will be patched to the output channels.
float spos: start position of the read/write pointer, in seconds from the beginning of the sound data.
int vecsize: vector size in samples. Size of the internal read/write buffer, defaults to DEF_VECSIZE, 256.
float sr: sampling rate in HZ. Defaults to DEF_SR, 44100.f. Since this class implements input from a self-describing format, in the READ mode the sampling rate will be obtained from the soundfile header.

Examples

SndWave is a specialisation of the SndIO class to deal with WAVE-format files. As such it takes the signal format information from the soundfile header and it also writes a header with full details. The output header writing is only complete when an object is destroyed. If the program fails to kill the object (for instance when a dynamic type is not deleted or when the program has exited early or crashed), the header will not be properly updated.

SndWave outfile(“output.wav”, OVERWRITE);

The following object can write to a Wave file. A simple raw-format to Wave conversion program using that object can be written like this:

```
SndFIO  infile(“input.raw”, READ);
SndIn     input(&infile, 1);
outfile.SetOutput(1, &input);

while(!infile.Eof()){

infile.Read();
SndIn.DoProcess();
```

```
SndWave.DoProcess();
}
```

Class SndWaveX

Description

The SndWaveX class implements RIFF-Wave Extensible format (WAVEFORMATEX) file input/output.

Construction

SndWaveX(**char*** name, **short** mode, **short** channels=1, **int** channelmask=0, **short** bits=16, **SndObj**** inputlist=0, **float** spos= 0.f, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Public Methods

void GetHeader(**WAVEFORMATEXTENSIBLE*** pheader)
int GetChannelMask()
bool IsWaveExtensible()

Details

construction

SndWaveX(**char*** name, **short** mode, **short** channels=1, **int** channelmask=0, **short** bits=16, **SndObj**** inputlist=0, **float** spos= 0.f, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

This method constructs an object of the SndWaveX class. Construction parameters are:

char* name: input/output Wave soundfile name.
short mode: file open mode. One of the four options: OVERWRITE, APPEND,INSERT or READ. The first three open the file for writing (output), the last one opens it for reading (input).
short channels: number of output channels.
int channelmask: channel mask identifying the multichannel format used (for multichannel output). This is a series of descriptors for the channels used in the multichannel encoding, which are or'ed (|) together (any combination from this set of 16):

SPEAKER_FRONT_LEFT
SPEAKER_FRONT_RIGHT
SPEAKER_FRONT_CENTER
SPEAKER_LOW_FREQUENCY
SPEAKER_BACK_LEFT
SPEAKER_BACK_RIGHT
SPEAKER_FRONT_LEFT_OF_CENTER
SPEAKER_FRONT_RIGHT_OF_CENTER
SPEAKER_BACK_CENTER
SPEAKER_SIDE_LEFT
SPEAKER_SIDE_RIGHT
SPEAKER_TOP_CENTER
SPEAKER_TOP_FRONT_LEFT
SPEAKER_TOP_FRONT_CENTER
SPEAKER_TOP_FRONT_RIGHT
SPEAKER_TOP_BACK_LEFT
SPEAKER_TOP_BACK_CENTER
SPEAKER_TOP_BACK_RIGHT
SPEAKER_RESERVED

short bits: number of bits per sample (sample size). Since this class implements input from a self-describing format, in the READ mode the sample precision will be obtained from the soundfile header (8, 16, 24 and 32-bit integer formats are supported, plus 64-bit floating-

point).
SndObj** inputlist: array of pointers to locations of SndObj-derived objects, which will be patched to the output channels.
float spos: start position of the read/write pointer, in seconds from the beginning of the sound data.
int vecsize: vector size in samples. Size of the internal read/write buffer, defaults to DEF_VECSIZE, 256.
float sr: sampling rate in HZ. Defaults to DEF_SR, 44100.f. Since this class implements input from a self-describing format, in the READ mode the sampling rate will be obtained from the soundfile header.

public methods

void GetHeader(**WAVEFORMATEXTENSIBLE*** pheader)

This method retrieves the file format header fields in one single structure. Here's the description:

```
struct wave_head{
  long        magic;
  long        len0;
  long        magic1
  long        magic2
  long        len;
  short       format;
  short       nchns;
  long        rate;
  long        aver;
  short       nBlockAlign;
  short       size;
              };

struct wav_ex {
 short wValidBitsPerSample;
 int dwChannelMask;
 GUID SubFormat;
};

struct WAVEFORMATEXTENSIBLE {
wave_head waveformatex;
wav_ex    waveformat_ext;
};
```

int GetChannelMask()

This method retrieves the channel mask for multichannel files. It is used to describe the format encoding of the different channels (see above).

bool IsWaveExtensible()

Checks if the file header describes a Waveformatex file.

Examples

SndWaveX is a specialisation of the SndWave class to deal with WAVE-formatex files. As such it takes the signal format information from the soundfile header and it also writes a header with full details. The output header writing is only complete when an object is destroyed. If the program fails to kill the object (for instance when a dynamic type is not

deleted or when the program has exited early or crashed), the header will not be properly updated.

```
SndWaveX  outfile(“output.wav”, OVERWRITE);
```

The following object can write to a Wave file. A simple raw-format to Wave conversion program using that object can be written like this:

```
SndFIO  infile(“input.raw”, READ);
SndIn     input(&infile, 1);
outfile.SetOutput(1, &input);

while(!infile.Eof()){

infile.Read();
SndIn.DoProcess();
SndWaveX.DoProcess();

}
```

Class SpecCart

Description

SpecCart converts input spectral frames in the polar format into the cartesian (or rectangular) format. This is often done before the overlap-add IFFT operation (see IFFT class). The polar format understood by SpecCart is the one obtained with SpecPolar: each frame consists of magnitude and phase pairs for each FFT channel (or bin), except for the first two pairs, which will contain the magnitude of the 0Hz and Nyquist components.

Construction

SpecCart();
SpecCart(**SndObj*** input, **int** vecsize=DEF_FFTSIZE, **float** sr=DEF_SR)

Details

construction

SpecCart();
SpecCart(**SndObj*** input, **int** vecsize=DEF_FFTSIZE, **float** sr=DEF_SR)

This methods construct an object of the SpecCart class:

SndObj* input: a spectral object containing an output in the polar format, as described above.
int vecsize: vector size, the size of the FFT frame. Defaults to 1024.
float sr: sampling rate in Hz, defaults to 44100.

