retune 1.0.0

by zplane.development

June 7, 2022

Contents

1 reTune Documentation

1.1 Introduction

reTune is zplane's real-time multi-pitch modification tool. It enables shifts of pitches belonging to the same pitch class to any other pitch class (e.g. a shift of all Cs to C#s). This enables a transformation of the input audio from any input key to any output key (e.g. from C major to C# minor). Since each pitch class can be mapped arbitrarily to any other pitch class, mappings not only include major and minor input and output keys, but any other scale such as church modes or pentatonic scales. reTune runs in real time and the pitch mapping can likewise be modified online during processing.

In addition to the mapping of individual pitches, reTune can correct each pitch by mapping notes to the nearest semitone thereby eliminating inaccurate intonation. The API provides control over various parameters such as the sensitivity of the pitch detection, the detection of transients as well as the smoothing of the pitch contour.

Description of basic functionality
Operating system compliance
Structure of document

1.2 API Documentation

1.2.1 General

reTune is a real-time algorithm. That means that in each call to the process function, ReTuneIf::ProcessData, the caller provides a number of input frames and reTune returns a number of output frames. Input and output of a single process call have a fixed latency. The caller can specify an output block size and given that the user provides the required number of input frames, each process call will write a full output block into the specified buffer. The required number of input frames needs to be retrieved by the function ReTuneIf::GetFramesNeeded before a process call and the correct number of frames should be passed to the process function to ensure the correct latency between input and output. Once all input frames have been provided, the remaining output blocks can be retrieved by the function ReTuneIf::FlushBuffer.

Preferences for reTune can be set both before and during processing. This includes the mapping of semitones, the pitch correction, as well as all other processing parameters.

1.2.2 Naming Conventions

The following naming conventions are used throughout this manual:

A **frame** denotes the number of audio samples per channel, i.e. 512 stereo frames correspond to 1024 float values (samples).

A **pitch class** describes the name of a pitch independent of the octave it occurred in. In other words, notes with pitches at C0, C1, C2, etc. all belong to pitch class C.

Equal temperament is assumed for the analysis and the **tuning frequency** describes the center frequency of the pitch A4 in this temperament system. If not set differently, a

tuning frequency of 440 Hz is assumed. Specifying a different tuning frequency shifts the center frequencies of all pitches up or down.

1.2.3 Instance Handling Functions

- **static Error_t ReTuneIf::CreateInstance (ReTuneIf*& pReTune, int iOutput↩ BlockSize, float fSampleRate, int iNumOfChannels)**
  Creates a new instance of reTune. The handle to the new instance is returned in parameter pReTune. The output block size is given in frames in parameter iOutputBlockSize. fSampleRate denotes the input samplerate and iNumOf↩ Channels contains the number of channels of the input audio.
  If the function fails, the return value is different from ReTuneIf::kNoError.
  The use of this function is required.

- **static Error_t ReTuneIf::DestroyInstance (ReTuneIf*& pReTune)**
  Destroys the instance of reTune given in parameter pReTune.
  If the function fails, the return value is different from ReTuneIf::kNoError.
  The use of this function is required.

1.2.4 Parameter Setting Functions

- **Error_t ReTuneIf::SetOffset(PitchClass ePitchClass, float fOffsetInSemitones)**
  Sets the offset for the pitch class given by ePitchClass. The offset is provided by fOffsetInSemitones. This function can be used to realize a mapping of an input pitch class to any output pitch class.

- **Error_t ReTuneIf::SetEnablePitchCorrection(PitchClass ePitchClass, bool bEnable)**
  Enables/disables pitch correction for the pitch class given by ePitchClass. If bEnable is set to true, pitch correction is enabled, if set to false, it is disabled.

- **Error_t ReTuneIf::SetPitchCorrection(float fPitchCorrectionFactor)**
  Sets the global pitch correction factor. Pitch correction is only applied to the enabled pitch classes (see ReTuneIf::SetEnablePitchCorrection). A pitch correction factor of 1 snaps all pitch contours to the center frequencies of the corresponding pitch classes. A pitch correction of 0 leaves the pitch contours unaltered. Any value in between 0 and 1 applies pitch correction proportionally.

- **Error_t ReTuneIf::SetInputTuningFreq(float fTuningFreq)**
  Sets the tuning frequency of the input audio in Hz. The input tuning is used to calibrate the analysis of the input pitches and should hence match the input audio.

