Loris Reference Manual

1.3

Generated by Doxygen 1.3.4

Thu Apr 7 22:49:00 2005

Contents

Chapter 1

Loris Hierarchical Index

1.1 Loris Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 2

Loris Class Index

2.1 Loris Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 3

Loris Class Documentation

3.1 Loris::AiffFile Class Reference

Class AiffFile represents sample data in a AIFF-format samples file, and manages file I/O and sample conversion.

```
#include <AiffFile.h>
```

Public Types

- typedef std::vector< double > samples_type

  The type of the sample storage in an AiffFile.

- typedef samples_type::size_type size_type

  The type of all size parameters for AiffFile.

- typedef std::vector< Marker > markers_type

  The type of AIFF marker storage in an AiffFile.

Public Member Functions

- AiffFile (const std::string &filename)

  Initialize an instance of AiffFile by importing sample data from the file having the specified filename or path.

- template<typename Iter> AiffFile (Iter begin_partials, Iter end_partials, double samplerate, double fadeTime=.001)

  Initialize an instance of AiffFile with samples rendered from a sequnence of Partials.

- AiffFile (double samplerate, size_type numFrames=0)

  Initialize an instance of AiffFile having the specified sample rate, preallocating numFrames samples, initialized to zero.

- AiffFile (const double ∗buffer, size_type bufferlength, double samplerate)

  Initialize an instance of AiffFile from a buffer of sample data, with the specified sample rate.

- AiffFile (const std::vector< double > &vec, double samplerate)

  Initialize an instance of AiffFile from a vector of sample data, with the specified sample rate.

- AiffFile (const AiffFile &other)

  Initialize this and AiffFile that is an exact copy, having all the same sample data, as another AiffFile.

- AiffFile & operator= (const AiffFile &rhs)

  Assignment operator: change this AiffFile to be an exact copy of the specified AiffFile, rhs, that is, having the same sample data.

- markers_type & markers (void)

  Return a reference to the Marker (see Marker.h) container for this AiffFile.

- const markers_type & markers (void) const

  Return a const reference to the Marker (see Marker.h) container for this AiffFile.

- double midiNoteNumber (void) const

  Return the fractional MIDI note number assigned to this AiffFile.

- size_type numFrames (void) const

  Return the number of sample frames represented in this AiffFile.

- double sampleRate (void) const

  Return the sampling freqency in Hz for the sample data in this AiffFile.

- samples_type & samples (void)

  Return a reference (or const reference) to the vector containing the floating-point sample data for this AiffFile.

- const samples_type & samples (void) const

  Return a const reference (or const reference) to the vector containing the floating-point sample data for this AiffFile.

- void addPartial (const Loris::Partial &p, double fadeTime=.001)

  Render the specified Partial using the (optionally) specified Partial fade time, and accumulate the resulting samples into the sample vector for this AiffFile.

- template<typename Iter> void addPartials (Iter begin_partials, Iter end_partials, double fadeTime=.001)

  Accumulate samples rendered from a sequence of Partials.

- void setMidiNoteNumber (double nn)

  Set the fractional MIDI note number assigned to this AiffFile.

- void write (const std::string &filename, unsigned int bps=16)

  Export the sample data represented by this AiffFile to the file having the specified filename or path.

3.1.1 Detailed Description

Class AiffFile represents sample data in a AIFF-format samples file, and manages file I/O and sample conversion.

Since the sound analysis and synthesis algorithms in Loris and the reassigned bandwidth-enhanced representation are monaural, AiffFile manages only monaural (single channel) AIFF-format samples files.

3.1.2 Constructor & Destructor Documentation

3.1.2.1 Loris::AiffFile::AiffFile (const std::string & *filename*) `[explicit]`

Initialize an instance of AiffFile by importing sample data from the file having the specified filename or path.

Parameters:

filename is the name or path of an AIFF samples file

3.1.2.2 template<typename Iter> Loris::AiffFile::AiffFile (Iter *begin_partials*, Iter *end_partials*, double *samplerate*, double *fadeTime* = .001)

Initialize an instance of AiffFile with samples rendered from a sequnence of Partials.

The Partials in the specified half-open (STL-style) range are rendered at the specified sample rate, using the (optionally) specified Partial fade time (see Synthesizer.h for an examplanation of fade time).

Parameters:

begin_partials is the beginning of a sequence of Partials

end_partials is (one-past) the end of a sequence of Partials

samplerate is the rate at which Partials are rendered

fadeTime is the Partial fade time for rendering the Partials on the specified range. If unspecified, the default fade time is 1 ms.

If compiled with NO_TEMPLATE_MEMBERS defined, this member accepts only PartialList::const_iterator arguments.

3.1.2.3 Loris::AiffFile::AiffFile (double *samplerate*, size_type *numFrames* = 0) `[explicit]`

Initialize an instance of AiffFile having the specified sample rate, preallocating numFrames samples, initialized to zero.

Parameters:

samplerate is the rate at which Partials are rendered

numFrames is the initial number of (zero) samples. If unspecified, no samples are preallocated.

3.1.2.4 Loris::AiffFile::AiffFile (const double ∗ *buffer*, size_type *bufferlength*, double *samplerate*)

Initialize an instance of AiffFile from a buffer of sample data, with the specified sample rate.

Parameters:

buffer is a pointer to a buffer of floating point samples.

bufferlength is the number of samples in the buffer.

samplerate is the sample rate of the samples in the buffer.

3.1.2.5 Loris::AiffFile::AiffFile (const std::vector< double > & *vec*, double *samplerate*)

Initialize an instance of AiffFile from a vector of sample data, with the specified sample rate.

Parameters:

vec is a vector of floating point samples.

samplerate is the sample rate of the samples in the vector.

3.1.2.6 Loris::AiffFile::AiffFile (const AiffFile & *other*)

Initialize this and AiffFile that is an exact copy, having all the same sample data, as another AiffFile.

Parameters:

other is the AiffFile to copy

3.1.3 Member Function Documentation

3.1.3.1 void Loris::AiffFile::addPartial (const Loris::Partial & *p*, double *fadeTime* = .001)

Render the specified Partial using the (optionally) specified Partial fade time, and accumulate the resulting samples into the sample vector for this AiffFile.

Parameters:

p is the partial to render into this AiffFile

fadeTime is the Partial fade time for rendering the Partials on the specified range. If unspecified, the default fade time is 1 ms.

3.1.3.2 template<typename Iter> void Loris::AiffFile::addPartials (Iter *begin_partials*, Iter *end_partials*, double *fadeTime* = .001)

Accumulate samples rendered from a sequence of Partials.

The Partials in the specified half-open (STL-style) range are rendered at this AiffFile's sample rate, using the (optionally) specified Partial fade time (see Synthesizer.h for an examplanation of fade time).

Parameters:

begin_partials is the beginning of a sequence of Partials

end_partials is (one-past) the end of a sequence of Partials

fadeTime is the Partial fade time for rendering the Partials on the specified range. If unspecified, the default fade time is 1 ms.

If compiled with NO_TEMPLATE_MEMBERS defined, this member accepts only PartialList::const_iterator arguments.

3.1.3.3 double Loris::AiffFile::midiNoteNumber (void) const

Return the fractional MIDI note number assigned to this AiffFile.

If the sound has no definable pitch, note number 60.0 is used.

3.1.3.4 size_type Loris::AiffFile::numFrames (void) const

Return the number of sample frames represented in this AiffFile.

A sample frame contains one sample per channel for a single sample interval (e.g. mono and stereo samples files having a sample rate of 44100 Hz both have 44100 sample frames per second of audio samples).

3.1.3.5 AiffFile& Loris::AiffFile::operator= (const AiffFile & *rhs*)

Assignment operator: change this AiffFile to be an exact copy of the specified AiffFile, rhs, that is, having the same sample data.

Parameters:

rhs is the AiffFile to replicate

3.1.3.6 void Loris::AiffFile::setMidiNoteNumber (double *nn*)

Set the fractional MIDI note number assigned to this AiffFile.

If the sound has no definable pitch, use note number 60.0 (the default).

Parameters:

nn is a fractional MIDI note number, 60 is middle C.

3.1.3.7 void Loris::AiffFile::write (const std::string & *filename*, unsigned int *bps* = 16)

Export the sample data represented by this AiffFile to the file having the specified filename or path.

Export signed integer samples of the specified size, in bits (8, 16, 24, or 32).

Parameters:

filename is the name or path of the AIFF samples file to be created or overwritten.

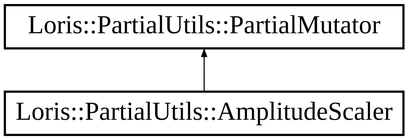
bps is the number of bits per sample to store in the samples file (8, 16, 24, or 32).If unspeicified, 16 bits

3.2 Loris::PartialUtils::AmplitudeScaler Class Reference

Scale the amplitude of the specified Partial according to an envelope representing a time-varying amplitude scale value.

`#include <PartialUtils.h>`

Inheritance diagram for Loris::PartialUtils::AmplitudeScaler::


Loris::PartialUtils::PartialMutator
Loris::PartialUtils::AmplitudeScaler

Public Member Functions

- AmplitudeScaler (double x)

  Construct a new AmplitudeScaler from a constant scale factor.

- AmplitudeScaler (const Envelope &e)

  Construct a new AmplitudeScaler from an Envelope representing a time-varying scale factor.

- void operator() (Partial &p) const

  Function call operator: apply a scale factor to the specified Partial.

3.2.1 Detailed Description

Scale the amplitude of the specified Partial according to an envelope representing a time-varying amplitude scale value.

3.3 Loris::Analyzer Class Reference

Class Analyzer represents a configuration of parameters for performing Reassigned Bandwidth-Enhanced Additive Analysis of sampled sounds.

```
#include <Analyzer.h>
```

Public Member Functions

- Analyzer (double resolutionHz)

  Construct a new Analyzer configured with the given frequency resolution (minimum instantaneous frequency difference between Partials).

- Analyzer (double resolutionHz, double windowWidthHz)

  Construct a new Analyzer configured with the given frequency resolution (minimum instantaneous frequency difference between Partials) and analysis window width (main lobe, zero-to-zero).

- Analyzer (const Analyzer &other)

  Construct a new Analyzer having identical parameter configuration to another Analyzer.

- ~Analyzer (void)

  Destroy this Analyzer.

- Analyzer & operator= (const Analyzer &rhs)

  Construct a new Analyzer having identical parameter configuration to another Analyzer.

- void configure (double resolutionHz, double windowWidthHz)

  Configure this Analyzer with the given frequency resolution (minimum instantaneous frequency difference between Partials) and analysis window width (main lobe, zero-to-zero, in Hz).

- void analyze (const std::vector< double > &vec, double srate)

  Analyze a vector of (mono) samples at the given sample rate (in Hz) and append the extracted Partials to Analyzer's PartialList (std::list of Partials).

- void analyze (const double *bufBegin, const double *bufEnd, double srate)

  Analyze a range of (mono) samples at the given sample rate (in Hz) and collect the resulting Partials.

- void analyze (const std::vector< double > &vec, double srate, const Envelope &reference)

  Analyze a vector of (mono) samples at the given sample rate (in Hz) and append the extracted Partials to Analyzer's PartialList (std::list of Partials).

- void analyze (const double ∗bufBegin, const double ∗bufEnd, double srate, const Envelope &reference)

  Analyze a range of (mono) samples at the given sample rate (in Hz) and append the extracted Partials to Analyzer's PartialList (std::list of Partials).

- double ampFloor (void) const

  Return the amplitude floor (lowest detected spectral amplitude), in (negative) dB, for this Analyzer.

- bool associateBandwidth (void) const

  Return true if this Analyzer is configured to peform bandwidth association to distribute noise energy among extracted Partials, and false if noise energy will be collected in noise Partials, labeled -1 in this Analyzer's PartialList.

- double bwRegionWidth (void) const

  Return the width (in Hz) of the Bandwidth Association regions used by this Analyzer.

- double cropTime (void) const

  Return the crop time (maximum temporal displacement of a time- frequency data point from the time-domain center of the analysis window, beyond which data points are considered "unreliable") for this Analyzer.

- double freqDrift (void) const

  Return the maximum allowable frequency difference between consecutive Breakpoints in a Partial envelope for this Analyzer.

- double freqFloor (void) const

  Return the frequency floor (minimum instantaneous Partial frequency), in Hz, for this Analyzer.

- double freqResolution (void) const

  Return the frequency resolution (minimum instantaneous frequency difference between Partials) for this Analyzer.

- double hopTime (void) const

  Return the hop time (which corresponds approximately to the average density of Partial envelope Breakpoint data) for this Analyzer.

- double sidelobeLevel (void) const

  Return the sidelobe attenutation level for the Kaiser analysis window in positive dB.

- double windowWidth (void) const

  Return the frequency-domain main lobe width (measured between zero-crossings) of the analysis window used by this Analyzer.

- void setAmpFloor (double x)

  Set the amplitude floor (lowest detected spectral amplitude), in (negative) dB, for this Analyzer.

- void setBwRegionWidth (double x)

  Set the width (in Hz) of the Bandwidth Association regions used by this Analyzer.

- void setCropTime (double x)

  Set the crop time (maximum temporal displacement of a time- frequency data point from the time-domain center of the analysis window, beyond which data points are considered "unreliable") for this Analyzer.

- void setFreqDrift (double x)

  Set the maximum allowable frequency difference between consecutive Breakpoints in a Partial envelope for this Analyzer.

- void setFreqFloor (double x)

  Set the frequency floor (minimum instantaneous Partial frequency), in Hz, for this Analyzer.

- void setFreqResolution (double x)

  Set the frequency resolution (minimum instantaneous frequency difference between Partials) for this Analyzer.

- void setHopTime (double x)

  Set the hop time (which corresponds approximately to the average density of Partial envelope Breakpoint data) for this Analyzer.

- void setSidelobeLevel (double x)

  Set the sidelobe attenutation level for the Kaiser analysis window in positive dB.

- void setWindowWidth (double x)

  Set the frequency-domain main lobe width (measured between zero-crossings) of the analysis window used by this Analyzer.

- PartialList & partials (void)

  Return a mutable reference to this Analyzer's list of analyzed Partials.

- const PartialList & partials (void) const

  Return an immutable (const) reference to this Analyzer's list of analyzed Partials.