Examples

SpecCart objects are often used to obtain the rectangular (real, imag) format of an input signal. This trivial example shows the conenctions for a signal to be converted into a polar format and then converted back into its original form using a SpecCart object:

```
FFT   spec(&window, &inobj);
SpecPolar magphi(&spec);
SpecCart   rectang(&magphi);
IFFT  waveform(&window, &rectang);
```

In real-world examples, we would manipulate the polar format signal in some way before converting it back. The polar form of spectra is often more musically meaningful than its rectangular version.

Class SpecCombine

Description

SpecCombine objects can combine two inputs to make up a spectrum. It takes an amplitude input object, which will have a vector size of fftsize/2+1, containing the amplitudes for all spectral channels, from 0Hz up to the Nyquist (inclusive). This is then combined with a phase input object, also with a vector size of fftsize/2+1, containing phases for all channels, from 0 to the Nyquist. The phase of the 0Hz and Nyquist channels, the first and last sample in the phase input object vector, is always taken to be 0, thus ignored. These two real functions are combined and transformed into rectangular format frames (in the same format produced by the FFT class), which can be converted to the time-domain using an IFFT object.

Construction

SpecCombine()
SpecCombine(**SndObj*** magin, **SndObj*** phasin,
int vecsize=DEF_FFTSIZE, **float** sr=DEF_SR)

Public Methods

void SetPhaseInput(**SndObj*** phasin)
void SetMagInput(**SndObj*** magin)

Messages

[connect] **“phase input”**
[connect] **“magnitude input”**

Details

construction

SpecCombine()
SpecCombine(**SndObj*** magin, **SndObj*** phasin,
int vecsize=DEF_FFTSIZE, **float** sr=DEF_SR)

These methods construct a SpecCombine object:

SndObj* magin: magnitude input object, a SndObj-derived object with a vector size of fftsize/2+1 samples, containing the magnitude spectra of a signal frame-by-frame.
SndObj* phasin: phase input object, similarly with a vector size of fftsize/2+1, which generates the phase spectral frames.
int vecsize: the object vector size, determining the fftsize of its spectral output.

public methods

void SetPhaseInput(**SndObj*** phasin)
void SetMagInput(**SndObj*** magin)

These two methods connect the two inputs of the SpecCombine class. The messages listed above can be used for the same purpose.

Examples

SpecCombine combines two arbitrary functions, for magnitudes and phases of a signal. It can be used to create completely new spectra, but it also can be used to reverse the operation of a SpecSplit object:

```
FFT          spec(&window, &inobj)
SpecSplit    split(&spec);
SpecCombine  combine(split.magnitude,split.phase);
IFFT         waveform(&window, &combine);
```

In a more useful scenario, the spectrum split by SpecSplit would be transformed by further processing, before being recombined.

Class SpecEnvTable

Description

The SpecEnvTable object builds a frequency response function table based on a spectral magnitude envelope and a linear phase response. The envelope is defined by a starting point and two arrays: (1) segment legnths and (2) end points of each segment. The segments can be either exponential or linear. The table will contain a spectrum consisting of complex pairs for each positive DFT point, except for the 0Hz and Nyquist points, which are purely real. Table sizes also determine the DFT size used and generally are set to a power-of-two value. The spectal table data format is the same employed by the SndObj spectral classes: 0Hz and Nyquist points, followed by all other spectral points from 1 to N/2 –1. The table values are not normalised. As a linear phase frequency response, it implies a total delay in samples of (N-1)/2 (N is the fftsize).

Construction

SpecEnvTable()
SpecEnvTable(**long** L**, int** segments**, float** start**,**
float* points**, float*** lengths**,float** type = 0.f**, float** nyquistamp=0.f)

Details

construction

SpecEnvTable()
SpecEnvTable(**long** L**, int** segments**, float** start**,**
float* points**, float*** lengths**,float** type = 0.f**, float** nyquistamp=0.f)

Constructs a SpecEnvTable object.

long L: table length.
int segments: number of envelope segments.
float start: starting value of envelope (0 Hz magnitude).
float* points: an array of floats, containing the end values of each segment. Must match the above number of segments.
float* lengths: an array of floats, containing the lengths of each segment. Must match the above number of segments. Segment lengths are normalised to the table size (added up and then each one is divided by that total and multiplied by the table size).
float type: type of curve. Linear = 0, inverse exponential < 0 < exponential.
float nyquistamp: Nyquist magnitude.

Class SpecIn

Description

This object receives one channel from a spectral SndFIO-derived object and outputs it in the SndObj spectral processing chain. The input is usually a PVOCEX object.

Construction

SpecIn()
SpecIn(**SndFIO*** input, **short** channel=1)

Public Methods

short SetInput(**SndObj*** input, **short** channel, **int** vecsize=DEF_FFTSIZE, **float** sr=DEF_SR)

Messages

[set] **“channel”**
[connect] **“input”**

Details

construction

SpecIn()
SpecIn(**SndFIO*** input, **short** channel=1 **int** vecsize=DEF_FFTSIZE, **float** sr=DEF_SR)

These methods construct an object of the SpecIn class. Construction parameters are:

SndFIO* input: pointer to the location of a SndFIO-derived (file input) object, generally a PVOCEX object.
short channel: audio channel from which a monophonic stream will be read. Defaults to channel 1.
int vecsize: the output vectorsize, defining the fft frame size, defaults to 1024.
float sr: sampling rate, defaults to 44100.

public methods

short SetInput(**SndObj*** input, **short** channel)

This method can be used to set the input object SpecIn will be reading from. The messages **“channel”** and **“input”** can be used to set a channel and an input to read from.