- **Error_t ReTuneIf::SetOutputTuningFreq(float fTuningFreq)**
  Sets the tuning frequency of the output audio.

- **Error_t ReTuneIf::SetSensitivity(float fSensitivity)**
  Sets how sensitive the algorithm should react to different f0 hypotheses. The parameter fSensitivity in dB specifies the level range below the most prominent f0 in which f0 hypotheses are taken into account. A sensitivity of 0 means that only the most prominent f0 hypothesis is taken into account, a value of -10 considers all f0 hypotheses up to 10dB below the most prominent f0. fSensitivity can be specified within the range [0dB..-40dB].

- **Error_t ReTuneIf::SetTransients(float fTransientVal)**
  Sets the transient detection strength between (0..1). Low values mean that the algorithm doesn't react very sensitive to transients while high values mean that is very sensitive.

- **Error_t ReTuneIf::SetSmoothing(float fSmooth)**
  Sets the smoothing time of pitch contours. A pitch shift might cause a gap in the pitch contour when an smooth transition is interrupted through a jump in pitch. The smoothing time specifies how quickly reTune will even out these gaps. Short smoothing times allow quick jumps while longer smoothing times result in softer pitch transitions. This feature is particularly useful for vocals.

1.2.5 Process Function

- **int ReTuneIf::GetFramesNeeded()**
  Returns the required number of input samples for the upcoming processing block. This function has to be called before each processing step to ensure correct input buffer sizes.
  The method may be called with a new output buffer size as parameter. This will change the output buffer size until it is called with another buffer size again.

- **int ReTuneIf::ProcessData(const float* const* ppfInputBlock, int iNum↩OfInputFrames, float** ppfOutputBlock)**
  Processes the input data and returns the output data at the specified latency. The input as well as the output data are given as an array of pointers to the data. This means that, for example, ppfInputBlock[0] is a pointer to the float input data buffer of the first channel while ppfInputBlock[1] is the pointer to the float input data buffer of the second input channel. The range of the audio input data is +/-1.0f. The parameter iNumOfInputFrames describes the number of input frames. Please make sure that this parameter iNumOfInputFrames equals the value returned by the function ReTuneIf::GetFramesNeeded() to ensure the requested output buffer size.

  If the function fails, the return value is different from ReTuneIf::kNoError. The use of this function is required.

- **int ReTuneIf::FlushBuffer(float** ppfOutputBlock)**
  Gets all the remaining internal frames when no more input data is available and writes them into the buffer ppfOutputBlock. Returns the number of written samples.

  The use of this function is optional.

- **int ReTuneIf::Reset()**
  Sets all internal buffers to their initial state. The call of this function is needed

before using the same instance of reTune for a different input signal. The use of this function is optional.

1.2.6 Usage Example

The command line example demonstrates how to use reTune to convert a song from major to minor or vice versa. The example application can be called by the following command

```
reTuneCl <inputFile> <outputFile> <rootPitch> <sourceScale> <targetScale>
```

The complete code can be found in the example source file reTuneClMain.cpp.

In the first step, we declare a pointer to the reTune instance and create an instance of the reTune class

```
ReTuneIf* pInstanceHandle = 0;




// create class instance
ReTuneIf::CreateInstance (pInstanceHandle,
                          kOutBlockSize,
                          pCInputFile->GetSampleRate(),
                          pCInputFile->GetNumOfChannels());
```

We set the input-to-output pitch mapping by calling

```
    GeneratePitchMapping (pInstanceHandle, commandLineParser.getParameter ("mapping"));
}
else
{
    GeneratePitchMapping (pInstanceHandle,
                          commandLineParser.getParameter ("key"),
                          commandLineParser.getParameter ("inputScale"),
                          commandLineParser.getParameter ("outputScale"));
```

with the source key and target key. Within this function, we call

```
pInstance->SetOffset ((ReTuneIf::PitchClass) sourceNote, semitoneDifference)
```

which allows us to define the mapping for each induvidual pitch class.