3.3.1 Detailed Description

Class Analyzer represents a configuration of parameters for performing Reassigned Bandwidth-Enhanced Additive Analysis of sampled sounds.

The analysis process yields a collection of Partials, each having a trio of synchronous, non-uniformly- sampled breakpoint envelopes representing the time-varying frequency, amplitude, and noisiness of a single bandwidth- enhanced sinusoid. These Partials are accumulated in the Analyzer.

The core analysis parameter is the frequency resolution, the minimum instantaneous frequency spacing between partials. All other parameters are initially configured according to this parameter (and the analysis window width, if specified). Subsequent parameter mutations are independent.

For more information about Reassigned Bandwidth-Enhanced Analysis and the Reassigned Bandwidth-Enhanced Additive Sound Model, refer to the Loris website: www.cerlsoundgroup.org/Loris/.

3.3.2 Constructor & Destructor Documentation

3.3.2.1 Loris::Analyzer::Analyzer (double *resolutionHz*) `[explicit]`

Construct a new Analyzer configured with the given frequency resolution (minimum instantaneous frequency difference between Partials).

All other Analyzer parameters are computed from the specified frequency resolution.

Parameters:

resolutionHz is the frequency resolution in Hz.

3.3.2.2 Loris::Analyzer::Analyzer (double *resolutionHz*, double *windowWidthHz*)

Construct a new Analyzer configured with the given frequency resolution (minimum instantaneous frequency difference between Partials) and analysis window width (main lobe, zero-to-zero).

All other Analyzer parameters are computed from the specified resolution and window width.

Parameters:

resolutionHz is the frequency resolution in Hz.

windowWidthHz is the main lobe width of the Kaiser analysis window in Hz.

3.3.2.3 Loris::Analyzer::Analyzer (const Analyzer & *other*)

Construct a new Analyzer having identical parameter configuration to another Analyzer.

The list of collected Partials is not copied.

Parameters:

other is the Analyzer to copy.

3.3.3 Member Function Documentation

3.3.3.1 double Loris::Analyzer::ampFloor (void) const

Return the amplitude floor (lowest detected spectral amplitude), in (negative) dB, for this Analyzer.

3.3.3.2 void Loris::Analyzer::analyze (const double ∗ *bufBegin*, const double ∗ *bufEnd*, double *srate*, const Envelope & *reference*)

Analyze a range of (mono) samples at the given sample rate (in Hz) and append the extracted Partials to Analyzer's PartialList (std::list of Partials).

Use the specified envelope as a frequency reference for Partial tracking.

Parameters:

bufBegin is a pointer to a buffer of floating point samples

bufEnd is (one-past) the end of a buffer of floating point samples

srate is the sample rate of the samples in the buffer

- ***reference*** is an Envelope having the approximate frequency contour expected of the resulting Partials.

3.3.3.3 void Loris::Analyzer::analyze (const std::vector< double > & *vec*, double *srate*, const Envelope & *reference*)

Analyze a vector of (mono) samples at the given sample rate (in Hz) and append the extracted Partials to Analyzer's PartialList (std::list of Partials).

Use the specified envelope as a frequency reference for Partial tracking.

Parameters:

- ***vec*** is a vector of floating point samples
- ***srate*** is the sample rate of the samples in the vector
- ***reference*** is an Envelope having the approximate frequency contour expected of the resulting Partials.

3.3.3.4 void Loris::Analyzer::analyze (const double ∗ *bufBegin*, const double ∗ *bufEnd*, double *srate*)

Analyze a range of (mono) samples at the given sample rate (in Hz) and collect the resulting Partials.

Parameters:

- ***bufBegin*** is a pointer to a buffer of floating point samples
- ***bufEnd*** is (one-past) the end of a buffer of floating point samples
- ***srate*** is the sample rate of the samples in the buffer

3.3.3.5 void Loris::Analyzer::analyze (const std::vector< double > & *vec*, double *srate*)

Analyze a vector of (mono) samples at the given sample rate (in Hz) and append the extracted Partials to Analyzer's PartialList (std::list of Partials).

Parameters:

- ***vec*** is a vector of floating point samples

srate is the sample rate of the samples in the vector

3.3.3.6 double Loris::Analyzer::bwRegionWidth (void) const

Return the width (in Hz) of the Bandwidth Association regions used by this Analyzer.

If zero, bandwidth enhancement is disabled.

3.3.3.7 void Loris::Analyzer::configure (double *resolutionHz*, double *windowWidthHz*)

Configure this Analyzer with the given frequency resolution (minimum instantaneous frequency difference between Partials) and analysis window width (main lobe, zero-to-zero, in Hz).

All other Analyzer parameters are (re-)computed from the frequency resolution and window width.

Parameters:

resolutionHz is the frequency resolution in Hz.

windowWidthHz is the main lobe width of the Kaiser analysis window in Hz.

There are three categories of analysis parameters:

- the resolution, and params that are usually related to (or identical to) the resolution (frequency floor and drift)
- the window width and params that are usually related to (or identical to) the window width (hop and crop times)
- independent parameters (bw region width and amp floor)

3.3.3.8 double Loris::Analyzer::freqDrift (void) const

Return the maximum allowable frequency difference between consecutive Breakpoints in a Partial envelope for this Analyzer.

3.3.3.9 double Loris::Analyzer::freqFloor (void) const

Return the frequency floor (minimum instantaneous Partial frequency), in Hz, for this Analyzer.

3.3.3.10 double Loris::Analyzer::freqResolution (void) const

Return the frequency resolution (minimum instantaneous frequency difference between Partials) for this Analyzer.

3.3.3.11 Analyzer& Loris::Analyzer::operator= (const Analyzer & *rhs*)

Construct a new Analyzer having identical parameter configuration to another Analyzer.

The list of collected Partials is not copied.

Parameters:

rhs is the Analyzer to copy.

3.3.3.12 void Loris::Analyzer::setAmpFloor (double *x*)

Set the amplitude floor (lowest detected spectral amplitude), in (negative) dB, for this Analyzer.

Parameters:

x is the new value of this parameter.

3.3.3.13 void Loris::Analyzer::setBwRegionWidth (double *x*)

Set the width (in Hz) of the Bandwidth Association regions used by this Analyzer.

If zero, bandwidth enhancement is disabled.

Parameters:

x is the new value of this parameter.

3.3.3.14 void Loris::Analyzer::setCropTime (double *x*)

Set the crop time (maximum temporal displacement of a time- frequency data point from the time-domain center of the analysis window, beyond which data points are considered "unreliable") for this Analyzer.

Parameters:

x is the new value of this parameter.

3.3.3.15 void Loris::Analyzer::setFreqDrift (double *x*)

Set the maximum allowable frequency difference between consecutive Breakpoints in a Partial envelope for this Analyzer.

Parameters:

x is the new value of this parameter.

3.3.3.16 void Loris::Analyzer::setFreqFloor (double *x*)

Set the frequency floor (minimum instantaneous Partial frequency), in Hz, for this Analyzer.

Parameters:

x is the new value of this parameter.

3.3.3.17 void Loris::Analyzer::setFreqResolution (double *x*)

Set the frequency resolution (minimum instantaneous frequency difference between Partials) for this Analyzer.

(Does not cause other parameters to be recomputed.)

Parameters:

x is the new value of this parameter.

3.3.3.18 void Loris::Analyzer::setHopTime (double *x*)

Set the hop time (which corresponds approximately to the average density of Partial envelope Breakpoint data) for this Analyzer.

Parameters:

x is the new value of this parameter.

3.3.3.19 void Loris::Analyzer::setSidelobeLevel (double *x*)

Set the sidelobe attenutation level for the Kaiser analysis window in positive dB.

More negative numbers (e.g. -90) give very good sidelobe rejection but cause the window to be longer in time. Less negative numbers raise the level of the sidelobes, increasing the likelihood of frequency-domain interference, but allow the window to be shorter in time.

Parameters:

x is the new value of this parameter.

3.3.3.20 void Loris::Analyzer::setWindowWidth (double *x*)

Set the frequency-domain main lobe width (measured between zero-crossings) of the analysis window used by this Analyzer.

Parameters:

x is the new value of this parameter.

3.3.3.21 double Loris::Analyzer::sidelobeLevel (void) const

Return the sidelobe attenutation level for the Kaiser analysis window in positive dB.

Larger numbers (e.g. 90) give very good sidelobe rejection but cause the window to be longer in time. Smaller numbers (like 60) raise the level of the sidelobes, increasing the likelihood of frequency-domain interference, but allow the window to be shorter in time.

3.3.3.22 double Loris::Analyzer::windowWidth (void) const

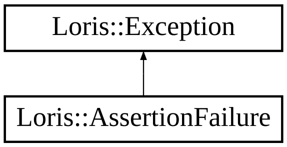
Return the frequency-domain main lobe width (measured between zero-crossings) of the analysis window used by this Analyzer.

3.4 Loris::AssertionFailure Class Reference

Class of exceptions thrown when an assertion (usually representing an invariant condition, and usually detected by the Assert macro) is violated.

`#include <Exception.h>`

Inheritance diagram for Loris::AssertionFailure::



Public Member Functions

- AssertionFailure (const std::string &str, const std::string &where="")

  string automatically using __FILE__ and __LINE__.

3.4.1 Detailed Description

Class of exceptions thrown when an assertion (usually representing an invariant condition, and usually detected by the Assert macro) is violated.

3.4.2 Constructor & Destructor Documentation

3.4.2.1 Loris::AssertionFailure::AssertionFailure (const std::string & *str*, const std::string & *where* = "")

string automatically using __FILE__ and __LINE__.

Parameters:

str is a string describing the exceptional condition

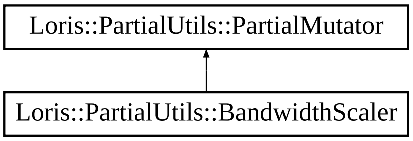
where is an option string describing the location in the source code from which the exception was thrown (generated automatically by the Throw macro).

3.5 Loris::PartialUtils::BandwidthScaler Class Reference

Scale the bandwidth of the specified Partial according to an envelope representing a time-varying bandwidth scale value.

```
#include <PartialUtils.h>
```

Inheritance diagram for Loris::PartialUtils::BandwidthScaler::


Loris::PartialUtils::PartialMutator
Loris::PartialUtils::BandwidthScaler

Public Member Functions

- BandwidthScaler (double x)

  Construct a new BandwidthScaler from a constant scale factor.

- BandwidthScaler (const Envelope &e)

  Construct a new BandwidthScaler from an Envelope representing a time-varying scale factor.

- void operator() (Partial &p) const

  Function call operator: apply a scale factor to the specified Partial.

3.5.1 Detailed Description

Scale the bandwidth of the specified Partial according to an envelope representing a time-varying bandwidth scale value.

3.6 Loris::Breakpoint Class Reference

Class Breakpoint represents a single breakpoint in the Partial parameter (frequency, amplitude, bandwidth) envelope.

```
#include <Breakpoint.h>
```

Public Member Functions

- Breakpoint (void)

  Construct a new Breakpoint with all parameters initialized to 0 (needed for STL containability).

- Breakpoint (double f, double a, double b, double p=0.)

  Construct a new Breakpoint with the specified parameters.

- double amplitude (void) const

  Return the amplitude of this Breakpoint.

- double bandwidth (void) const

  Return the bandwidth (noisiness) coefficient of this Breakpoint.

- double frequency (void) const

  Return the frequency of this Breakpoint.

- double phase (void) const

  Return the phase of this Breakpoint.

- void setAmplitude (double x)

  Set the amplitude of this Breakpoint.

- void setBandwidth (double x)

  Set the bandwidth (noisiness) coefficient of this Breakpoint.

- void setFrequency (double x)

  Set the frequency of this Breakpoint.

- void setPhase (double x)

  Set the phase of this Breakpoint.

- void addNoiseEnergy (double enoise)

    Add noise (bandwidth) energy to this Breakpoint by computing new amplitude and bandwidth values.

3.6.1 Detailed Description

Class Breakpoint represents a single breakpoint in the Partial parameter (frequency, amplitude, bandwidth) envelope.

Instantaneous phase is also stored, but is only used at the onset of a partial, or when it makes a transition from zero to nonzero amplitude.

Loris Partials represent reassigned bandwidth-enhanced model components. A Partial consists of a chain of Breakpoints describing the time-varying frequency, amplitude, and bandwidth (noisiness) of the component. For more information about Reassigned Bandwidth-Enhanced Analysis and the Reassigned Bandwidth-Enhanced Additive Sound Model, refer to the Loris website: www.cerlsoundgroup.org/Loris/.

Breakpoint is a leaf class, do not subclass.

3.6.2 Constructor & Destructor Documentation

3.6.2.1 Loris::Breakpoint::Breakpoint (double *f*, double *a*, double *b*, double *p* = 0.)

Construct a new Breakpoint with the specified parameters.

Parameters:

f is the intial frequency.

a is the initial amplitude.

b is the initial bandwidth.

p is the initial phase, if specified (if unspecified, 0 is assumed).

3.6.3 Member Function Documentation

3.6.3.1 void Loris::Breakpoint::addNoiseEnergy (double *enoise*)

Add noise (bandwidth) energy to this Breakpoint by computing new amplitude and bandwidth values.

enoise may be negative, but noise energy cannot be removed (negative energy added) in excess of the current noise energy.

Parameters:

enoise is the amount of noise energy to add to this Breakpoint.

3.6.3.2 void Loris::Breakpoint::setAmplitude (double *x*)

Set the amplitude of this Breakpoint.

Parameters:

x is the new amplitude

3.6.3.3 void Loris::Breakpoint::setBandwidth (double *x*)

Set the bandwidth (noisiness) coefficient of this Breakpoint.

Parameters:

x is the new bandwidth

3.6.3.4 void Loris::Breakpoint::setFrequency (double *x*)

Set the frequency of this Breakpoint.

Parameters:

x is the new frequency.

3.6.3.5 void Loris::Breakpoint::setPhase (double *x*)

Set the phase of this Breakpoint.

Parameters:

x is the new phase.

3.7 Loris::BreakpointEnvelope Class Reference

A BreakpointEnvelope represents a linear segment breakpoint function with infinite extension at each end (that is, evalutaing the envelope past either end of the breakpoint function yields the value at the nearest end point).

```
#include <BreakpointEnvelope.h>
```

Public Member Functions

- BreakpointEnvelope (void)

  Construct a new BreakpointEnvelope having no breakpoints (and an implicit value of 0 everywhere).