Examples

SpecIn objects are utilities used to interface between spectral SndFIO and SndObj objects. Most SndObj classes cannot take a direct input from a file or a device, so SndFIO classes are there to implement these actions. SndObj objects also cannot connect to SndFIO objects, so the SpecIn class is there to provide this connection:

```
SndPVOCEX input(“specfile.pvx”,READ);
SpecIn   inspec(&input, 1);
PVS      synth(&window, &inspec);
```

In the above example, we are reading the first channel of an input RIFF-Wave file. A processing loop would look like this:

```
while(processing_on){
```

```
input.Read();
inspec.DoProcess();
synth.DoProcess();
output.Write();

}
```

In the case of multichannel input, we would use multiple SpecIn objects:

```
PVOCEX  input(“stereospec.pvx”, READ);
SpecIn      left_channel_spec(&input, 1);
SpecIn    right_channel_spec(&input, 2);
```

Each SpecIn then would output the data from each respective input channel.

Class SpecInterp

Description

This class performs simple interpolation of two spectra, according to an interpolation value, set between 0 and 1. The class does not make any assumption about the format of the input, as it interpolates each element of the input objects output frames using the same interpolation values. As such, the input can be in rectangular, polar or PV format, but generally the most effective results will be with PV inputs.

Construction

SpecInterp()
SpecInterp(**float** i_offset, **SndObj*** input1, **SndObj*** input2, **SndObj*** interpobj=0,
int vecsize=DEF_FFTSIZE, **float** sr=DEF_SR)

Public Methods

void SetInterp(**float** i_offset, **SndObj*** interpobj=0)

Messages

[set, connect] **"interpolation"**

Details

construction

SpecInterp()
SpecInterp(**float** i_offset, **SndObj*** input1, **SndObj*** input2, **SndObj*** interpobj=0,
int vecsize=DEF_FFTSIZE, **float** sr=DEF_SR)

These methods construct and object of the SpecInterp type:

float i_offset: interpolation amount offset value, which is added to the interpolation input object signal. Combined interpolation amounts above 1 or below 0 are clipped to those values. 0 takes the signal from input1 and 1 takes the signal from input2, values in between interpolate.
SndObj* input1, input2: input spectral objects.
SndObj* interpobj: input interpolation amount object, generating a time-varying signal that controls the interpolation amount.
int vecsize: object vectorsize, equivalent to the FFT frame size, defaults to 1024.
float sr: sampling rate, in Hz (defaults to 44100).

public methods

void SetInterp(**float** i_offset, **SndObj*** interpobj=0)

Sets the interpolation amount and/or connects to the interpolation amount control object. The message **"interpolation"** can also be used for the same purpose.

Examples

The SpecInterp provides a simple interpolation operation for spectra:

```
PVA       spec1(&window, &inobj1);
PVA       spec2(&window, &inobj2);
SpecInterp interp(0.5f, &spec1, &spec2);
PVS       synth(&window, &interp);
```

The example above interpolates the two spectra. In this case, amplitudes and frequencies are interpolated by the same amount. For individual interpolation of each parameter, see PVMorph.

Class SpecMult

Description

This class implements a complex multiplication of two spectra in rectangular format (with the real parts of 0Hz and Nyquist forming the first pair in the frame). This is almost equivalent to the time-domain convolution of two sounds, except that, because of the nature of the STFT analysis, the 'tail' of the convolution is discarded. SpecMult objects can be used for filtering and cross-synthesis purposes. The second input spectrum can be a time-varying signal from a SndObj or a single-frame spectrum stored in a Table-derived object (of fftsize length containing a rectangular spectral frame in a real fft format).

Construction

SpecMult()
SpecMult(**SndObj*** input1, **SndObj*** input2, **int** vecsize=DEF_FFTSIZE,
float sr=DEF_SR)
SpecMult(**Table*** spectab, **SndObj*** input1, **int** vecsize=DEF_FFTSIZE,
float sr=DEF_SR)

Public Methods

void SetInput2(**SndObj*** input2)
void SetTable(**Table** *spectab)

Messages

[connect] **"input 2"**
[connect] **"table"**

Details

construction

SpecMult()
SpecMult(**SndObj*** input1, **SndObj*** input2, **int** vecsize=DEF_FFTSIZE,
float sr=DEF_SR)
SpecMult(**Table*** spectab, **SndObj*** input1, **int** vecsize=DEF_FFTSIZE,
float sr=DEF_SR)

These methods construct and object of the SpecMult type:

SndObj* input1, input2: input spectral objects whose output is in rectangular format.
Table* spectab: Table-derived object containing a single real fft frame in rectangular format (fftsize/2 re and im pairs, with the first pair being re[0Hz] and re[SR/2]).
int vecsize: object vectorsize, equivalent to the FFT frame size, defaults to 1024.
float sr: sampling rate, in Hz (defaults to 44100).

public methods

void SetInput2(**SndObj*** input2)

This is used to set the second input signal object, with the same effect as the connect message **"input 2"**. When invoked, this sets the second input to be read from a SndObj instead of a Table object.

void SetTable(**Table** *spectab)

This is used to set the input table object, with the same effect as the connect message **"table"**. When invoked, this sets the second input to be read from a Table object instead of a SndObj.

Examples

Two spectra can be 'crossed' with each other by multiplication. This emphasizes the common components and eliminates the ones that differ:

```
FFT       spec1(&window, &inobj1);
FFT       spec2(&window, &inobj2);
SpecMult cross(&spec1, &spec2);
IFFT       waveform(&window, &cross);
```

Class SpecPolar

Description

SpecPolar objects convert rectangular spectra in the FFT format (see class FFT) to the polar format, consisting of magnitude and phase pairs. The only exception is the first pair of values in the FFT frame which contains the 0Hz and Nyquist magnitudes, respectively. SpecPolar output can be converted back into rectangular (or cartesian) format by SpecCart objects.

Construction

SpecPolar()
SpecPolar(**SndObj*** input,**int** vecsize=DEF_FFTSIZE, **float** sr=DEF_SR)

Details

construction

SpecPolar()
SpecPolar(**SndObj*** input,**int** vecsize=DEF_FFTSIZE, **float** sr=DEF_SR)

These methods construct and object of the SpecPolar type:

SndObj* input1: input spectral object, generating output in rectangular format.
int vecsize: object vectorsize, equivalent to the FFT frame size, defaults to 1024.
float sr: sampling rate, in Hz (defaults to 44100).