Next, we read chunks of data from our input file.

```
while (bReadNextFrame)
{
    int iInputSize = pInstanceHandle->GetFramesNeeded();

    iNumFramesRead = pCInputFile->Read (apfInputBuffer, iInputSize);
    if (iNumFramesRead < iInputSize)
    {
        for (ch = 0; ch < pCInputFile->GetNumOfChannels(); ch++)
            memset (&apfInputBuffer[ch][iNumFramesRead], 0, (iInputSize - iNumFramesRead) * sizeof (
  float));
        bReadNextFrame = false;
    }
```

Each chunk is processed

```
int iNumOfOutputFrames = pInstanceHandle->ProcessData (apfInputBuffer,
                                                       iInputSize,
                                                       apfOutputBuffer);
```

and the output is written to the output file.

```
if (iNumOfOutputFrames != kOutBlockSize)
{
    printf ("\nerror!");
    return -1;
}
```

After all input frames have been processed, we obtain the remaining output files by calling

```
while ((iNumOfOutputFrames = pInstanceHandle->FlushBuffer (apfOutputBuffer)) > 0)
{
    if (iNumOfOutputFrames > 0 && commandLineParser.getParameter ("output") != "" && pCOutputFile->
  IsFileOpen())
    {
        pCOutputFile->Write (apfOutputBuffer, iNumOfOutputFrames);
    }
```

Finally we destroy the ReTuneIf instance

```
ReTuneIf::DestroyInstance (pInstanceHandle);
```

This example demonstrates the basic use of the reTune API. Information about additional parameter settings can be found in Parameter Setting Functions.

1.3 Support

Support for the source code is - within the limits of the agreement - available from:

zplane.development
grunewaldstr. 83
d-10823 berlin
germany

fon: +49.30.854 09 15.0
fax: +49.30.854 09 15.5

@: info@zplane.de

2 Class Index

2.1 Class List

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

3 File Index

3.1 File List

Here is a list of all files with brief descriptions:

4 Class Documentation

4.1 ReTuneIf Class Reference

```
#include <reTune/ReTuneIf.h>
```

Public Types

- enum Error_t {
  kNoError, kMemError, kInvalidFunctionParamError, kNotInitializedError,
  kUnknownError, kNumErrors }
- enum Version_t {
  kMajor, kMinor, kPatch, kRevision,
  kNumVersionInts }
- enum PitchClass {
  C, CSharp, D, DSharp,
  E, F, FSharp, G,
  GSharp, A, ASharp, B,
  kNumOfPitchClasses }

Public Member Functions

- virtual int ProcessData (const float ∗const ∗ppfInputBlock, int iNumOfInput↩ Frames, float ∗∗ppfOutputBlock)=0

  does the actual processing if the number of frames provided is as retrieved by ReTuneIf::GetFramesNeeded() this function always returns the number of frames as specified when calling ReTuneIf::CreateInstance(..)
- virtual int FlushBuffer (float ∗∗ppfOutputBlock)=0

  gets the last frames in the internal buffer
- virtual int Reset ()=0

  clears the internal buffers of the reTune algorithm. you call this to avoid the remaining samples in the process buffer being audible when you e.g. stop playback and start it again at an other time position. other parameters (pitch offsets, sensitivity, etc.) are not reset.
- virtual int GetFramesNeeded ()=0

  returns the sample frames needed for the next process call to complete
- virtual int GetFramesNeeded (int iNewOutputBlockSize)=0

  returns the sample frames needed for the next process call to complete for a new output blocksize
- virtual int GetLatency ()=0

  returns the current latency of the algorithm

- virtual Error_t SetOffset (PitchClass ePitchClass, float fOffsetInSemitones)=0

  sets the pitch offset for each pitch class in semitones
- virtual float GetOffset (PitchClass ePitchClass) const =0

  returns the currently chosen offset for each pitch class
- virtual Error_t SetEnablePitchCorrection (PitchClass ePitchClass, bool bEnable)=0

  en-/disables the pitch correction for each pitch class
- virtual bool GetEnablePitchCorrection (PitchClass ePitchClass) const =0

  returns if pitch correction is en/disbaled for each pitch class
- virtual Error_t SetInputTuningFreq (float fTuningFreq)=0

  sets the tuning frequency of the input audio, this is important to do a proper input frequency to pitch mapping
- virtual float GetInputTuningFreq () const =0

  returns the current input tuning frequency
- virtual Error_t SetOutputTuningFreq (float fTuningFreq)=0

  sets the tuning frequency of the output audio
- virtual float GetOutputTuningFreq () const =0

  returns the current output tuning frequency
- virtual Error_t SetSensitivity (float fSensitivity)=0

  sets how sensitive the algorithm should react to different f0 hypotheses (0..1). 0 means that it will be only take the strongest f0 hypothesis while higher values mean that more f0 hypotheses are taken as f0s - this may lead to too many f0s, usually 0.25 is a good value
- virtual float GetSensitivity () const =0

  returns the current sensitivity
- virtual Error_t EnableShorterTransientMode (bool bShouldBeEnabled)=0