- BreakpointEnvelope (double initialValue)

  Construct and return a new BreakpointEnvelope having a single breakpoint at 0 (and an implicit value everywhere) of initialValue.

- virtual BreakpointEnvelope ∗ clone (void) const

  Return an exact copy of this BreakpointEnvelope (polymorphic copy, following the Prototype pattern).

- virtual double valueAt (double t) const

  Return the linearly-interpolated value of this BreakpointEnvelope at the specified time.

- void insert (double time, double value)

  Insert a breakpoint representing the specified (time, value) pair into this BreakpointEnvelope.

- void insertBreakpoint (double time, double value)

  Insert a breakpoint representing the specified (time, value) pair into this BreakpointEnvelope.

3.7.1 Detailed Description

A BreakpointEnvelope represents a linear segment breakpoint function with infinite extension at each end (that is, evalutaing the envelope past either end of the breakpoint function yields the value at the nearest end point).

BreakpointEnvelope implements the Envelope interface, described by the abstract class Envelope.

BreakpointEnvelope inherits the types

- `size_type`
- `value_type`
- `iterator`
- `const_iterator`

and the member functions

- size_type size(void) const
- bool empty(void) const
- iterator begin(void)
- const_iterator begin(void) const
- iterator end(void)
- const_iterator end(void) const

from std::map< double, double >.

3.7.2 Constructor & Destructor Documentation

3.7.2.1 Loris::BreakpointEnvelope::BreakpointEnvelope (void)

Construct a new BreakpointEnvelope having no breakpoints (and an implicit value of 0 everywhere).

3.7.2.2 Loris::BreakpointEnvelope::BreakpointEnvelope (double *initialValue*) `[explicit]`

Construct and return a new BreakpointEnvelope having a single breakpoint at 0 (and an implicit value everywhere) of initialValue.

Parameters:

initialValue is the value of this BreakpointEnvelope at time 0.

3.7.3 Member Function Documentation

3.7.3.1 void Loris::BreakpointEnvelope::insert (double *time*, double *value*)

Insert a breakpoint representing the specified (time, value) pair into this BreakpointEnvelope.

If there is already a breakpoint at the specified time, it will be replaced with the new breakpoint.

Parameters:

time is the time at which to insert a new breakpoint

value is the value of the new breakpoint

3.7.3.2 void Loris::BreakpointEnvelope::insertBreakpoint (double *time*, double *value*)

Insert a breakpoint representing the specified (time, value) pair into this BreakpointEnvelope.

Same as insert, retained for backwards-compatibility.

3.7.3.3 virtual double Loris::BreakpointEnvelope::valueAt (double *t*) const `[virtual]`

Return the linearly-interpolated value of this BreakpointEnvelope at the specified time.

Parameters:

t is the time at which to evaluate this BreakpointEnvelope.

3.8 Loris::Channelizer Class Reference

Class Channelizer represents an algorithm for automatic labeling of a sequence of Partials.

`#include <Channelizer.h>`

Public Member Functions

- Channelizer (const Envelope &refChanFreq, int refChanLabel)

  Exceptions:

  InvalidArgument *if refChanLabel is not positive.*

- Channelizer (const Channelizer &other)

  Construct a new Channelizer that is an exact copy of another.

- Channelizer & operator= (const Channelizer &rhs)

  Assignment operator: make this Channelizer an exact copy of another.

- ∼Channelizer (void)

  Destroy this Channelizer.

- void channelize (Partial &partial) const

  Label a Partial with the number of the frequency channel containing the greatest portion of its (the Partial's) energy.

- template<typename Iter> void channelize (Iter begin, Iter end) const

  Assign each Partial in the specified half-open (STL-style) range the label corresponding to the frequency channel containing the greatest portion of its (the Partial's) energy.

- template<typename Iter> void operator() (Iter begin, Iter end) const

  Function call operator: same as channelize().

Static Public Member Functions

- template<typename Iter> void channelize (Iter begin, Iter end, const Envelope &refChanFreq, int refChanLabel)

  Static member that constructs an instance and applies it to a sequence of Partials.

3.8.1 Detailed Description

Class Channelizer represents an algorithm for automatic labeling of a sequence of Partials.

Partials must be labeled in preparation for morphing (see Morpher) to establish correspondences between Partials in the morph source and target sounds.

Channelized partials are labeled according to their adherence to a harmonic frequency structure with a time-varying fundamental frequency. The frequency spectrum is partitioned into non-overlapping channels having time-varying center frequencies that are harmonic (integer) multiples of a specified reference frequency envelope, and each channel is identified by a unique label equal to its harmonic number. Each Partial is assigned the label corresponding to the channel containing the greatest portion of its (the Partial's) energy.

A reference frequency Envelope for channelization and the channel number to which it corresponds (1 for an Envelope that tracks the Partial at the fundamental frequency) must be specified. The reference Envelope can be constructed explcitly, point by point (using, for example, the BreakpointEnvelope class), or constructed automatically using the FrequencyReference class.

Channelizer is a leaf class, do not subclass.

3.8.2 Constructor & Destructor Documentation

3.8.2.1 Loris::Channelizer::Channelizer (const Envelope & *refChanFreq*, int *refChanLabel*)

Exceptions:

InvalidArgument if refChanLabel is not positive.

Parameters:

refChanFreq is an Envelope representing the center frequency of a channel.

refChanLabel is the corresponding channel number (i.e. 1 if refChanFreq is the lowest-frequency channel, and all other channels are harmonics of refChanFreq, or 2 if refChanFreq tracks the second harmonic, etc.).

3.8.2.2 Loris::Channelizer::Channelizer (const Channelizer & *other*)

Construct a new Channelizer that is an exact copy of another.

The copy represents the same set of frequency channels, constructed from the same reference Envelope and channel number.

Parameters:

other is the Channelizer to copy

3.8.3 Member Function Documentation

3.8.3.1 template<typename Iter> void Loris::Channelizer::channelize (Iter *begin*, Iter *end*, const Envelope & *refChanFreq*, int *refChanLabel*) `[static]`

Static member that constructs an instance and applies it to a sequence of Partials.

Construct a Channelizer using the specified Envelope and reference label, and use it to channelize a sequence of Partials.

Parameters:

begin is the beginning of a sequence of Partials to channelize.

end is the end of a sequence of Partials to channelize.

refChanFreq is an Envelope representing the center frequency of a channel.

refChanLabel is the corresponding channel number (i.e. 1 if refChanFreq is the lowest-frequency channel, and all other channels are harmonics of refChanFreq, or 2 if refChanFreq tracks the second harmonic, etc.).

Exceptions:

InvalidArgument if refChanLabel is not positive.

If compiled with NO_TEMPLATE_MEMBERS defined, then begin and end must be PartialList::iterators, otherwise they can be any type of iterators over a sequence of Partials.

3.8.3.2 template<typename Iter> void Loris::Channelizer::channelize (Iter *begin*, Iter *end*) const

Assign each Partial in the specified half-open (STL-style) range the label corresponding to the frequency channel containing the greatest portion of its (the Partial's) energy.

Parameters:

begin is the beginning of the range of Partials to channelize

end is (one-past) the end of the range of Partials to channelize

If compiled with NO_TEMPLATE_MEMBERS defined, then begin and end must be PartialList::iterators, otherwise they can be any type of iterators over a sequence of Partials.

3.8.3.3 void Loris::Channelizer::channelize (Partial & *partial*) const

Label a Partial with the number of the frequency channel containing the greatest portion of its (the Partial's) energy.

Parameters:

partial is the Partial to label.

3.8.3.4 Channelizer& Loris::Channelizer::operator= (const Channelizer & *rhs*)

Assignment operator: make this Channelizer an exact copy of another.

This Channelizer is made to represent the same set of frequency channels, constructed from the same reference Envelope and channel number as rhs.

Parameters:

rhs is the Channelizer to copy

3.9 Loris::PartialUtils::Cropper Class Reference

Trim a Partial by removing Breakpoints outside a specified time span.

`#include <PartialUtils.h>`

3.9.1 Detailed Description

Trim a Partial by removing Breakpoints outside a specified time span.

Insert a Breakpoint at the boundary when cropping occurs.

3.10 Loris::Dilator Class Reference

Class Dilator represents an algorithm for non-uniformly expanding and contracting the Partial parameter envelopes according to the initial and target (desired) times of temporal features.

```
#include <Dilator.h>
```

Public Member Functions

- Dilator (void)

  Construct a new Dilator with no time points.

- template<typename Iter1, typename Iter2> Dilator (Iter1 ibegin, Iter1 iend, Iter2 tbegin)

  Construct a new Dilator using a range of initial time points and a range of target (desired) time points.

- void insert (double i, double t)

  Insert a pair of initial and target time points.

- void dilate (Partial &p) const

  Replace the Partial envelope with a new envelope having the same Breakpoints at times computed to align temporal features in the sorted sequence of initial time points with their counterparts the sorted sequence of target time points.

- void operator() (Partial &p) const

  Function call operator: same as dilate(Partial & p).

- void dilate (Marker &m) const

  Compute a new time for the specified Marker using warpTime(), exactly as Partial Breakpoint times are recomputed.

- void operator() (Marker &m) const

  Function call operator: same as dilate(Marker & m).

- template<typename Iter> void dilate (Iter dilate_begin, Iter dilate_end) const

  Non-uniformly expand and contract the parameter envelopes of the each Partial in the specified half-open range according to this Dilator's stored initial and target (desired) times.

- template<typename Iter> void operator() (Iter dilate_begin, Iter dilate_end) const

  Function call operator: same as dilate(Iter dilate_begin, Iter dilate_end).

- double warpTime (double currentTime) const

  Return the dilated time value corresponding to the specified initial time.

Static Public Member Functions

- template<typename PartialsIter, typename TimeIter1, typename TimeIter2> void dilate (PartialsIter dilate_begin, PartialsIter dilate_end, TimeIter1 ibegin, TimeIter1 iend, TimeIter2 tbegin)

  Static member that constructs an instance and applies it to a sequence of Partials.

3.10.1 Detailed Description

Class Dilator represents an algorithm for non-uniformly expanding and contracting the Partial parameter envelopes according to the initial and target (desired) times of temporal features.

It is frequently necessary to redistribute temporal events in this way in preparation for a sound morph. For example, when morphing instrument tones, it is common to align the attack, sustain, and release portions of the source sounds by dilating or contracting those temporal regions.

This same procedure can be applied to the Markers stored in AiffFile, SdifFile, and SpcFile (see Marker.h).

3.10.2 Constructor & Destructor Documentation

3.10.2.1 template<typename Iter1, typename Iter2> Loris::Dilator::Dilator (Iter1 *ibegin*, Iter1 *iend*, Iter2 *tbegin*)

Construct a new Dilator using a range of initial time points and a range of target (desired) time points.

The client must ensure that the target range has at least as many elements as the initial range.

Parameters:

ibegin is the beginning of a sequence of initial, or source, time points.

iend is (one-past) the end of a sequence of initial, or source, time points.

tbegin is the beginning of a sequence of target time points; this sequence must be as long as the sequence of initial time point described by ibegin and iend.

If compiled with NO_TEMPLATE_MEMBERS defined, this member accepts only const double $*$ arguments.

3.10.3 Member Function Documentation

3.10.3.1 template<typename PartialsIter, typename TimeIter1, typename TimeIter2> void Loris::Dilator::dilate (PartialsIter *dilate_begin*, PartialsIter *dilate_end*, TimeIter1 *ibegin*, TimeIter1 *iend*, TimeIter2 *tbegin*) `[static]`

Static member that constructs an instance and applies it to a sequence of Partials.

Parameters:

dilate_begin is the beginning of a sequence of Partials to dilate.

dilate_end is (one-past) the end of a sequence of Partials to dilate.

ibegin is the beginning of a sequence of initial, or source, time points.

iend is (one-past) the end of a sequence of initial, or source, time points.

tbegin is the beginning of a sequence of target time points; this sequence must be as long as the sequence of initial time point described by ibegin and iend.

If compiled with NO_TEMPLATE_MEMBERS defined, this member accepts only PartialList::const_iterator arguments. Otherwise, this member also works for sequences of Markers. If compiled with NO_TEMPLATE_MEMBERS defined, this member accepts only const double $*$ arguments for the times, otherwise, any iterator will do..

See also:

Dilator::dilate(Partial & p) const
Dilator::dilate(Marker & m) const

3.10.3.2 template<typename Iter> void Loris::Dilator::dilate (Iter *dilate_begin*, Iter *dilate_end*) const

Non-uniformly expand and contract the parameter envelopes of the each Partial in the specified half-open range according to this Dilator's stored initial and target (desired) times.

Parameters:

dilate_begin is the beginning of a sequence of Partials to dilate.

dilate_end is (one-past) the end of a sequence of Partials to dilate.

If compiled with NO_TEMPLATE_MEMBERS defined, this member accepts only PartialList::const_iterator arguments. Otherwise, this member also works for sequences of Markers.

See also:

Dilator::dilate(Partial & p) const
Dilator::dilate(Marker & m) const

3.10.3.3 void Loris::Dilator::dilate (Marker & *m*) const

Compute a new time for the specified Marker using warpTime(), exactly as Partial Breakpoint times are recomputed.

This can be used to dilate the Markers corresponding to a collection of Partials.

Parameters:

m is the Marker whose time should be recomputed.

3.10.3.4 void Loris::Dilator::dilate (Partial & *p*) const

Replace the Partial envelope with a new envelope having the same Breakpoints at times computed to align temporal features in the sorted sequence of initial time points with their counterparts the sorted sequence of target time points.

Depending on the specification of initial and target time points, the dilated Partial may have Breakpoints at times less than 0, even if the original Partial did not.

It is possible to have duplicate time points in either sequence. Duplicate initial time points result in very localized stretching. Duplicate target time points result in very localized compression.

If all initial time points are greater than 0, then an implicit time point at 0 is assumed in both initial and target sequences, so the onset of a sound can be stretched without explcitly specifying a zero point in each vector. (This seems most intuitive, and only looks like an inconsistency if clients are using negative time points in their Dilator, or Partials having Breakpoints before time 0, both of which are probably unusual circumstances.)