Examples

SpecPolar objects are often used to obtain the polar (magnitude, phase) format of an input signal. This trivial example shows the conenctions for a signal to be converted into a polar format using SpecPolar and then converted back into its original form:

```
FFT   spec(&window, &inobj);
SpecPolar magphi(&spec);
SpecCart   rectang(&magphi);
IFFT  waveform(&window, &rectang);
```

In real-world examples, we would manipulate the polar format signal in some way before converting it back. The polar form of spectra is often more musically meaningful than its rectangular version.

Class SpecSplit

Description

This class splits a spectral input singal (in rectangular format) into two outputs, consisting of magnitudes and phases of the signal. These outputs are presented separately, through member SndObj objects and together in the output vector. The output vector has to be fftsize+2 samples long, and it contains the amplitudes of all FFT channels, from 0 to the Nyquist (inclusive) followed by the phases, ordered similarly. The phases for channel 0 (0 Hz)and channel fftsize/2 (Nyquist) are always 0.

Construction

SpecSplit()
SpecSplit(**SndObj*** input,**int** vecsize=DEF_FFTSIZE+2, **float** sr=DEF_SR)

Public Members

SndObj* magnitude
SndObj* phase

Details

construction

SpecSplit()
SpecSplit(**SndObj*** input,**int** vecsize=DEF_FFTSIZE+2, **float** sr=DEF_SR)

SndObj* input1: input spectral object, generating output in rectangular format.
int vecsize: object vectorsize, equivalent to the FFT frame size, defaults to 1026 (DEF_FFTSIZE+2).
float sr: sampling rate, in Hz (defaults to 44100).

public members

SndObj* magnitude
SndObj* phase

These two pointers to SndObj objects provide acess to the magnitudes and phases, individually. These objects have output vectors that are fftsize/2+1samples long (vecsize/2). Any object taking them as input should have vectors that match that size.

Examples

SpecSplit extracts the magnitude and phase of an input spectrum:

```
FFT          spec(&window, &inobj)
SpecSplit    split(&spec);
SpecCombine  combine(split.magnitude,split.phase);
IFFT         waveform(&window, &combine);
```

In a more useful scenario, the spectrum split by SpecSplit would be transformed by further processing, before being recombined.

Class SpecThresh

Description

The class SpecThresh implements a simple thresholding process, which eliminates all components of the spectrum below a certain threshold (0-1). It takes an input in rectangular format (re, im: as produced by the FFT class, for instance) and outputs a thinner spectrum, in the same format. For each input frame, the maximum amplitude is found and then the components whose amplitude is below the threshold*max are eliminated

Construction

SpecThresh()
SpecThresh(**float** threshold, **SndObj*** input, **int** vecsize=DEF_FFTSIZE,
float sr=DEF_SR)

Public Methods

void SetThreshold(**float** thresh)

Messages

[set] **"threshold"**

Details

construction

SpecThresh()
SpecThresh(**float** threshold, **SndObj*** input, **int** vecsize=DEF_FFTSIZE,
float sr=DEF_SR)

These methods construct and object of the SpecInterp type:

float thresh: threshold, between 0 and 1, components with amplitude below the threshold are eliminated.
SndObj* input1: input spectral object, generating output in rectangular format.
int vecsize: object vectorsize, equivalent to the FFT frame size, defaults to 1024.
float sr: sampling rate, in Hz (defaults to 44100).

public methods

void SetThreshold(**float** thresh)

This sets the threshold value, the same as sending the message **"threshold"** to Set().

Examples

The following example shows the connections for a process that would eliminate all components that are below 1% of the maximum amplitude of each frame:

```
FFT   spec(&window, &inobj);
SpecPolar thresh(0.01, &spec);
IFFT  waveform(&window, &thresh);
```

Class SpecVoc

Description

SpecVoc objects take two rectangular spectral inputs and ouput a spectral signal in rectangular format, consisting of the magnitudes of the first spectral input and the phases of the second. This is moreor less the spectral equivalent of the time-domain channel vocoder.

Construction

SpecVoc();
SpecVoc(**SndObj*** input, **SndObj*** input2, **int** vecsize=DEF_FFTSIZE,
float sr=DEF_SR)

Details

construction

SpecVoc();
SpecVoc(**SndObj*** input, **SndObj*** input2, **int** vecsize=DEF_FFTSIZE,
float sr=DEF_SR)

These methods construct and object of the SpecVoc type:

SndObj* input1: input spectral object (output in rectangular format) from which the amplitudes will be extracted.
SndObj* input2: input spectral object (output in rectangular format) from which the phases will be extracted (phases can be thought of as the frequencies of each spectral component).
int vecsize: object vectorsize, equivalent to the FFT frame size, defaults to 1024.
float sr: sampling rate, in Hz (defaults to 44100)

Examples

Two spectra can be combined with each other using SpecVoc. The amplitudes of the first input will modulate the phases (frequencies) of the other. This resembles the operation of the classic vocoder.

```
FFT       spec1(&window, &inobj1);
FFT       spec2(&window, &inobj2);
SpecVoc vocoder(&spec1, &spec2);
IFFT       waveform(&window, &vocoder);
```

Class StringFlt

Description

The StringFlt object is a basic string resonator, built with a combination of comb, allpass and lowpass filters. Its parameters include filter (string) frequency (freq. offset and freq. control input object), feedback gain or decay factor and input object.