  Enable mode for input audio that contains short transients (e.g. piano).
- virtual Error_t SetTransients (float fTransientVal)=0

  sets the transient behavior strength between (0..1.5). Values below 1.0 mean that the algorithm should attenuate transients while values above 1.0 mean that transients should be amplified. default is 1.0
- virtual float GetTransients () const =0

  returns the current transient value (between 0..1.5)
- virtual Error_t SetSmoothing (float fSmooth)=0

  sets the smoothing speed for transitions between shifted notes. If a note is shifted the transition may be smoothed in order to avoid jumps - this is especially useful for vocals.
- virtual float GetSmoothing () const =0

  returns the current smoothing speed
- virtual Error_t SetPitchCorrection (float fPitchCorrectionFactor)=0

  sets the pitch correction for the enabled pitch classes. 0 means that no pitch correction is performed while 1 means that each pitch in an enabled pitch class is moved to the center pitch of the corresponding pitch class.
- virtual float GetPitchCorrection () const =0

  returns the current pitch correction factor

- virtual Error_t SetMaxEnvelopeEstimationFreq (float fMaxEnvelopeEstimation↩ Freq)=0

  sets the internal estimation order frequency, this one should be larger than the most prominent affected pitch. This should be well above the fundamental of the vocals or the main instrument. Needs some fiddling around.
- virtual float GetMaxEnvelopeEstimationFreq () const =0

  returns the currently chosen max envelope estimation frequency
- virtual Error_t SetEnvelopeShiftFactor (float fEnvelopeShiftFactor)=0

  sets the total shift factor for the envelope as factor. 1.o leaves the envelope in total untouched, 2.0 or 0.5 correspond to an octave up or down.
- virtual float GetEnvelopeShiftFactor () const =0

  returns the current envelope shift factor
- virtual void SetEnableEnvelopeCalculation (bool bEnable)=0

  enables envelope calculation and modification. if not needed it's better to disable as calculation takes a bit of time.
- virtual bool GetEnableEnvelopeCalculation () const =0

  returns if the envelope estimation is enabled

Static Public Member Functions

- static const char ∗ GetVersion ()
- static const char ∗ GetBuildDate ()
- static Error_t CreateInstance (ReTuneIf ∗&pReTune, int iOutputBlockSize, float fSampleRate, int iNumOfChannels)

  creates an instance of zplane's reTune class
- static Error_t DestroyInstance (ReTuneIf ∗&pReTune)

  destroys an instance of the zplane's reTune class

4.1.1 Detailed Description

Definition at line 46 of file ReTuneIf.h.

4.1.2 Member Enumeration Documentation

Error_t `enum ReTuneIf::Error_t`

Enumerator

kNoError	no error occurred
kMemError	memory allocation failed
kInvalidFunctionParamError	one or more function parameters are not valid
kNotInitializedError	instance has not been initialized yet
kUnknownError	unknown error occurred
kNumErrors	

Definition at line 49 of file ReTuneIf.h.

```
50      {
51          kNoError,
52          kMemError,
53          kInvalidFunctionParamError,
54          kNotInitializedError,
55          kUnknownError,
56          kNumErrors
57      };
```

PitchClass `enum ReTuneIf::PitchClass`

Enumerator

C	
CSharp	
D	
DSharp	
E	
F	
FSharp	
G	
GSharp	
A	
ASharp	
B	
kNumOfPitchClasses	

Definition at line 68 of file ReTuneIf.h.

```
69      {
70          C,
71          CSharp,
72          D,
73          DSharp,
74          E,
75          F,
76          FSharp,
77          G,
78          GSharp,
79          A,
80          ASharp,
81          B,
82          kNumOfPitchClasses
83      };
```

Version_t `enum ReTuneIf::Version_t`

Enumerator

kMajor	

Enumerator

kMinor	
kPatch	
kRevision	
kNumVersionInts	

Definition at line 59 of file ReTuneIf.h.

```
60     {
61         kMajor,
62         kMinor,
63         kPatch,
64         kRevision,
65         kNumVersionInts
66     };
```