Parameters:

p is the Partial to dilate.

3.10.3.5 void Loris::Dilator::insert (double *i*, double *t*)

Insert a pair of initial and target time points.

Specify a pair of initial and target time points to be used by this Dilator, corresponding, for example, to the initial and desired time of a particular temporal feature in an analyzed sound.

Parameters:

i is an initial, or source, time point

t is a target time point

The time points will be sorted before they are used. If, in the sequences of initial and target time points, there are exactly the same number of initial time points preceding i as target time points preceding t, then time i will be warped to time t in the dilation process.

3.10.3.6 template<typename Iter> void Loris::Dilator::operator() (Iter *dilate_begin*, Iter *dilate_end*) const

Function call operator: same as dilate(Iter dilate_begin, Iter dilate_end).

If compiled with NO_TEMPLATE_MEMBERS defined, this member accepts only PartialList::const_iterator arguments. Otherwise, this member also works for sequences of Markers.

See also:

Dilator::dilate(Partial & p) const
Dilator::dilate(Marker & m) const

3.10.3.7 void Loris::Dilator::operator() (Marker & *m*) const

Function call operator: same as dilate(Marker & m).

See also:

Dilator::dilate(Marker & m) const

3.10.3.8 void Loris::Dilator::operator() (Partial & *p*) const

Function call operator: same as dilate(Partial & p).

See also:

Dilator::dilate(Partial & p) const

3.10.3.9 double Loris::Dilator::warpTime (double *currentTime*) const

Return the dilated time value corresponding to the specified initial time.

Parameters:

currentTime is a pre-dilated time.

Returns:

the dilated time corresponding to the initial time currentTime

3.11 Loris::Distiller Class Reference

Class Distiller represents an algorithm for "distilling" a group of Partials that logically represent a single component into a single Partial.

```
#include <Distiller.h>
```

Public Member Functions

- Distiller (double partialFadeTime=0.001, double partialSilentTime=0.0001)

  Construct a new Distiller using the specified fade time for gaps between Partials.

- template<typename Container> Container::iterator distill (Container &partials)

  Distill labeled Partials in a collection leaving only a single Partial per non-zero label.

- template<typename Container> Container::iterator operator() (Container &partials)

  Function call operator: same as distill(PartialList & partials).

Static Public Member Functions

- template<typename Container> Container::iterator distill (Container &partials, double partialFadeTime, double partialSilentTime=0.0001)

  Static member that constructs an instance and applies it to a sequence of Partials.

3.11.1 Detailed Description

Class Distiller represents an algorithm for "distilling" a group of Partials that logically represent a single component into a single Partial.

The sound morphing algorithm in Loris requires that Partials in a given source be labeled uniquely, that is, no two Partials can have the same label. The Distiller enforces this condition. All Partials identified with a particular frequency channel (see Channelizer), and, therefore, having a common label, are distilled into a single Partial, leaving at most a single Partial per frequency channel and label. Channels that contain no Partials are not represented in the distilled data. Partials that are not labeled, that is, Partials having label 0, are are "collated " into groups of non-overlapping (in time)

Partials, assigned an unused label (greater than the label associated with any frequency channel), and fused into a single Partial per group. "Collating" is a bit like "sifting" but non-overlapping Partials are grouped without regard to frequency proximity. This algorithm produces the smallest-possible number of collated Partials. Thanks to Ulrike Axen for providing this optimal algorithm.

Distillation modifies the Partial container (a PartialList). All Partials in the distilled range having a common label are replaced by a single Partial in the distillation process.

3.11.2 Constructor & Destructor Documentation

3.11.2.1 Loris::Distiller::Distiller (double *partialFadeTime* = 0.001, double *partialSilentTime* = 0.0001) `[explicit]`

Construct a new Distiller using the specified fade time for gaps between Partials.

When two non-overlapping Partials are distilled into a single Partial, the distilled Partial fades out at the end of the earlier Partial and back in again at the onset of the later one. The fade time is the time over which these fades occur. By default, use a 1 ms fade time. The gap time is the additional time over which a Partial faded out must remain at zero amplitude before it can fade back in. By default, use a gap time of one tenth of a millisecond, to prevent a pair of arbitrarily close null Breakpoints being inserted.

Parameters:

- ***partialFadeTime*** is the time (in seconds) over which Partials joined by distillation fade to and from zero amplitude. Default is 0.001 (one millisecond).
- ***partialSilentTime*** is the minimum duration (in seconds) of the silent (zero-amplitude) gap between two Partials joined by distillation. (Default is 0.0001 (one tenth of a millisecond).

3.11.3 Member Function Documentation

3.11.3.1 template<typename Container> Container::iterator Loris::Distiller::distill (Container & *partials*, double *partialFadeTime*, double *partialSilentTime* = 0.0001) `[static]`

Static member that constructs an instance and applies it to a sequence of Partials.

Construct a Distiller using default parameters, and use it to distill a sequence of Partials.

Postcondition:

All Partials in the collection are uniquely-labeled

Parameters:

partials is the collection of Partials to distill in-place

partialFadeTime is the time (in seconds) over which Partials joined by distillation fade to and from zero amplitude.

partialSilentTime is the minimum duration (in seconds) of the silent (zero-amplitude) gap between two Partials joined by distillation. (Default is 0.0001 (one tenth of a millisecond).

Returns:

the position of the end of the range of distilled Partials, which is either the end of the collection, oor the position of the first collated Partial, composed of unlabeled Partials in the original collection.

If compiled with NO_TEMPLATE_MEMBERS defined, then partials must be a PartialList, otherwise it can be any container type storing Partials that supports at least bidirectional iterators.

3.11.3.2 template<typename Container> Container::iterator Loris::Distiller::distill (Container & *partials*)

Distill labeled Partials in a collection leaving only a single Partial per non-zero label.

See also:

Distiller::distill(Container & partials)

3.11.3.3 template<typename Container> Container::iterator Loris::Distiller::operator() (Container & *partials*)

Function call operator: same as distill(PartialList & partials).

See also:

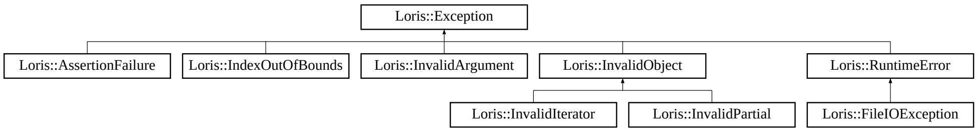
Distiller::distill(Container & partials)

3.12 Loris::Exception Class Reference

Exception is a generic exception class for reporting exceptional circumstances in Loris.

`#include <Exception.h>`

Inheritance diagram for Loris::Exception::



Public Member Functions

- Exception (const std::string &str, const std::string &where="")

  string automatically using __FILE__ and __LINE__.

- virtual ∼Exception (void) throw ()

  Destroy this Exception.

- const char ∗ what (void) const throw ()

  C-style string (char pointer).

- Exception & append (const std::string &str)

  Append the specified string to this Exception's description, and return a reference to this Exception.

- const std::string & str (void) const

  Return a read-only refernce to this Exception's description string.

Protected Attributes

- std::string _sbuf

  string for storing the exception description

3.12.1 Detailed Description

Exception is a generic exception class for reporting exceptional circumstances in Loris.

Exception is derived from std:exception, and is the base for a hierarchy of derived exception classes in Loris.

3.12.2 Constructor & Destructor Documentation

3.12.2.1 Loris::Exception::Exception (const std::string & *str*, const std::string & *where* = "")

string automatically using __FILE__ and __LINE__.

Parameters:

str is a string describing the exceptional condition

where is an option string describing the location in the source code from which the exception was thrown (generated automatically by the Throw macro).

3.12.3 Member Function Documentation

3.12.3.1 Exception& Loris::Exception::append (const std::string & *str*)

Append the specified string to this Exception's description, and return a reference to this Exception.

Parameters:

str is text to append to the exception description

Returns:

a reference to this Exception.

3.12.3.2 const std::string& Loris::Exception::str (void) const

Return a read-only refernce to this Exception's description string.

Returns:

a string describing the exceptional condition

3.12.3.3 const char* Loris::Exception::what (void) const throw ()

C-style string (char pointer).

Overrides std::exception::what.

Returns:

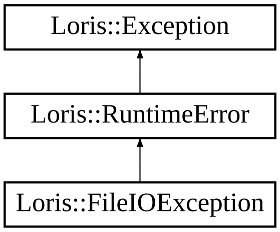
a C-style string describing the exceptional condition.

3.13 Loris::FileIOException Class Reference

Class of exceptions thrown when file input or output fails.

`#include <Exception.h>`

Inheritance diagram for Loris::FileIOException::



Public Member Functions

- FileIOException (const std::string &str, const std::string &where="")

  string automatically using __FILE__ and __LINE__.

3.13.1 Detailed Description

Class of exceptions thrown when file input or output fails.

3.13.2 Constructor & Destructor Documentation

3.13.2.1 Loris::FileIOException::FileIOException (const std::string & *str*, const std::string & *where* = "")

string automatically using __FILE__ and __LINE__.

Parameters:

str is a string describing the exceptional condition

where is an option string describing the location in the source code from which the exception was thrown (generated automatically by the Throw macro).

3.14 Loris::FourierTransform Class Reference

FourierTransform provides a simplified interface to the FFTW library (www.fftw.org).

`#include <FourierTransform.h>`

Public Types

- typedef std::vector< std::complex< double > >::size_type size_type

  An unsigned integral type large enough to represent the length of any transform.

- typedef std::vector< std::complex< double > >::iterator iterator

  The type of a non-const iterator of (complex) transform samples.

- typedef std::vector< std::complex< double > >::const_iterator const_iterator

  The type of a const iterator of (complex) transform samples.

Public Member Functions

- FourierTransform (size_type len)

  Exceptions:

  RuntimeError *if the necessary buffers cannot be allocated, or there is an error configuring FFTW.*

- FourierTransform (const FourierTransform &rhs)

  Initialize a new FourierTransform that is a copy of another, having the same size and the same buffer contents.

- ∼FourierTransform (void)

  Free the resources associated with this FourierTransform.

- FourierTransform & operator= (const FourierTransform &rhs)

  Make this FourierTransform a copy of another, having the same size and buffer contents.

- std::complex< double > & operator[ ] (size_type index)

  Access (read/write) a transform sample by index.

- const std::complex< double > & operator[ ] (size_type index) const

Access (read-only) a transform sample by index.

- iterator begin (void)

  Return an iterator refering to the beginning of the sequence of complex samples in the transform buffer.

- iterator end (void)

  complex samples in the transform buffer.

- const_iterator begin (void) const

  Return a const iterator refering to the beginning of the sequence of complex samples in the transform buffer.

- const_iterator end (void) const

  complex samples in the transform buffer.

- void transform (void)

  Compute the Fourier transform of the samples stored in the transform buffer.

- size_type size (void) const

  Return the length of the transform (in samples).

3.14.1 Detailed Description

FourierTransform provides a simplified interface to the FFTW library (www.fftw.org).

Loris uses the FFTW library to perform efficient Fourier transforms of arbitrary length. Clients store and access the in-place transform data as a sequence of std::complex< double >. Samples are stored in the FourierTransform instance using subscript or iterator access, the transform is computed by the transform member, and the transformed samples replace the input samples, and are accessed by subscript or iterator. FourierTransform computes a complex transform, so it can be used to invert a transform of real samples as well. Uses the standard library complex class, which implements arithmetic operations. Does not use FFTW "wisdom" to speed up transform computation.

3.14.2 Constructor & Destructor Documentation

3.14.2.1 Loris::FourierTransform::FourierTransform (const FourierTransform & *rhs*)

Initialize a new FourierTransform that is a copy of another, having the same size and the same buffer contents.

Parameters:

rhs is the instance to copy

Exceptions:

RuntimeError if the necessary buffers cannot be allocated, or there is an error configuring FFTW.

3.14.3 Member Function Documentation

3.14.3.1 const_iterator Loris::FourierTransform::begin (void) const

Return a const iterator refering to the beginning of the sequence of complex samples in the transform buffer.

Returns:

a const iterator refering to the first position in the transform buffer.

3.14.3.2 iterator Loris::FourierTransform::begin (void)

Return an iterator refering to the beginning of the sequence of complex samples in the transform buffer.

Returns:

a non-const iterator refering to the first position in the transform buffer.

3.14.3.3 const_iterator Loris::FourierTransform::end (void) const

complex samples in the transform buffer.

Returns:

a const iterator refering to one past the last position in the transform buffer.

3.14.3.4 iterator Loris::FourierTransform::end (void)

complex samples in the transform buffer.

Returns:

a non-const iterator refering to one past the last position in the transform buffer.

3.14.3.5 FourierTransform& Loris::FourierTransform::operator= (const FourierTransform & *rhs*)

Make this FourierTransform a copy of another, having the same size and buffer contents.

Parameters:

rhs is the instance to copy

Returns:

a refernce to this instance

Exceptions:

RuntimeError if the necessary buffers cannot be allocated, or there is an error configuring FFTW.

3.14.3.6]

const std::complex< double >& Loris::FourierTransform::operator[ ] (size_type *index*) const

Access (read-only) a transform sample by index.

Use this member to fill the transform buffer before computing the transform, and to access the samples after computing the transform. (inlined for speed)

Parameters:

index is the index or rank of the complex transform sample to access. Zero is the first position in the buffer.

Returns:

const reference to the std::complex< double > at the specified position in the buffer.

3.14.3.7]

std::complex< double >& Loris::FourierTransform::operator[ ] (size_type *index*)

Access (read/write) a transform sample by index.

Use this member to fill the transform buffer before computing the transform, and to access the samples after computing the transform. (inlined for speed)

Parameters:

index is the index or rank of the complex transform sample to access. Zero is the first position in the buffer.

Returns:

non-const reference to the std::complex< double > at the specified position in the buffer.

3.14.3.8 size_type Loris::FourierTransform::size (void) const

Return the length of the transform (in samples).

Returns:

the length of the transform in samples.

3.14.3.9 void Loris::FourierTransform::transform (void)

Compute the Fourier transform of the samples stored in the transform buffer.

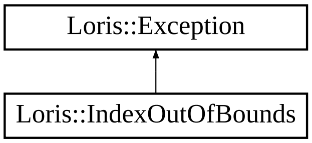
The samples stored in the transform buffer (accessed by index or by iterator) are replaced by the transformed samples, in-place.