Construction

StringFlt()
StringFlt(**float** fr, **float** fdbgain, **SndObj*** InObj, **SndObj*** InFrObj = 0,
int vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

StringFlt(**float** fr, **SndObj*** InObj, **float** decay**, SndObj*** InFrObj = 0,
int vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Public Methods

void SetFreq(**float** fr, **SndObj*** InFrObj=0)
void SetDecay(**float** decay)
void SetFdbgain(**float** fdbgain)

Messages

[set, connect] **"frequency"**
[set] **"feedback gain"**
[set] **"decay factor"**

Details

construction

StringFlt()
StringFlt(**float** fr, **float** fdbgain, **SndObj*** InObj, **SndObj*** InFrObj = 0,
int vecsize=DEF_VECSIZE, **float** sr=DEF_SR)
StringFlt(**float** fr, **SndObj*** InObj, **float** decay**, SndObj*** InFrObj = 0,
int vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

These methods construct an object of the StringFlt class. Construction parameters are:

float fr: frequency offset, in Hz. The frequency of the filter is equivalent to that of sympathetically vibrating string.
float fdbgain: gain factor of the internal comb filter, which will rescale the signal before it re-enters the delay line. Normally < 1, anything over 1 will cause the signal to continually grow, with possibly disastrous results. Different frequencies will have different decay times for the same value of feedback gain.
float decay: alternatively, the third constructor constructs an object whose decay factor can be directly controlled. The decay factor is given in dB/sec. This allows for stretching as well as shortening the decay, as well as maintaining the same decay time across all frequencies.
SndObj* InObj: input object, pointer to the location of a SndObj-derived class.
SndObj* InFrObj: frequency control input, pointer to the location of a SndObj-derived class. The fundamental frequency can be controlled by a time-varying signal from another SndObj-derived object. A signal is fed from the input object and added to the frequency offset value. Defaults to 0, *no frequency input object*
int vecsize: vector size in samples. Size of the internal DSP buffer, defaults to DEF_VECSIZE, 256.
float sr: sampling rate in HZ. Defaults to DEF_SR, 44100.f.

public methods

void SetFreq(**float** fr, **SndObj*** InFrObj=0)
void SetFdbgain(**float** fdbgain)
void SetDecay(**float** decay)

These methods set/connect the two main parameters of the object: frequency and internal comb filter feedback gain. The messages listed above can also be used for these purposes. If SetDecay() is used, setting the feedback gain directly will have no effect.

Examples

The streson program (src/examples/streson.cpp) shows a complete example of the application of string filters:

```
StringFlt** strings = new StringFlt*[nstrs];
for(i=0;i<nstrs; i++)
 strings[i] = new StringFlt(fr[i], sound, decay, 0, DEF_VECSIZE, sr);
```

The program sets a user-defined number of strings. These are then mixed and sent to the output. The processing loop looks like this:

```
for(n=0; n < end; n++){

 input->Read();         // input from ADC
 sound->DoProcess();    // sound input
  for(i=0; i < nstrs; i++){
     strings[i]->DoProcess(); // string filters
  }

  mix->DoProcess();       // mixer
  atten->DoProcess();    // attenuation
  output->Write();

                  }
```

Class SyncGrain

Description

The **SyncGrain** class implements synchronous granular synthesis. The source sound for the grains is obtained by reading a function table containing the samples of the source waveform (use SndTable for sampled-sound or one of the wave-drawing tables for standard waveforms). The grain generator has full control of frequency (grains/sec), overall amplitude, grain pitch (a sampling increment) and grain size (in secs), both as fixed or time-varying (signal) parameters. An extra parameter is the grain pointer speed (or rate), which controls which position the generator will start reading samples in the table for each successive grain. It is measured in fractions of grain size, so a value of 1 (the default) will make each successive grain read from where the previous grain should finish. A value of 0.5 will make the next grain start at the midway position from the previous grain start and finish, etc.. A value of 0 will make the generator read always from the start of the table. This control gives extra flexibility for creating timescale modifications in the resynthesis.

SyncGrain will generate any number of parallel grain streams (which will depend on grain density/frequency), up to the olaps value (default 100). The number of streams (overlapped grains) is dtermined by grainsize*grain_freq. More grain overlaps will demand more calculations and the synthesis might not run in realtime (depending on processor power).

SyncGrain can simulate FOF-like formant synthesis, provided that a suitable shape is used as grain envelope and a sinewave as the grain wave. For this use, grain sizes of around 0.04 secs can be used. The formant centre frequency is determined by the grain pitch. Since this is a sampling increment, in order to use a frequency in Hz, that value has to be scaled by tablesize/sr. Grain frequency will determine the fundamental.

Construction

SyncGrain()
SyncGrain(**Table*** wavetable, **Table*** envtable, **float** fr, **float** amp, **float** pitch, **float** grsize, **float** prate=1.f, **SndObj*** inputfr=0, **SndObj*** inputamp=0, **SndObj*** inputpitch=0, **SndObj*** inputgrsize=0, **int** olaps=100, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Public Methods

void **SetWaveTable**(Table* wavetable)
void **SetEnvelopeTable**(Table* envtable)
void **SetFreq**(float fr, SndObj* inputfr=0)
void **SetAmp**(float amp, SndObj* inputamp=0)
void **SetPitch**(float pitch, SndObj* inputpitch=0)
void **SetGrainSize**(float grsize, SndObj* inputgrsize=0)
void **SetPointerRate**(float prate)

Messages

[set, connect] **"frequency"**
[set, connect] **"grain size"**
[set, connect]**"grain pitch"**
[set] **"pointer rate"**
[set, connect] **"amplitude"**
[connect] **"source table"**
[connect] **"envelope table"**

Details

construction

SyncGrain()
SyncGrain(**Table*** wavetable, **Table*** envtable, **float** fr, **float** amp, **float** pitch, **float** grsize, **float** prate=1.f, **SndObj*** inputfr=0, **SndObj*** inputamp=0, **SndObj*** inputpitch=0, **SndObj*** inputgrsize=0, **int** olaps=100, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

These methods construct an object of the SyncGrain class. Construction parameters are:

Table* wavetable: pointer to a table object containing the source wave for the grains.
Table* envtable: pointer to a table object containing an envelope shape to be used to shape the amplitude of the grains
float fr: grain frequency (or density) offset in Hz (grains/sec).
float amp: overall amplitude offset.
float pitch: grain pitch offset. This is a sampling increment, so that, for single-cycle wavetables, it will relate to 'real' pitch in Hz when scaled by tablesize/sr.
float grsize: grain size offset in seconds.
float prate: pointer rate (speed). The reading pointer rate in relation to grain size: a value of 1 will make the read pointer read the wavetable skiping grainsize positions along it. A value of 0 will make the table be read always from the start position.
SndObj* inputfr: frequency input, a pointer to a SndObj object whose signal will be use to control grain frequency (density).
SndObj* inputamp: amplitude input , a pointer to a SndObj object whose signal will be use to control the overall amplitude.
SndObj* inputpitch: input pitch, a pointer to a SndObj object whose signal will be use to control grain pitch.
SndObj* inputgrsize: grainsize input, a pointer to a SndObj object whose signal will be use to control grain size.
int olaps: maximum number of overlaps. It should be calculated according to max_grainsize * max_frequency. Used to allocated memory for parallel grain streams.
int vecsize: vector size in samples. Size of the internal DSP buffer, defaults to DEF_VECSIZE, 256.
float sr: sampling rate in HZ. Defaults to DEF_SR, 44100.f.

public methods

void **SetWaveTable**(Table* wavetable)
void **SetEnvelopeTable**(Table* envtable)

These methods set the tables used by a SyncGrain object for synthesis

void **SetFreq**(float fr, SndObj* inputfr=0)
void **SetAmp**(float amp, SndObj* inputamp=0)
void **SetPitch**(float pitch, SndObj* inputpitch=0)
void **SetGrainSize**(float grsize, SndObj* inputgrsize=0)

These methods set the four basic granular synthesis parameters: grain frequency (density), overall amplitude, grain pitch and size.

void **SetPointerRate**(float prate)

This method sets the pointer speed, or rate, in relation to grain size: a value of 1 will make the read pointer read the wavetable skiping grainsize positions along it. A value of 0 will make the table be read always from the start position.

All of the above operations can also be effected by sending the appropriate set/connect messages to an object, as listed above.

Examples

SyncGrain objects take their source sound from a Table object. The SndTable object stores samples read from an input SndFIO object. The example below sets a table with 88200 samples (2 secs at 44100Hz) from the first channel of a SndFIO object named input:

```
SndTable  sound(88200, &input, 1);
TriSegTable envel(2000, 0, 100.f, 1.f, 1500.f, 0.75f, 400.f, 0.f, 0.8f);
SyncGrain  corn(&sound, &envel, 40.f, 10000.f, 1.f, 0.05f, 0.5f);
```

The SyncGrain object will generate 40 grains of 50 msecs each, keeping the original pitch and timescale. There are 2 overlaps between grains, because of the relationship between density and duration (40*0.05 = 2). If the pointer rate was running at the original speed, the sound would be time-compressed. With pointer rate at 0.5, we restore the original length. The processing loop only requires the call to SyncGrain::DoProcess() [and the output object Write()]:

```
while(processing_on){

corn.DoProcess();
output.Write();

}
```

Class Table

Description

The Table class is the abstract base class for all the maths function-table objects in the library. It provides basic methods to access the core elements of the table model, as well as the pure virtual methods **MakeTable()** and **ErrorMessage()**, which are implemented in the derived classes, and a virtual destructor. As this is an abstract class, it does not have constructors and it is not used directly.

Public Methods

long GetLen()
float* GetTable()
float Lookup(**int** pos)
virtual short MakeTable()
virtual char* ErrorMessage()

Details

public methods

long GetLen()

This method returns the length of the function table.

float* GetTable()

This method provides basic access to the function table itself. It returns a pointer to the first location of an array of floats, the length of which can be obtained by GetLen().

float Lookup(**int** pos)

This method performs a tableLookup combined with a simple modulus operation, returning the value at the table mod[table_length]. It is an alternative way to access the table values (the other is to use the pointer returned by GetTable())

virtual short MakeTable()
virtual char* ErrorMessage()

These methods should be implemented in the derived classes to provide the class functionality. MakeTable() builds the function table with an arbitrary algorithm or formula. ErrorMessage() returns error strings relative to error codes. Derived classes should call MakeTable() in the their constructor.

Class Tap

Description

The Tap class is used to create a fixed-delaytime tap on a DelayLine object. The delaytime should be set to be not bigger than the tapped DelayLine object delaytime.

Construction

Tap()
Tap(**float** delaytime, **DelayLine*** DLine, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Public Methods

void SetDelayLine(**DelayLine*** DLine)
void SetDelayTime(**float** delaytime)

Messages

[connect] **“delay line”**
[set] **“delaytime”**

Details

construction

Tap()
Tap(**float** delaytime, **DelayLine*** DLine, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

These methods construct an object of the Tap class. Construction parameters are:

float delaytime: delay time of the tap, in seconds. It must be smaller than the delaytime of the tapped object
DelayLine* DLine: the DelayLine object which will be tapped by this class
int vecsize: vector size in samples. Size of the internal DSP buffer, defaults to DEF_VECSIZE, 256.
float sr: sampling rate in HZ. Defaults to DEF_SR, 44100.f.

public methods

void SetDelayLine(**DelayLine*** DLine)
void SetDelayTime(**float** delaytime)

These methods connect the DelayLine object to be tapped and set the delaytime, respectively. The messages listed above can be used for the same purposes.

Examples

The following connections create three taps into a delay line:

```
DelayLine delay(1.f, &oscillator);
Tap tap1(0.01f, &delay);
Tap tap2(0.045f, & delay);
Tap tap3(0.078f, &delay);
// constructs a DelayLine object and taps it at three points: at 10, 45 and 78 milliseconds
```

Class Tapi

Description

The Tapi class is used to create a variable-delaytime interpolating tap on a DelayLine object. The variable delaytime should never be bigger than the tapped DelayLine object delaytime

Construction

Tapi()
Tapi(**SndObj*** delayinput, **DelayLine*** DLine, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Public Methods

short SetDelayInput**(SndObj*** InObj)

Messages

[connect] **"delay input"**

Details

construction

Tapi()
Tapi(**SndObj*** delayinput, **DelayLine*** DLine, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

These methods construct an object of the Tapi class. Construction parameters are:

SndObj* delayinput: sets the variable delay input object. This object would normally output a signal varying at most from 0 to the maximum delay time set by the DelayLine object being tapped.
DelayLine* DLine: the DelayLine object which will be tapped by this class
int vecsize: vector size in samples. Size of the internal DSP buffer, defaults to DEF_VECSIZE, 256.
float sr: sampling rate in HZ. Defaults to DEF_SR, 44100.f.

public methods

short SetDelayInput**(SndObj*** InObj)

This method sets the variable delay tap input object.