3.15 Loris::IndexOutOfBounds Class Reference

Class of exceptions thrown when a subscriptable object is accessed with an index that is out of range.

`#include <Exception.h>`

Inheritance diagram for Loris::IndexOutOfBounds::



Public Member Functions

- IndexOutOfBounds (const std::string &str, const std::string &where="")

  string automatically using __FILE__ and __LINE__.

3.15.1 Detailed Description

Class of exceptions thrown when a subscriptable object is accessed with an index that is out of range.

3.15.2 Constructor & Destructor Documentation

3.15.2.1 Loris::IndexOutOfBounds::IndexOutOfBounds (const std::string & *str*, const std::string & *where* = "")

string automatically using __FILE__ and __LINE__.

Parameters:

str is a string describing the exceptional condition

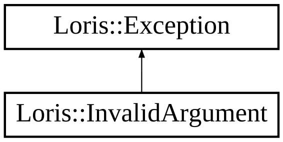
where is an option string describing the location in the source code from which the exception was thrown (generated automatically by the Throw macro).

3.16 Loris::InvalidArgument Class Reference

Class of exceptions thrown when a function argument is found to be invalid.

```
#include <Exception.h>
```

Inheritance diagram for Loris::InvalidArgument::



Public Member Functions

- InvalidArgument (const std::string &str, const std::string &where="")

  string automatically using __FILE__ and __LINE__.

3.16.1 Detailed Description

Class of exceptions thrown when a function argument is found to be invalid.

3.16.2 Constructor & Destructor Documentation

3.16.2.1 Loris::InvalidArgument::InvalidArgument (const std::string & *str*, const std::string & *where* = "")

string automatically using __FILE__ and __LINE__.

Parameters:

str is a string describing the exceptional condition

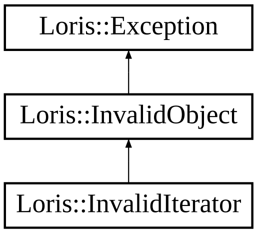
where is an option string describing the location in the source code from which the exception was thrown (generated automatically by the Throw macro).

3.17 Loris::InvalidIterator Class Reference

Class of exceptions thrown when an Iterator is found to be badly configured or otherwise invalid.

`#include <Exception.h>`

Inheritance diagram for Loris::InvalidIterator::



Public Member Functions

- InvalidIterator (const std::string &str, const std::string &where="")
  string automatically using __FILE__ and __LINE__.

3.17.1 Detailed Description

Class of exceptions thrown when an Iterator is found to be badly configured or otherwise invalid.

3.17.2 Constructor & Destructor Documentation

3.17.2.1 Loris::InvalidIterator::InvalidIterator (const std::string & *str*, const std::string & *where* = "")

string automatically using __FILE__ and __LINE__.

Parameters:

str is a string describing the exceptional condition

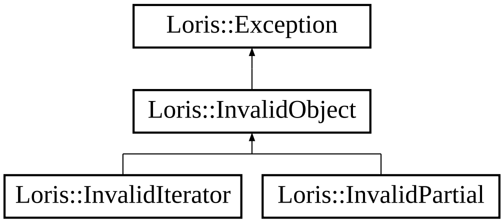
where is an option string describing the location in the source code from which the exception was thrown (generated automatically by the Throw macro).

3.18 Loris::InvalidObject Class Reference

Class of exceptions thrown when an object is found to be badly configured or otherwise invalid.

`#include <Exception.h>`

Inheritance diagram for Loris::InvalidObject::



Public Member Functions

- InvalidObject (const std::string &str, const std::string &where="")

  string automatically using __FILE__ and __LINE__.

3.18.1 Detailed Description

Class of exceptions thrown when an object is found to be badly configured or otherwise invalid.

3.18.2 Constructor & Destructor Documentation

3.18.2.1 Loris::InvalidObject::InvalidObject (const std::string & *str*, const std::string & *where* = "")

string automatically using __FILE__ and __LINE__.

Parameters:

str is a string describing the exceptional condition

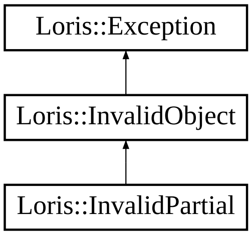
where is an option string describing the location in the source code from which the exception was thrown (generated automatically by the Throw macro).

3.19 Loris::InvalidPartial Class Reference

Class of exceptions thrown when a Partial is found to be badly configured or otherwise invalid.

`#include <Partial.h>`

Inheritance diagram for Loris::InvalidPartial::



Public Member Functions

- InvalidPartial (const std::string &str, const std::string &where="")

  string automatically using __FILE__ and __LINE__.

3.19.1 Detailed Description

Class of exceptions thrown when a Partial is found to be badly configured or otherwise invalid.

3.19.2 Constructor & Destructor Documentation

3.19.2.1 Loris::InvalidPartial::InvalidPartial (const std::string & *str*, const std::string & *where* = "")

string automatically using __FILE__ and __LINE__.

Parameters:

str is a string describing the exceptional condition

where is an option string describing the location in the source code from which the exception was thrown (generated automatically byt he Throw macro).

3.20 Loris::Morpher Class Reference

Class Morpher performs sound morphing and Partial parameter envelope interpolation according to a trio of frequency, amplitude, and bandwidth morphing functions, described by Envelopes.

`#include <Morpher.h>`

Public Member Functions

- Morpher (const Envelope &f)

  Construct a new Morpher using the same morphing envelope for frequency, amplitude, and bandwidth (noisiness).

- Morpher (const Envelope &ff, const Envelope &af, const Envelope &bwf)

  Construct a new Morpher using the specified morphing envelopes for frequency, amplitude, and bandwidth (noisiness).

- Morpher (const Morpher &rhs)

  Construct a new Morpher that is a duplicate of rhs.

- ∼Morpher (void)

  Destroy this Morpher.

- Morpher & operator= (const Morpher &rhs)
- Partial morphPartial (const Partial &src, const Partial &tgt, int assignLabel)

  Morph a pair of Partials to yield a new morphed Partial.

- void morph (PartialList::const_iterator beginSrc, PartialList::const_iterator endSrc, PartialList::const_iterator beginTgt, PartialList::const_iterator endTgt)

  Morph two sounds (collections of Partials labeled to indicate correspondences) into a single labeled collection of Partials.

- void crossfade (PartialList::const_iterator beginSrc, PartialList::const_iterator endSrc, PartialList::const_iterator beginTgt, PartialList::const_iterator endTgt, Partial::label_type label=0)

  Crossfade Partials with no correspondences.

- Breakpoint morphBreakpoints (const Breakpoint &srcBkpt, const Breakpoint &tgtBkpt, double time) const

Compute morphed parameter values at the specified time, using the source and target Breakpoints (assumed to correspond exactly to the specified time).

- Breakpoint morphSrcBreakpoint (const Breakpoint &bp, const Partial &tgtPartial, double time) const

  Compute morphed parameter values at the specified time, using the source Breakpoint (assumed to correspond exactly to the specified time) and the target Partial (whose parameters are examined at the specified time).

- Breakpoint morphTgtBreakpoint (const Breakpoint &bp, const Partial &tgtPartial, double time) const

  Compute morphed parameter values at the specified time, using the target Breakpoint (assumed to correspond exactly to the specified time) and the source Partial (whose parameters are examined at the specified time).

- Breakpoint fadeSrcBreakpoint (Breakpoint bp, double time) const

  Compute morphed parameter values at the specified time, using the source Breakpoint, assumed to correspond exactly to the specified time, and assuming that there is no corresponding target Partial, so the source Breakpoint should be simply faded.

- Breakpoint fadeTgtBreakpoint (Breakpoint bp, double time) const

  Compute morphed parameter values at the specified time, using the target Breakpoint, assumed to correspond exactly to the specified time, and assuming that there is not corresponding source Partial, so the target Breakpoint should be simply faded.

- void setFrequencyFunction (const Envelope &f)

  Assign a new frequency morphing envelope to this Morpher.

- void setAmplitudeFunction (const Envelope &f)

  Assign a new amplitude morphing envelope to this Morpher.

- void setBandwidthFunction (const Envelope &f)

  Assign a new bandwidth morphing envelope to this Morpher.

- const Envelope & frequencyFunction (void) const

  Return a reference to this Morpher's frequency morphing envelope.

- const Envelope & amplitudeFunction (void) const

  Return a reference to this Morpher's amplitude morphing envelope.

- const Envelope & bandwidthFunction (void) const

  Return a reference to this Morpher's bandwidth morphing envelope.

- double amplitudeShape (void) const

  Return the shaping parameter for the amplitude moprhing function (only used in new log-amplitude morphing).

- void setAmplitudeShape (double x)

  Set the shaping parameter for the amplitude moprhing function (only used in new log-amplitude morphing).

- double minBreakpointGap (void) const

  Return the minimum time gap (secs) between two Breakpoints in the morphed Partials.

- void setMinBreakpointGap (double x)

  Set the minimum time gap (secs) between two Breakpoints in the morphed Partials.

- Partial::label_type sourceReferenceLabel (void) const

  Return the label of the Partial to be used as a reference Partial for the source sequence in a morph of two Partial sequences.

- Partial::label_type targetReferenceLabel (void) const

  Return the label of the Partial to be used as a reference Partial for the target sequence in a morph of two Partial sequences.

- void setSourceReferenceLabel (Partial::label_type l)

  Set the label of the Partial to be used as a reference Partial for the source sequence in a morph of two Partial sequences.

- void setTargetReferenceLabel (Partial::label_type l)

  Set the label of the Partial to be used as a reference Partial for the target sequence in a morph of two Partial sequences.

- PartialList & partials (void)

  Return a reference to this Morpher's list of morphed Partials.

- const PartialList & partials (void) const

  Return a const reference to this Morpher's list of morphed Partials.

3.20.1 Detailed Description

Class Morpher performs sound morphing and Partial parameter envelope interpolation according to a trio of frequency, amplitude, and bandwidth morphing functions, described by Envelopes.

Sound morphing is achieved by interpolating the time-varying frequencies, amplitudes, and bandwidths of corresponding partials obtained from reassigned bandwidth-enhanced analysis of the source and target sounds. Partial correspondences may be established by labeling, using instances of the Channelizer and Distiller classes.

The Morpher collects morphed Partials in a PartialList, that is accessible to clients.

For more information about sound morphing using the Reassigned Bandwidth-Enhanced Additive Sound Model, refer to the Loris website: www.cerlsoundgroup.org/Loris/.

Morpher is a leaf class, do not subclass.

3.20.2 Constructor & Destructor Documentation

3.20.2.1 Loris::Morpher::Morpher (const Envelope & *f*)

Construct a new Morpher using the same morphing envelope for frequency, amplitude, and bandwidth (noisiness).

Parameters:

f is the Envelope to clone for all three morphing functions.

3.20.2.2 Loris::Morpher::Morpher (const Envelope & *ff*, const Envelope & *af*, const Envelope & *bwf*)

Construct a new Morpher using the specified morphing envelopes for frequency, amplitude, and bandwidth (noisiness).

Parameters:

ff is the Envelope to clone for the frequency morphing function

af is the Envelope to clone for the amplitude morphing function

bwf is the Envelope to clone for the bandwidth morphing function

3.20.2.3 Loris::Morpher::Morpher (const Morpher & *rhs*)

Construct a new Morpher that is a duplicate of rhs.

Parameters:

rhs is the Morpher to duplicate

3.20.3 Member Function Documentation

3.20.3.1 double Loris::Morpher::amplitudeShape (void) const

Return the shaping parameter for the amplitude moprhing function (only used in new log-amplitude morphing).

This shaping parameter controls the slope of the amplitude morphing function, for values greater than 1, this function gets nearly linear (like the old amplitude morphing function), for values much less than 1 (e.g. 1E-5) the slope is gently curved and sounds pretty "linear", for very small values (e.g. 1E-12) the curve is very steep and sounds un-natural because of the huge jump from zero amplitude to very small amplitude.

3.20.3.2 void Loris::Morpher::crossfade (PartialList::const_iterator *beginSrc*, PartialList::const_iterator *endSrc*, PartialList::const_iterator *beginTgt*, PartialList::const_iterator *endTgt*, Partial::label_type *label* = 0)

Crossfade Partials with no correspondences.

Unlabeled Partials (having the specified label) are considered to have no correspondences, so they are just faded out, and not actually morphed. Consistent with the morphing behavior, crossfaded Partials are thinned, if necssary, so that no two Breakpoints are closer in time than the minBreakpointGap.

The Partials in the first range are treated as components of the source sound, corresponding to a morph function value of 0, and those in the second are treated as components of the target sound, corresponding to a morph function value of 1.

The crossfaded Partials are stored in the Morpher's PartialList.

Parameters:

beginSrc is the beginning of the sequence of Partials corresponding to a morph function value of 0.

endSrc is (one past) the end of the sequence of Partials corresponding to a morph function value of 0.

beginTgt is the beginning of the sequence of Partials corresponding to a morph function value of 1.

endTgt is (one past) the end of the sequence of Partials corresponding to a morph function value of 1.

label is the label to associate with unlabeled Partials (default is 0).

3.20.3.3 Breakpoint Loris::Morpher::fadeSrcBreakpoint (Breakpoint *bp*, double *time*) const

Compute morphed parameter values at the specified time, using the source Breakpoint, assumed to correspond exactly to the specified time, and assuming that there is no corresponding target Partial, so the source Breakpoint should be simply faded.

Parameters:

bp is the Breakpoint corresponding to a morph function value of 0.

time is the time corresponding to bp (used to evaluate the morphing functions).

Returns:

the faded Breakpoint

3.20.3.4 Breakpoint Loris::Morpher::fadeTgtBreakpoint (Breakpoint *bp*, double *time*) const

Compute morphed parameter values at the specified time, using the target Breakpoint, assumed to correspond exactly to the specified time, and assuming that there is not corresponding source Partial, so the target Breakpoint should be simply faded.

Parameters:

bp is the Breakpoint corresponding to a morph function value of 1.

time is the time corresponding to bp (used to evaluate the morphing functions).

Returns:

the faded Breakpoint

3.20.3.5 double Loris::Morpher::minBreakpointGap (void) const

Return the minimum time gap (secs) between two Breakpoints in the morphed Partials.