Examples

The following example sets up a flanging effect created by varying the delay of the tap between close to 0 and 10 msecs. The oscillator is using a waveshape that is always positive (a hamming window shape).

```
Oscili   lfo(&hamming, 1.2f, 0.01f);
DelayLine delay(0.02f, &inobj);
Tapi  flange(&lfo, &delay);
```

Class TpTz

Description

This class implements a user-defined two-pole two-zero filter from input coefficients.

Construction

TpTz();
TpTz(**double** a, **double** a1, **double** a2, **double** b1, **double** b2, **SndObj*** input,
int vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Public Methods

void SetParam(**double** a, **double** a1, **double** a2, **double** b1, **double** b2)

Messages

[set] **"coefficient a0"**
[set] **"coefficient a1"**
[set] **"coefficient a2"**
[set] **"coefficient b1"**
[set] **"coefficient b2"**

Details

construction

TpTz();
TpTz(**double** a, **double** a1, **double** a2, **double** b1, **double** b2, **SndObj*** input,
int vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

These methods construct a TpTz object:

double a: the coefficient 'a', the input scaling coefficient.
double a1: the coefficient 'a1', multiplier of the 1-sample input delay.
double a2: the coefficient 'a2', multiplier of the 2-sample input delay.
double b1: the coefficient 'b1', multiplier of the 1-sample delayed output.
double b2: the coefficient 'b2', multiplier of the 2-sample delayed output.
SndObj* input: input signal object, a SndObj-derived object.
int vecsize: the object vector size, defaults to 256.
float sr: the object sampling rate: defaults to 44100.

public methods

void SetParam(**double** a, **double** a1, **double** a2, **double** b1, **double** b2)

This method sets the filter coefficients, as described above. The respective set messages can also be used to set individual coefficient values.

Examples

TpTz objects are provided so that any 2^{nd} order feedback filter can be implemented from its coefficients. The example below shows the parameters to implement a butterworth band-reject design, with a bandwidth of 100Hz and a centre frequency of 1000Hz:

```
double tmp1 = tan(PI*100/DEF_SR);
double tmp2 = 2*cos(2*PI*1000/DEF_SR);
double  par1 = 1/(1+tmp1);
double  par2 = -tmp2*par1
```

```
double  par3 = (1-tmp1)*par2

TpTz  butterworth_bandreject(par1,par2,par1,par2,par3,&inobj);
```

The processing loop will then feature a call to the filter DoProcess() method:

```
while(processing_on){
(...)
butterworth_bandreject.DoProcess();
(...)
}
```

Class TrisegTable

Description

The SndTable object builds a three-segment function table, with linear or exponentially-shaped segments

Construction

TrisegTable()
TrisegTable(**long** L, **long** L, **float** init, **float** seg1, **float** p1, **float** seg2, **float** p2, **float** seg3, **float** fin, **float** type = 0.f)
TrisegTable(**long** L, **float*** TSPoints, **float** type = 0.f)

Public Methods

void SetCurve(**float** init, **float** seg1, **float** p1, **float** seg2, **float** p2, **float** seg3, **float** fin, **float** type = 0.f)
void SetCurve(**float*** TSPoints, **float** type = 0.f)

Details

construction

TrisegTable()
TrisegTable(**long** L, **long** L, **float** init, **float** seg1, **float** p1, **float** seg2, **float** p2, **float** seg3, **float** fin, **float** type = 0.f)
TrisegTable(**long** L, **float*** TSPoints, **float** type = 0.f)

Constructs a TrisegTable object.

long L: table length.
float init, fin: inital and final points of the curve.
float p1, p2: intermediary points.
float seg1, seg2, seg3: lengths of the three segments.
float type: type of curve. Linear = 0, inverse exponential < 0 < exponential.
float* TSPoints: pointer to the location of a 7-element float array containing the three-segment curve breakpoints. They should be arranged in the following order: { init, seg1, p1, seg2, p2, seg3, fin } .

public methods

void SetCurve(**float** init, **float** seg1, **float** p1, **float** seg2, **float** p2, **float** seg3, **float** fin, **float** type = 0.f)
void SetCurve(**float*** TSPoints, **float** type = 0.f)

These methods set the function table parameters. MakeTable() should be invoked after any parameter resetting.

Class Unit

Description

This object generates signals which can be used for testing applications. The options are unit sample (1 sample of **amp** value followed by 0s), unit step (DC with **amp** offset) and ramp (ramping signal starting from **amp** increasing **step** times each sample)

Construction

Unit()
Unit(**float** amp, **short** mode=UNIT_SAMPLE, **float** step=0.f, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Public Methods

void SetAmp(**float** amp)
void SetStep(**float** step)

Messages

[set] **“mode”**
[set] **“step”**
[set] **“amplitude”**

Details

construction

Unit()
Unit(**float** amp, **short** mode=UNIT_SAMPLE, **float** step=0.f, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

These methods construct an object of the Unit class. Construction parameters are:

float amp: amplitude.
short mode: type of output test signal, UNIT_SAMPLE, UNIT_STEP, RAMP
float step: value added to the previous sample every sample period (starting from **amp**) , used in the RAMP mode.
int vecsize: vector size in samples. Size of the internal DSP buffer, defaults to DEF_VECSIZE, 256.
float sr: sampling rate in HZ. Defaults to DEF_SR, 44100.f

public methods

void SetAmp(**float** amp)
void SetStep(**float** step)

These methods set the two Unit parameters, step and amp. These, plus the generation mode, can also be set using the messages listed above.

Examples

A ramp signal can be generated with the following Unit object:

Unit ramp(0, RAMP, 1.f);

This will generate a ramp from 0, increasing by 1 every sample.

Class UsrDefTable

Description

The UsrDefTable implements a multi-purpose user-defined table object. It builds the object around an user-defined array of floats.

Construction

UsrDefTable()
UsrDefTable(**long** L, **float*** values)

Public Methods

void SetTable(**long** L, **float*** values)

Details

construction

UsrDefTable()
UsrDefTable(**long** L, **float*** values)

Constructs a UsrDefTable object.

long L: table length.
float* values: pointer to the first location of an array of floats of length L.

public methods

void SetTable(**long** L, **float*** values)

This method sets the function table parameters. MakeTable() should be invoked after any parameter resetting.

Class UsrHarmTable

Description

Harmonic function table. Generates any type of wave with any number of harmonics.

Construction

UsrHarmTable()
UsrHarmTable(**long** L, **int** harm, **float*** amps)

Public Methods

void SetHarm(**int** harm, **float** *amps)

Details

construction

UsrHarmTable()
UsrHarmTable(**long** L, **int** harm, **float*** amps)

Constructs a HarmTable object.

long L: table length.
int harm: number of harmonics. Defaults to 1
float* amps: pointer to the first location of an array of floats containing the individual harmonics amplitudes, starting from the lowest order harmonic. The size of the array is related to the number of harmonics defined.

public methods

void SetHarm(**int** harm, **float** *amps)

This method sets the function table parameters. MakeTable() should be invoked after any parameter resetting.

Class VDelay

Description

The VDelay object is a variable delay processor with feedback and feedfoward connections. The delay time is controlled by an input SndObj-derived object, and the various gain controls can also be dynamically controlled by other objects

Construction

VDelay()
VDelay(**float** maxdelaytime, **float** fdbgain, **float** fwdgain, **float** dirgain, **SndObj*** InObj,
SndObj* InVdtime, **SndObj*** InFdbgain=0, **SndObj*** InFwdgain=0,
SndObj* InDirgain=0, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)
VDelay(**float** maxdelaytime, **float** delaytime, **float** fdbgain, **float** fwdgain, **float** dirgain,
SndObj* InObj, **SndObj*** InVdtime=0, **SndObj*** InFdbgain=0, **SndObj*** InFwdgain=0,
SndObj* InDirgain=0, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

Public Methods

void SetMaxDelayTime(**float** MaxDelaytime)
void SetDelayTime(**float** delaytime)
void SetVdtInput(**SndObj*** InVdtime)
void SetFdbgain(**float** fdbgain, **SndObj*** InFdbgain=0)
void SetFwdgain(**float** fwdgain, **SndObj*** InFwdgain=0)
void SetDirgain(**float** fwdgain, **SndObj*** InDirgain=0)

Messages

[set,connect] **"delaytime"**
[set] **"maxdelaytime"**
[set,connect] **"direct gain"**
[set,connect] **"feedback gain"**
[set,connect] **"feedforward gain"**

Details

construction

VDelay()
VDelay(**float** maxdelaytime, **float** fdbgain, **float** fwdgain, **float** dirgain, **SndObj*** InObj,
SndObj* InVdtime, **SndObj*** InFdbgain=0, **SndObj*** InFwdgain=0,
SndObj* InDirgain=0, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)
VDelay(**float** maxdelaytime, **float** delaytime, **float** fdbgain, **float** fwdgain, **float** dirgain,
SndObj* InObj, **SndObj*** InVdtime=0, **SndObj*** InFdbgain=0, **SndObj*** InFwdgain=0,
SndObj* InDirgain=0, **int** vecsize=DEF_VECSIZE, **float** sr=DEF_SR)

These methods construct an object of the VDelay class. Construction parameters are:

float maxdelaytime: sets the max delay time of the process, in seconds. If the variable delay input signal exceed this value, it is ignored and the delay time is set to *maxdelaytime* seconds. The variable delay input signal varies the input above and below *maxdelaytime/2* seconds (a positive signal increase the delay above it and negative one decreases it).
float delaytime: delaytime offset, added to the signal from the input variable delaytime object, if there is one [use third constructor for this parameter].
float fdbgain: gain offset factor of the signal re-entering the delay line. Normally < 1.
float fwdgain: gain offset factor of the forward fed signal. Normally < 1.
float dirgain: gain offset factor of the signal bypassing the network (direct signal), which is added to the signal coming out of it. Normally < 1.

SndObj* InObj: input object, pointer to the location of a SndObj-derived class.
SndObj* InVdtime: variable delaytime control input object, pointer to the location of a SndObj-derived class. This input object controls the varying delay time and it should generate a signal centered on 0 (i.e. having a DC offset of 0), when there is no delaytime offset. For flanging applications, a low-frequency oscillator is normally used. In the third constructor this argument defaults to 0.
SndObj* InFdbGain: feedback gain control input, pointer to the location of a SndObj-derived class. The output signal from this object is added to the fdbgain value before it is used. Defaults to 0, *no frequency input object*
SndObj* InFwdGain: feedforward gain control input, pointer to the location of a SndObj-derived class. The output signal from this object is added to the fwdgain value before it is used. Defaults to 0, *no frequency input object*
SndObj* InDirGain: direct gain control input, pointer to the location of a SndObj-derived class. The output signal from this object is added to the dirgain value before it is used. Defaults to 0, *no frequency input object*
int vecsize: vector size in samples. Size of the internal DSP buffer, defaults to DEF_VECSIZE, 256.
float sr: sampling rate in HZ. Defaults to DEF_SR, 44100.f.

public methods

void SetMaxDelayTime(**float** MaxDelaytime)
void SetDelayTime(**float** delaytime)
void SetVdtInput(**SndObj*** InVdtime)
void SetFdbgain(**float** fdbgain, **SndObj*** InFdbgain=0)
void SetFwdgain(**float** fwdgain, **SndObj*** InFwdgain=0)
void SetDirgain(**float** fwdgain, **SndObj*** InDirgain=0)

These methods set the various parameters associated with the object. Set/Connect messages, as listed above, can also be used to change these values.

Examples

VDelay objects are used to generate variable-delay effects, such as vibrato, chorus and flanging. The connections shown below demonstrating the implementation of a flanging effect:

```
Oscili   lfo(&sine, 1.2f, 0.01f);
VDelay  flange(0.02f, 0.5f, 0.5f, 0.5f, &inobj, &lfo);
```

The processing loop would look something like this:

```
while(processing_on){

inobj.DoProcess();
lfo.DoProcess();
flange.DoProcess();
output.Write();

}
```