Morphing two Partials can generate a third Partial having Breakpoints arbitrarily close together in time, and this makes morphs huge. Raising this threshold limits the Breakpoint density in the morphed Partials. Default is 1/10 ms.

3.20.3.6 void Loris::Morpher::morph (PartialList::const_iterator *beginSrc*, PartialList::const_iterator *endSrc*, PartialList::const_iterator *beginTgt*, PartialList::const_iterator *endTgt*)

Morph two sounds (collections of Partials labeled to indicate correspondences) into a single labeled collection of Partials.

Unlabeled Partials (having label 0) are crossfaded. The morphed and crossfaded Partials are stored in the Morpher's PartialList.

The Partials in the first range are treated as components of the source sound, corresponding to a morph function value of 0, and those in the second are treated as components of the target sound, corresponding to a morph function value of 1.

See also:

crossfade, morphPartial

Parameters:

beginSrc is the beginning of the sequence of Partials corresponding to a morph function value of 0.

endSrc is (one past) the end of the sequence of Partials corresponding to a morph function value of 0.

beginTgt is the beginning of the sequence of Partials corresponding to a morph function value of 1.

endTgt is (one past) the end of the sequence of Partials corresponding to a morph function value of 1.

3.20.3.7 Breakpoint Loris::Morpher::morphBreakpoints (const Breakpoint & *srcBkpt*, const Breakpoint & *tgtBkpt*, double *time*) const

Compute morphed parameter values at the specified time, using the source and target Breakpoints (assumed to correspond exactly to the specified time).

Parameters:

srcBkpt is the Breakpoint corresponding to a morph function value of 0.

tgtBkpt is the Breakpoint corresponding to a morph function value of 1.

time is the time corresponding to srcBkpt (used to evaluate the morphing functions and tgtPartial).

Returns:

the morphed Breakpoint

3.20.3.8 Partial Loris::Morpher::morphPartial (const Partial & *src*, const Partial & *tgt*, int *assignLabel*)

Morph a pair of Partials to yield a new morphed Partial.

Dummy Partials (having no Breakpoints) don't contribute to the morph, except to cause their opposite to fade out. Either (or neither) the source or target Partial may be a dummy Partial (no Breakpoints), but not both. The morphed Partial has Breakpoints at times corresponding to every Breakpoint in both source Partials, omitting Breakpoints that would be closer than the minBreakpointGap to their predecessor. The new morphed Partial is assigned the specified label and returned.

Parameters:

src is the Partial corresponding to a morph function value of 0, evaluated at the specified time.

tgt is the Partial corresponding to a morph function value of 1, evaluated at the specified time.

assignLabel is the label assigned to the morphed Partial

Returns:

the morphed Partial

3.20.3.9 Breakpoint Loris::Morpher::morphSrcBreakpoint (const Breakpoint & *bp*, const Partial & *tgtPartial*, double *time*) const

Compute morphed parameter values at the specified time, using the source Breakpoint (assumed to correspond exactly to the specified time) and the target Partial (whose parameters are examined at the specified time).

Precondition:

the target Partial may not be a dummy Partial (no Breakpoints).

Parameters:

srcBkpt is the Breakpoint corresponding to a morph function value of 0.

tgtPartial is the Partial corresponding to a morph function value of 1, evaluated at the specified time.

time is the time corresponding to srcBkpt (used to evaluate the morphing functions and tgtPartial).

newpis the morphed Partial under construction, the morphed Breakpoint is added to this Partial.

3.20.3.10 Breakpoint Loris::Morpher::morphTgtBreakpoint (const Breakpoint & *bp*, const Partial & *tgtPartial*, double *time*) const

Compute morphed parameter values at the specified time, using the target Breakpoint (assumed to correspond exactly to the specified time) and the source Partial (whose parameters are examined at the specified time).

Precondition:

the source Partial may not be a dummy Partial (no Breakpoints).

Parameters:

tgtBkpt is the Breakpoint corresponding to a morph function value of 1.

srcPartial is the Partial corresponding to a morph function value of 0, evaluated at the specified time.

time is the time corresponding to srcBkpt (used to evaluate the morphing functions and tgtPartial).

newpis the morphed Partial under construction, the morphed Breakpoint is added to this Partial.

3.20.3.11 Morpher& Loris::Morpher::operator= (const Morpher & *rhs*)

Parameters:

rhs is the Morpher to duplicate

3.20.3.12 void Loris::Morpher::setAmplitudeShape (double *x*)

Set the shaping parameter for the amplitude moprhing function (only used in new log-amplitude morphing).

This shaping parameter controls the slope of the amplitude morphing function, for values greater than 1, this function gets nearly linear (like the old amplitude morphing function), for values much less than 1 (e.g. 1E-5) the slope is gently curved and sounds pretty "linear", for very small values (e.g. 1E-12) the curve is very steep and sounds un-natural because of the huge jump from zero amplitude to very small amplitude.

Parameters:

x is the new shaping parameter, it must be positive.

3.20.3.13 void Loris::Morpher::setMinBreakpointGap (double *x*)

Set the minimum time gap (secs) between two Breakpoints in the morphed Partials.

Morphing two Partials can generate a third Partial having Breakpoints arbitrarily close together in time, and this makes morphs huge. Raising this threshold limits the Breakpoint density in the morphed Partials. Default is 1/10 ms.

Parameters:

x is the new minimum gap in seconds, it must be positive

Exceptions:

InvalidArgument if the specified gap is not positive

3.20.3.14 void Loris::Morpher::setSourceReferenceLabel (Partial::label_type *l*)

Set the label of the Partial to be used as a reference Partial for the source sequence in a morph of two Partial sequences.

The reference partial is used to compute frequencies for very low-amplitude Partials whose frequency estimates are not considered reliable. The reference Partial is considered to have good frequency estimates throughout. Setting the reference label to 0 indicates that no reference Partial should be used for the source sequence.

3.20.3.15 void Loris::Morpher::setTargetReferenceLabel (Partial::label_type *l*)

Set the label of the Partial to be used as a reference Partial for the target sequence in a morph of two Partial sequences.

The reference partial is used to compute frequencies for very low-amplitude Partials whose frequency estimates are not considered reliable. The reference Partial is considered to have good frequency estimates throughout. Setting the reference label to 0 indicates that no reference Partial should be used for the target sequence.

3.20.3.16 Partial::label_type Loris::Morpher::sourceReferenceLabel (void) const

Return the label of the Partial to be used as a reference Partial for the source sequence in a morph of two Partial sequences.

The reference partial is used to compute frequencies for very low-amplitude Partials whose frequency estimates are not considered reliable. The reference Partial is considered to have good frequency estimates throughout. The default label of 0 indicates that no reference Partial should be used for the source sequence.

3.20.3.17 Partial::label_type Loris::Morpher::targetReferenceLabel (void) const

Return the label of the Partial to be used as a reference Partial for the target sequence in a morph of two Partial sequences.

The reference partial is used to compute frequencies for very low-amplitude Partials whose frequency estimates are not considered reliable. The reference Partial is considered to have good frequency estimates throughout. The default label of 0 indicates that no reference Partial should be used for the target sequence.

3.21 Loris::Partial Class Reference

An instance of class Partial represents a single component in the reassigned bandwidth-enhanced additive model.

`#include <Partial.h>`

Public Types

- typedef std::map< double, Breakpoint > container_type

  underlying Breakpoint container type, used by the iterator types defined below:

- typedef int label_type

  32 bit type for labeling Partials

- typedef Partial_Iterator iterator

  non-const iterator over (time, Breakpoint) pairs in this Partial

- typedef Partial_ConstIterator const_iterator

  const iterator over (time, Breakpoint) pairs in this Partial

- typedef container_type::size_type size_type

  size type for number of Breakpoints in this Partial

Public Member Functions

- Partial (void)

  Retun a new empty (no Breakpoints) Partial.

- Partial (const_iterator beg, const_iterator end)

  Retun a new Partial from a half-open (const) iterator range of time-Breakpoint pairs.

- Partial (const Partial &other)

  Return a new Partial that is an exact copy (has an identical set of Breakpoints, at identical times, and the same label) of another Partial.

- ∼Partial (void)

  Destroy this Partial.

- Partial & operator= (const Partial &other)

  Make this Partial an exact copy (has an identical set of Breakpoints, at identical times, and the same label) of another Partial.

- iterator begin (void)

  Return an iterator refering to the position of the first Breakpoint in this Partial's envelope, or end() if there are no Breakpoints in the Partial.

- const_iterator begin (void) const

  Return a const iterator refering to the position of the first Breakpoint in this Partial's envelope, or end() if there are no Breakpoints in the Partial.

- iterator end (void)

  Return an iterator refering to the position past the last Breakpoint in this Partial's envelope.

- const_iterator end (void) const

  Return a const iterator refering to the position past the last Breakpoint in this Partial's envelope.

- iterator erase (iterator beg, iterator end)

  Breakpoint removal: erase the Breakpoints in the specified range, and return an iterator referring to the position after the, erased range.

- iterator findAfter (double time)

  Return an iterator refering to the insertion position for a Breakpoint at the specified time (that is, the position of the first Breakpoint at a time later than the specified time).

- const_iterator findAfter (double time) const

  Return a const iterator refering to the insertion position for a Breakpoint at the specified time (that is, the position of the first Breakpoint at a time later than the specified time).

- iterator insert (double time, const Breakpoint &bp)

  Breakpoint insertion: insert a copy of the specified Breakpoint in the parameter envelope at time (seconds), and return an iterator refering to the position of the inserted Breakpoint.

- size_type size (void) const

  Return the number of Breakpoints in this Partial.

- double duration (void) const

  Return the duration (in seconds) spanned by the Breakpoints in this Partial.

- double endTime (void) const

  Return the time (in seconds) of the last Breakpoint in this Partial.

- Breakpoint & first (void)

  Return a reference to the first Breakpoint in the Partial's envelope.

- const Breakpoint & first (void) const

  Return a const reference to the first Breakpoint in the Partial's envelope.

- double initialPhase (void) const

  Return the phase (in radians) of this Partial at its start time (the phase of the first Breakpoint).

- label_type label (void) const

  Return the 32-bit label for this Partial as an integer.

- Breakpoint & last (void)

  Return a reference to the last Breakpoint in the Partial's envelope.

- const Breakpoint & last (void) const

  Return a const reference to the last Breakpoint in the Partial's envelope.

- size_type numBreakpoints (void) const

  Same as size(). Return the number of Breakpoints in this Partial.

- double startTime (void) const

  Return the time (in seconds) of the first Breakpoint in this Partial.

- void absorb (const Partial &other)

  Absorb another Partial's energy as noise (bandwidth), by accumulating the other's energy as noise energy in the portion of this Partial's envelope that overlaps (in time) with the other Partial's envelope.

- void setLabel (label_type l)

  Set the label for this Partial to the specified 32-bit value.

- iterator erase (iterator pos)

  Remove the Breakpoint at the position of the given iterator, invalidating the iterator.

- iterator findNearest (double time)

  Return an iterator refering to the position of the Breakpoint in this Partial nearest the specified time.

- const_iterator findNearest (double time) const

  Return a const iterator refering to the position of the Breakpoint in this Partial nearest the specified time.

- Partial split (iterator pos)

  Break this Partial at the specified position (iterator).

- double amplitudeAt (double time, double fadeTime=ShortestSafeFadeTime) const

  Return the interpolated amplitude of this Partial at the specified time.

- double bandwidthAt (double time) const

  Return the interpolated bandwidth (noisiness) coefficient of this Partial at the specified time.

- double frequencyAt (double time) const

  Return the interpolated frequency (in Hz) of this Partial at the specified time.

- double phaseAt (double time) const

  Return the interpolated phase (in radians) of this Partial at the specified time.

- Breakpoint parametersAt (double time, double fadeTime=ShortestSafeFadeTime) const

  Return the interpolated parameters of this Partial at the specified time, same as building a Breakpoint from the results of frequencyAt, ampitudeAt, bandwidthAt, and phaseAt, but performs only one Breakpoint envelope search.

Static Public Attributes

- const double ShortestSafeFadeTime

  Define the default fade time for computing amplitude at the ends of a Partial.

3.21.1 Detailed Description

An instance of class Partial represents a single component in the reassigned bandwidth-enhanced additive model.

A Partial consists of a chain of Breakpoints describing the time-varying frequency, amplitude, and bandwidth (or noisiness) envelopes of the component, and a 4-byte label. The Breakpoints are non-uniformly distributed in time. For more information about

Reassigned Bandwidth-Enhanced Analysis and the Reassigned Bandwidth-Enhanced Additive Sound Model, refer to the Loris website: www.cerlsoundgroup.org/Loris/.

The constituent time-tagged Breakpoints are accessible through Partial:iterator and Partial::const_iterator interfaces. These iterator classes implement the interface for bidirectional iterators in the STL, including pre and post-increment and decrement, and dereferencing. Dereferencing a Partial::itertator or Partial::const_itertator yields a reference to a Breakpoint. Additionally, these iterator classes have breakpoint() and time() members, returning the Breakpoint (by reference) at the current iterator position and the time (by value) corresponding to that Breakpoint.

Partial is a leaf class, do not subclass.

Most of the implementation of Partial delegates to a few container-dependent members. The following members are container-dependent, the other members are implemented in terms of these: default construction copy (construction) operator= (assign) operator== (equivalence) size insert(pos, Breakpoint) erase(b, e) findAfter(time) begin (const and non-const) end (const and non-const) first (const and non-const) last (const and non-const)

3.21.2 Constructor & Destructor Documentation

3.21.2.1 Loris::Partial::Partial (const_iterator *beg*, const_iterator *end*)

Retun a new Partial from a half-open (const) iterator range of time-Breakpoint pairs.

Parameters:

- ***beg*** is the beginning of the range of time-Breakpoint pairs to insert into the new Partial.
- ***end*** is the end of the range of time-Breakpoint pairs to insert into the new Partial.

3.21.2.2 Loris::Partial::Partial (const Partial & *other*)

Return a new Partial that is an exact copy (has an identical set of Breakpoints, at identical times, and the same label) of another Partial.

Parameters:

- ***other*** is the Partial to copy.

3.21.3 Member Function Documentation

3.21.3.1 void Loris::Partial::absorb (const Partial & *other*)

Absorb another Partial's energy as noise (bandwidth), by accumulating the other's energy as noise energy in the portion of this Partial's envelope that overlaps (in time) with the other Partial's envelope.

Parameters:

other is the Partial to absorb.

3.21.3.2 double Loris::Partial::amplitudeAt (double *time*, double *fadeTime* = ShortestSafeFadeTime) const

Return the interpolated amplitude of this Partial at the specified time.

If non-zero fadeTime is specified, then the amplitude at the ends of the Partial is computed using a linear fade. The default fadeTime is ShortestSafeFadeTime, see the definition of ShortestSafeFadeTime, above.

Parameters:

time is the time in seconds at which to evaluate the Partial.

fadeTime is the duration in seconds over which Partial amplitudes fade at the ends. The default value is ShortestSafeFadeTime, 1 ns.

Returns:

The amplitude of this Partial at the specified time.

Precondition:

The Partial must have at least one Breakpoint.

Exceptions:

InvalidPartial if the Partial has no Breakpoints.

3.21.3.3 double Loris::Partial::bandwidthAt (double *time*) const

Return the interpolated bandwidth (noisiness) coefficient of this Partial at the specified time.

At times beyond the ends of the Partial, return the bandwidth coefficient at the nearest envelope endpoint.

Parameters:

time is the time in seconds at which to evaluate the Partial.

Returns:

The bandwidth of this Partial at the specified time.

Precondition:

The Partial must have at least one Breakpoint.

Exceptions:

InvalidPartial if the Partial has no Breakpoints.

3.21.3.4 double Loris::Partial::duration (void) const

Return the duration (in seconds) spanned by the Breakpoints in this Partial.

Note that the synthesized onset time will differ, depending on the fade time used to synthesize this Partial (see class Synthesizer).

3.21.3.5 const_iterator Loris::Partial::end (void) const

Return a const iterator refering to the position past the last Breakpoint in this Partial's envelope.

The iterator returned by end() (like the iterator returned by the end() member of any STL container) does not refer to a valid Breakpoint.

3.21.3.6 iterator Loris::Partial::end (void)

Return an iterator refering to the position past the last Breakpoint in this Partial's envelope.

The iterator returned by end() (like the iterator returned by the end() member of any STL container) does not refer to a valid Breakpoint.

3.21.3.7 double Loris::Partial::endTime (void) const

Return the time (in seconds) of the last Breakpoint in this Partial.

Note that the synthesized onset time will differ, depending on the fade time used to synthesize this Partial (see class Synthesizer).

3.21.3.8 iterator Loris::Partial::erase (iterator *pos*)

Remove the Breakpoint at the position of the given iterator, invalidating the iterator.

Return a iterator referring to the next valid position, or to the end of the Partial if the last Breakpoint is removed.

Parameters:

pos is the position of the time-Breakpoint pair to be removed.

Returns:

The position (iterator) of the time-Breakpoint pair after the one that was removed.

Postcondition:

The iterator pos is invalid.

3.21.3.9 iterator Loris::Partial::erase (iterator *beg*, iterator *end*)

Breakpoint removal: erase the Breakpoints in the specified range, and return an iterator referring to the position after the, erased range.

Parameters:

beg is the beginning of the range of Breakpoints to erase

end is the end of the range of Breakpoints to erase

Returns:

The position of the first Breakpoint after the range of removed Breakpoints, or end() if the last Breakpoint in the Partial was removed.

3.21.3.10 const_iterator Loris::Partial::findAfter (double *time*) const

Return a const iterator refering to the insertion position for a Breakpoint at the specified time (that is, the position of the first Breakpoint at a time later than the specified time).

Parameters:

time is the time in seconds to find

Returns:

The last position (iterator) at which a Breakpoint at the specified time could be inserted (the position of the first Breakpoint later than time).

3.21.3.11 iterator Loris::Partial::findAfter (double *time*)

Return an iterator refering to the insertion position for a Breakpoint at the specified time (that is, the position of the first Breakpoint at a time later than the specified time).

Parameters:

time is the time in seconds to find

Returns:

The last position (iterator) at which a Breakpoint at the specified time could be inserted (the position of the first Breakpoint later than time).

3.21.3.12 const_iterator Loris::Partial::findNearest (double *time*) const

Return a const iterator refering to the position of the Breakpoint in this Partial nearest the specified time.

Parameters:

time is the time to find.

Returns:

The position (iterator) of the time-Breakpoint pair nearest (in time) to the specified time.

3.21.3.13 iterator Loris::Partial::findNearest (double *time*)

Return an iterator refering to the position of the Breakpoint in this Partial nearest the specified time.

Parameters:

time is the time to find.

Returns:

The position (iterator) of the time-Breakpoint pair nearest (in time) to the specified time.

3.21.3.14 const Breakpoint& Loris::Partial::first (void) const

Return a const reference to the first Breakpoint in the Partial's envelope.

Exceptions:

InvalidPartial if there are no Breakpoints.

3.21.3.15 Breakpoint& Loris::Partial::first (void)

Return a reference to the first Breakpoint in the Partial's envelope.

Exceptions:

InvalidPartial if there are no Breakpoints.

3.21.3.16 double Loris::Partial::frequencyAt (double *time*) const

Return the interpolated frequency (in Hz) of this Partial at the specified time.

At times beyond the ends of the Partial, return the frequency at the nearest envelope endpoint.

Parameters:

time is the time in seconds at which to evaluate the Partial.

Returns:

The frequency of this Partial at the specified time.

Precondition:

The Partial must have at least one Breakpoint.

Exceptions:

InvalidPartial if the Partial has no Breakpoints.

3.21.3.17 double Loris::Partial::initialPhase (void) const

Return the phase (in radians) of this Partial at its start time (the phase of the first Breakpoint).

Note that the initial synthesized phase will differ, depending on the fade time used to synthesize this Partial (see class Synthesizer).

3.21.3.18 iterator Loris::Partial::insert (double *time*, const Breakpoint & *bp*)

Breakpoint insertion: insert a copy of the specified Breakpoint in the parameter envelope at time (seconds), and return an iterator refering to the position of the inserted Breakpoint.

Parameters:

time is the time in seconds at which to insert the new Breakpoint.

bp is the new Breakpoint to insert.

Returns:

the position (iterator) of the newly-inserted time-Breakpoint pair.

3.21.3.19 const Breakpoint& Loris::Partial::last (void) const

Return a const reference to the last Breakpoint in the Partial's envelope.

Exceptions:

InvalidPartial if there are no Breakpoints.

3.21.3.20 Breakpoint& Loris::Partial::last (void)

Return a reference to the last Breakpoint in the Partial's envelope.

Exceptions:

InvalidPartial if there are no Breakpoints.

3.21.3.21 Partial& Loris::Partial::operator= (const Partial & *other*)

Make this Partial an exact copy (has an identical set of Breakpoints, at identical times, and the same label) of another Partial.

Parameters:

other is the Partial to copy.

3.21.3.22 Breakpoint Loris::Partial::parametersAt (double *time*, double *fadeTime* = ShortestSafeFadeTime) const

Return the interpolated parameters of this Partial at the specified time, same as building a Breakpoint from the results of frequencyAt, ampitudeAt, bandwidthAt, and phaseAt, but performs only one Breakpoint envelope search.

If non-zero fadeTime is specified, then the amplitude at the ends of the Partial is coomputed using a linear fade. The default fadeTime is ShortestSafeFadeTime.

Parameters:

time is the time in seconds at which to evaluate the Partial.

fadeTime is the duration in seconds over which Partial amplitudes fade at the ends. The default value is ShortestSafeFadeTime, 1 ns.

Returns:

A Breakpoint describing the parameters of this Partial at the specified time.

Precondition:

The Partial must have at least one Breakpoint.

Exceptions:

InvalidPartial if the Partial has no Breakpoints.

3.21.3.23 double Loris::Partial::phaseAt (double *time*) const

Return the interpolated phase (in radians) of this Partial at the specified time.

At times beyond the ends of the Partial, return the extrapolated from the nearest envelope endpoint (assuming constant frequency, as reported by frequencyAt()).

Parameters:

time is the time in seconds at which to evaluate the Partial.

Returns:

The phase of this Partial at the specified time.

Precondition:

The Partial must have at least one Breakpoint.

Exceptions:

InvalidPartial if the Partial has no Breakpoints.

3.21.3.24 size_type Loris::Partial::size (void) const

Return the number of Breakpoints in this Partial.

Returns:

The number of Breakpoints in this Partial.

3.21.3.25 Partial Loris::Partial::split (iterator *pos*)

Break this Partial at the specified position (iterator).

The Breakpoint at the specified position becomes the first Breakpoint in a new Partial. Breakpoints at the specified position and subsequent positions are removed from this Partial and added to the new Partial, which is returned.

Parameters:

pos is the position at which to split this Partial.

Returns:

A new Partial consisting of time-Breakpoint pairs beginning with pos and extending to the end of this Partial.

Postcondition:

All positions beginning with pos and extending to the end of this Partial have been removed.

3.21.3.26 double Loris::Partial::startTime (void) const

Return the time (in seconds) of the first Breakpoint in this Partial.

Note that the synthesized onset time will differ, depending on the fade time used to synthesize this Partial (see class Synthesizer).

3.21.4 Member Data Documentation

3.21.4.1 const double Loris::Partial::ShortestSafeFadeTime `[static]`

Define the default fade time for computing amplitude at the ends of a Partial.

Floating point round-off errors make fadeTime == 0.0 dangerous and unpredictable. 1 ns is short enough to prevent rounding errors in the least significant bit of a 48-bit mantissa for times up to ten hours.

1 nanosecond, see Partial.C

3.22 Loris::Partial_ConstIterator Class Reference

Const iterator for the Loris::Partial Breakpoint map.

`#include <Partial.h>`

Public Member Functions

- Partial_ConstIterator (void)

  Construct a new iterator referring to no position in any Partial.

- Partial_ConstIterator (const Partial_Iterator &other)

  Construct a new const iterator from a non-const iterator.

- Partial_ConstIterator & operator++ ()

  Pre-increment operator - advance the position of the iterator and return the iterator itself.

- Partial_ConstIterator & operator– ()

  Pre-decrement operator - move the position of the iterator back by one and return the iterator itself.

- Partial_ConstIterator operator++ (int)

  Post-increment operator - advance the position of the iterator and return a copy of the iterator before it was advanced.

- Partial_ConstIterator operator– (int)

  Post-decrement operator - move the position of the iterator back by one and return a copy of the iterator before it was decremented.

- const Breakpoint & operator $*$ (void) const

  Dereference operator.

- const Breakpoint $*$ operator $\rightarrow$ (void) const

  Pointer operator.

- const Breakpoint & breakpoint (void) const

  Breakpoint accessor.

- double time (void) const

Time accessor.

Friends

- bool operator== (const Partial_ConstIterator &lhs, const Partial_ConstIterator &rhs)

  Equality comparison operator.

- bool operator!= (const Partial_ConstIterator &lhs, const Partial_ConstIterator &rhs)

  Inequality comparison operator.

3.22.1 Detailed Description

Const iterator for the Loris::Partial Breakpoint map.

Wraps the non-const iterator for the (time,Breakpoint) pair container Partial::container_type. Partial_Iterator implements a bidirectional iterator interface, and additionally offers time and Breakpoint (reference) access through time() and breakpoint() members.

3.22.2 Constructor & Destructor Documentation

3.22.2.1 Loris::Partial_ConstIterator::Partial_ConstIterator (const Partial_Iterator & *other*)

Construct a new const iterator from a non-const iterator.

Parameters:

other a non-const iterator from which to make a read-only copy.

3.22.3 Member Function Documentation

3.22.3.1 const Breakpoint& Loris::Partial_ConstIterator::breakpoint (void) const

Breakpoint accessor.

Returns:
A const reference to the Breakpoint at the position of this iterator.

3.22.3.2 const Breakpoint& Loris::Partial_ConstIterator::operator $*$ (void) const

Dereference operator.

Returns:
A const reference to the Breakpoint at the position of this iterator.

3.22.3.3 Partial_ConstIterator Loris::Partial_ConstIterator::operator++ (int)

Post-increment operator - advance the position of the iterator and return a copy of the iterator before it was advanced.

The int argument is unused compiler magic.

Returns:
An iterator that is a copy of this iterator before being advanced.

Precondition:
The iterator must be a valid position before the end in some Partial.

3.22.3.4 Partial_ConstIterator& Loris::Partial_ConstIterator::operator++ ()

Pre-increment operator - advance the position of the iterator and return the iterator itself.

Returns:
This iterator (reference to self).

Precondition:
The iterator must be a valid position before the end in some Partial.

3.22.3.5 Partial_ConstIterator Loris::Partial_ConstIterator::operator– (int)

Post-decrement operator - move the position of the iterator back by one and return a copy of the iterator before it was decremented.

The int argument is unused compiler magic.

Returns:
An iterator that is a copy of this iterator before being decremented.

Precondition:
The iterator must be a valid position after the beginning in some Partial.

3.22.3.6 Partial_ConstIterator& Loris::Partial_ConstIterator::operator– ()

Pre-decrement operator - move the position of the iterator back by one and return the iterator itself.

Returns:
This iterator (reference to self).

Precondition:
The iterator must be a valid position after the beginning in some Partial.

3.22.3.7 const Breakpoint∗ Loris::Partial_ConstIterator::operator → (void) const

Pointer operator.

Returns:
A const pointer to the Breakpoint at the position of this iterator.

3.22.3.8 double Loris::Partial_ConstIterator::time (void) const

Time accessor.

Returns:

The time in seconds of the Breakpoint at the position of this iterator.

3.22.4 Friends And Related Function Documentation

3.22.4.1 bool operator!= (const Partial_ConstIterator & *lhs*, const Partial_ConstIterator & *rhs*) `[friend]`

Inequality comparison operator.

Parameters:

lhs the iterator on the left side of the operator.

rhs the iterator on the right side of the operator.

Returns:

false if the two iterators refer to the same position in the same Partial, true otherwise.

3.22.4.2 bool operator== (const Partial_ConstIterator & *lhs*, const Partial_ConstIterator & *rhs*) `[friend]`

Equality comparison operator.

Parameters:

lhs the iterator on the left side of the operator.

rhs the iterator on the right side of the operator.

Returns:

true if the two iterators refer to the same position in the same Partial, false otherwise.

3.23 Loris::Partial_Iterator Class Reference

Non-const iterator for the Loris::Partial Breakpoint map.

`#include <Partial.h>`

Public Member Functions

- Partial_Iterator (void)

  Construct a new iterator referring to no position in any Partial.

- Partial_Iterator & operator++ ()

  Pre-increment operator - advance the position of the iterator and return the iterator itself.

- Partial_Iterator & operator– ()

  Pre-decrement operator - move the position of the iterator back by one and return the iterator itself.

- Partial_Iterator operator++ (int)

  Post-increment operator - advance the position of the iterator and return a copy of the iterator before it was advanced.

- Partial_Iterator operator– (int)

  Post-decrement operator - move the position of the iterator back by one and return a copy of the iterator before it was decremented.

- Breakpoint & operator ∗ (void) const

  Dereference operator.

- Breakpoint ∗ operator → (void) const

  Pointer operator.

- Breakpoint & breakpoint (void) const

  Breakpoint accessor.

- double time (void) const

  Time accessor.

Friends

- bool operator== (const Partial_Iterator &lhs, const Partial_Iterator &rhs)

  Equality comparison operator.

- bool operator!= (const Partial_Iterator &lhs, const Partial_Iterator &rhs)

  Inequality comparison operator.

3.23.1 Detailed Description

Non-const iterator for the Loris::Partial Breakpoint map.

Wraps the non-const iterator for the (time,Breakpoint) pair container Partial::container_type. Partial_Iterator implements a bidirectional iterator interface, and additionally offers time and Breakpoint (reference) access through time() and breakpoint() members.

3.23.2 Member Function Documentation

3.23.2.1 Breakpoint& Loris::Partial_Iterator::breakpoint (void) const

Breakpoint accessor.

Returns:

A const reference to the Breakpoint at the position of this iterator.

3.23.2.2 Breakpoint& Loris::Partial_Iterator::operator $*$ (void) const

Dereference operator.

Returns:

A reference to the Breakpoint at the position of this iterator.

3.23.2.3 Partial_Iterator Loris::Partial_Iterator::operator++ (int)

Post-increment operator - advance the position of the iterator and return a copy of the iterator before it was advanced.

The int argument is unused compiler magic.

Returns:
An iterator that is a copy of this iterator before being advanced.

Precondition:
The iterator must be a valid position before the end in some Partial.

3.23.2.4 Partial_Iterator& Loris::Partial_Iterator::operator++ ()

Pre-increment operator - advance the position of the iterator and return the iterator itself.

Returns:
This iterator (reference to self).

Precondition:
The iterator must be a valid position before the end in some Partial.

3.23.2.5 Partial_Iterator Loris::Partial_Iterator::operator– (int)

Post-decrement operator - move the position of the iterator back by one and return a copy of the iterator before it was decremented.

The int argument is unused compiler magic.

Returns:
An iterator that is a copy of this iterator before being decremented.

Precondition:
The iterator must be a valid position after the beginning in some Partial.

3.23.2.6 Partial_Iterator& Loris::Partial_Iterator::operator– ()

Pre-decrement operator - move the position of the iterator back by one and return the iterator itself.

Returns:
This iterator (reference to self).

Precondition:
The iterator must be a valid position after the beginning in some Partial.

3.23.2.7 Breakpoint∗ Loris::Partial_Iterator::operator → (void) const

Pointer operator.

Returns:
A pointer to the Breakpoint at the position of this iterator.

3.23.2.8 double Loris::Partial_Iterator::time (void) const

Time accessor.

Returns:
The time in seconds of the Breakpoint at the position of this iterator.

3.23.3 Friends And Related Function Documentation

3.23.3.1 bool operator!= (const Partial_Iterator & *lhs*, const Partial_Iterator & *rhs*) `[friend]`

Inequality comparison operator.

Parameters:

lhs the iterator on the left side of the operator.

rhs the iterator on the right side of the operator.

Returns:

false if the two iterators refer to the same position in the same Partial, true otherwise.

3.23.3.2 bool operator== (const Partial_Iterator & *lhs*, const Partial_Iterator & *rhs*) `[friend]`

Equality comparison operator.

Parameters:

lhs the iterator on the left side of the operator.

rhs the iterator on the right side of the operator.

Returns:

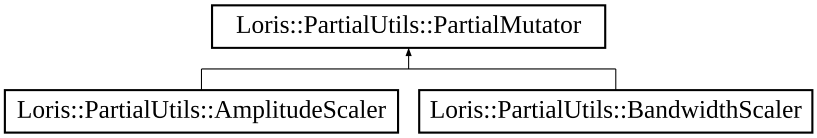
true if the two iterators refer to the same position in the same Partial, false otherwise.

3.24 Loris::PartialUtils::PartialMutator Class Reference

PartialMutator is an abstract base class for Partial mutators, functors that operate on Partials according to a time-varying envelope.

```
#include <PartialUtils.h>
```

Inheritance diagram for Loris::PartialUtils::PartialMutator::


Loris::PartialUtils::PartialMutator
Loris::PartialUtils::AmplitudeScaler
Loris::PartialUtils::BandwidthScaler

Public Member Functions

- PartialMutator (double x)

  Construct a new PartialMutator from a constant mutation factor.

- PartialMutator (const Envelope &e)

  Construct a new PartialMutator from an Envelope representing a time-varying mutation factor.

- PartialMutator (const PartialMutator &rhs)

  Construct a new PartialMutator that is a copy of another.

- virtual ~PartialMutator (void)

  Destroy this PartialMutator, deleting its Envelope.

- PartialMutator & operator= (const PartialMutator &rhs)

  Make this PartialMutator a duplicate of another one.

- virtual void operator() (Partial &p) const =0

  Function call operator: apply a mutation factor to the specified Partial.

3.24.1 Detailed Description

PartialMutator is an abstract base class for Partial mutators, functors that operate on Partials according to a time-varying envelope.

The base class manages a polymorphic Envelope instance that provides the time-varying mutation parameters.

Invariant:

env is a non-zero pointer to a valid instance of a class derived from the abstract class Envelope.

3.24.2 Member Function Documentation

3.24.2.1 virtual void Loris::PartialUtils::PartialMutator::operator() (Partial & *p*) const `[pure virtual]`

Function call operator: apply a mutation factor to the specified Partial.

Derived classes must implement this member.

Implemented in Loris::PartialUtils::AmplitudeScaler, and Loris::PartialUtils::BandwidthScaler.

3.24.2.2 PartialMutator& Loris::PartialUtils::PartialMutator::operator= (const PartialMutator & *rhs*)

Make this PartialMutator a duplicate of another one.

Parameters:

rhs is the PartialMutator to copy.

3.25 Loris::Resampler Class Reference

Class Resampler represents an algorithm for resampling Partial envelopes at regular time intervals.

```
#include <Resampler.h>
```

Public Member Functions

- Resampler (double sampleInterval)

  Construct a new Resampler using the specified sampling interval.

- void resample (Partial &p) const

  is performed in-place.

- void operator() (Partial &p) const

  Function call operator: same as resample(p).

- template<typename Iter> void resample (Iter begin, Iter end) const

  Resample all Partials in the specified (half-open) range using this Resampler's stored sampling interval, so that the Breakpoints in the Partial envelopes will all lie on a common temporal grid.

- template<typename Iter> void operator() (Iter begin, Iter end) const

  Function call operator: same as resample(begin, end).

Static Public Member Functions

- template<typename Iter> void resample (Iter begin, Iter end, double sampleInterval)

  Static member that constructs an instance and applies it to a sequence of Partials.

3.25.1 Detailed Description

Class Resampler represents an algorithm for resampling Partial envelopes at regular time intervals.

Resampling makes the envelope data more suitable for exchange (as SDIF data, for example) with other applications that cannot process raw (continuously-distributed) reassigned data. Resampling will often greatly reduce the size of the data (by greatly reducing the number of Breakpoints in the Partials) without adversely affecting the quality of the reconstruction.

3.25.2 Constructor & Destructor Documentation

3.25.2.1 Loris::Resampler::Resampler (double *sampleInterval*) `[explicit]`

Construct a new Resampler using the specified sampling interval.

Parameters:

sampleInterval is the resampling interval in seconds, Breakpoint data is computed at integer multiples of sampleInterval seconds.

Exceptions:

InvalidArgument if sampleInterval is not positive.

3.25.3 Member Function Documentation

3.25.3.1 template<typename Iter> void Loris::Resampler::resample (Iter *begin*, Iter *end*, double *sampleInterval*) `[static]`

Static member that constructs an instance and applies it to a sequence of Partials.

Construct a Resampler using the specified resampling interval, and use it to channelize a sequence of Partials.

Parameters:

begin is the beginning of a sequence of Partials to resample.

end is the end of a sequence of Partials to resample.

sampleInterval is the resampling interval in seconds, Breakpoint data is computed at integer multiples of sampleInterval seconds.

Exceptions:

InvalidArgument if sampleInterval is not positive.

If compiled with NO_TEMPLATE_MEMBERS defined, then begin and end must be PartialList::iterators, otherwise they can be any type of iterators over a sequence of Partials.

3.25.3.2 template<typename Iter> void Loris::Resampler::resample (Iter *begin*, Iter *end*) const

Resample all Partials in the specified (half-open) range using this Resampler's stored sampling interval, so that the Breakpoints in the Partial envelopes will all lie on a common temporal grid.

The Breakpoint times in the resampled Partial will comprise a contiguous sequence of integer multiples of the sampling interval, beginning with the multiple nearest to the Partial's start time and ending with the multiple nearest to the Partial's end time. Resampling is performed in-place.

Parameters:

begin is the beginning of the range of Partials to resample

end is (one-past) the end of the range of Partials to resample

If compiled with NO_TEMPLATE_MEMBERS defined, then begin and end must be PartialList::iterators, otherwise they can be any type of iterators over a sequence of Partials.

3.25.3.3 void Loris::Resampler::resample (Partial & *p*) const

is performed in-place.

Parameters:

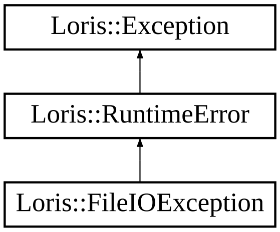
p is the Partial to resample

3.26 Loris::RuntimeError Class Reference

Class of exceptions thrown when an unanticipated runtime error is encountered.

`#include <Exception.h>`

Inheritance diagram for Loris::RuntimeError::



Public Member Functions

- RuntimeError (const std::string &str, const std::string &where="")

  string automatically using __FILE__ and __LINE__.

3.26.1 Detailed Description

Class of exceptions thrown when an unanticipated runtime error is encountered.

3.26.2 Constructor & Destructor Documentation

3.26.2.1 Loris::RuntimeError::RuntimeError (const std::string & *str*, const std::string & *where* = "")

string automatically using __FILE__ and __LINE__.

Parameters:

str is a string describing the exceptional condition

where is an option string describing the location in the source code from which the exception was thrown (generated automatically by the Throw macro).

3.27 Loris::Sieve Class Reference

Class Sieve represents an algorithm for identifying channelized (see Channelizer) Partials that overlap in time, and selecting the longer one to represent the channel.

```
#include <Sieve.h>
```

Public Member Functions

- Sieve (double partialFadeTime=0.001)

  Construct a new Sieve using the specified partial fade time.

- template<typename Iter> void sift (Iter sift_begin, Iter sift_end)

  Sift labeled Partials on the specified half-open (STL-style) range.

Static Public Member Functions

- template<typename Iter> void sift (Iter sift_begin, Iter sift_end, double partialFadeTime)

  Static member that constructs an instance and applies it to a sequence of Partials.

3.27.1 Detailed Description

Class Sieve represents an algorithm for identifying channelized (see Channelizer) Partials that overlap in time, and selecting the longer one to represent the channel.

The identification of overlap includes the time needed for Partials to fade to and from zero amplitude in synthesis (

See also:

Synthesizer) or distillation. (
Distiller)

In some cases, the energy redistribution effected by the distiller (see Distiller) is undesirable. In such cases, the partials can be sifted before distillation. The sifting process in Loris identifies all the partials that would be rejected (and converted to noise energy) by the distiller and assigns them a label of 0. These sifted partials can then be identified and treated sepearately or removed altogether, or they can be passed through the distiller unlabeled, and crossfaded in the morphing process (

See also:

Morpher).

3.27.2 Constructor & Destructor Documentation

3.27.2.1 Loris::Sieve::Sieve (double *partialFadeTime* = 0.001) `[explicit]`

Construct a new Sieve using the specified partial fade time.

If unspecified, the fade time defaults to one millisecond (0.001 s).

Parameters:

partialFadeTime is the extra time (in seconds) added to each end of a Partial to accomodate the fade to and from zero amplitude. Default is 0.001 (one millisecond). The Partial fade time must be non-negative.

Exceptions:

InvalidArgument if partialFadeTime is negative.

3.27.3 Member Function Documentation

3.27.3.1 template<typename Iter> void Loris::Sieve::sift (Iter *sift_begin*, Iter *sift_end*, double *partialFadeTime*) `[static]`

Static member that constructs an instance and applies it to a sequence of Partials.

Construct a Sieve using the specified Partial fade time (in seconds), and use it to sift a sequence of Partials.

Parameters:

sift_begin is the beginning of the range of Partials to sift

sift_end is (one-past) the end of the range of Partials to sift

partialFadeTime is the extra time (in seconds) added to each end of a Partial to accomodate the fade to and from zero amplitude. The Partial fade time must be non-negative.

Exceptions:

InvalidArgument if partialFadeTime is negative.

If compiled with NO_TEMPLATE_MEMBERS defined, then begin and end must be PartialList::iterators, otherwise they can be any type of iterators over a sequence of Partials.

3.27.3.2 template<typename Iter> void Loris::Sieve::sift (Iter *sift_begin*, Iter *sift_end*)

Sift labeled Partials on the specified half-open (STL-style) range.

Parameters:

- ***sift_begin*** is the beginning of the range of Partials to sift
- ***sift_end*** is (one-past) the end of the range of Partials to sift

If compiled with NO_TEMPLATE_MEMBERS defined, then sift_begin and sift_end must be PartialList::iterators, otherwise they can be any type of iterators over a sequence of Partials.

Index