4.1.3 Member Function Documentation

CreateInstance() `static Error_t ReTuneIf::CreateInstance (`
`ReTuneIf *& pReTune,`
`int iOutputBlockSize,`
`float fSampleRate,`
`int iNumOfChannels ) [static]`

creates an instance of zplane's reTune class

Parameters

pReTune	: returns a pointer to the class instance
iOutputBlockSize	: desired max number of frames at the output, if not changed by GetFramesNeeded(.) this one is the default output block size, maximum is 1024 sample frames
fSampleRate	: input sample rate
iNumOfChannels	: number of channels (currently limited to 2)

Returns

ReTuneIf::Error_t : returns an error or noError

DestroyInstance() `static Error_t ReTuneIf::DestroyInstance (`
`ReTuneIf *& pReTune ) [static]`

destroys an instance of the zplane's reTune class

Parameters

pReTune	: pointer to the instance to be destroyed

Returns

ReTuneIf::Error_t : returns an error or noError

EnableShorterTransientMode() `virtual Error_t ReTuneIf::EnableShorterTransient↩ Mode ( bool bShouldBeEnabled ) [pure virtual]`

Enable mode for input audio that contains short transients (e.g. piano).

Parameters

bShouldBeEnabled	: enable flag

FlushBuffer() `virtual int ReTuneIf::FlushBuffer ( float ** ppfOutputBlock ) [pure virtual]`

gets the last frames in the internal buffer

Parameters

ppfOutputBlock	double pointer to the output buffer of samples [channels][samples]

Returns

int : returns the number of frames returned

GetBuildDate() `static const char* ReTuneIf::GetBuildDate ( ) [static]`

GetEnableEnvelopeCalculation() `virtual bool ReTuneIf::GetEnableEnvelope↩ Calculation ( ) const [pure virtual]`

returns if the envelope estimation is enabled

Returns

if envelope estimation is enabled or not

GetEnablePitchCorrection() `virtual bool ReTuneIf::GetEnablePitchCorrection ( PitchClass ePitchClass ) const [pure virtual]`

returns if pitch correction is en/disbaled for each pitch class

Parameters

ePitchClass	: the pitch class

Returns

bool : returns if it is en-/disbaled

GetEnvelopeShiftFactor() `virtual float ReTuneIf::GetEnvelopeShiftFactor ( ) const [pure virtual]`

returns the current envelope shift factor

Returns

the envelope shift factor

GetFramesNeeded() [1/2] `virtual int ReTuneIf::GetFramesNeeded ( ) [pure virtual]`

returns the sample frames needed for the next process call to complete

Returns

int : returns the number of sample frames

GetFramesNeeded() [2/2] `virtual int ReTuneIf::GetFramesNeeded ( int iNewOutputBlockSize ) [pure virtual]`

returns the sample frames needed for the next process call to complete for a new output blocksize

Parameters

iNewOutputBlockSize	: the new output blocksize

Returns

int : returns the number of sample frames

GetInputTuningFreq() `virtual float ReTuneIf::GetInputTuningFreq ( ) const [pure virtual]`

returns the current input tuning frequency

Returns

float : returns the current input tuning frequency

GetLatency() `virtual int ReTuneIf::GetLatency ( ) [pure virtual]`

returns the current latency of the algorithm

Returns

int : returns the latency in sample frames

GetMaxEnvelopeEstimationFreq() `virtual float ReTuneIf::GetMaxEnvelope↩ EstimationFreq ( ) const [pure virtual]`

returns the currently chosen max envelope estimation frequency

Returns

the max envelope estimation frequency in Hz

GetOffset() `virtual float ReTuneIf::GetOffset ( PitchClass ePitchClass ) const [pure virtual]`

returns the currently chosen offset for each pitch class

Parameters

ePitchClass	: the pitch class

Returns

float : returns the offset in semitones

GetOutputTuningFreq() `virtual float ReTuneIf::GetOutputTuningFreq ( ) const [pure virtual]`

returns the current output tuning frequency

Returns

float : returns the current output tuning frequency

GetPitchCorrection() `virtual float ReTuneIf::GetPitchCorrection ( ) const [pure virtual]`

returns the current pitch correction factor

Returns

float : returns the current pitch correction factor

GetSensitivity() `virtual float ReTuneIf::GetSensitivity ( ) const [pure virtual]`

returns the current sensitivity

Returns

float : returns the current sensitivity

GetSmoothing() `virtual float ReTuneIf::GetSmoothing ( ) const [pure virtual]`

returns the current smoothing speed

Returns

float : returns the current smoothing speed

GetTransients() `virtual float ReTuneIf::GetTransients ( ) const [pure virtual]`

returns the current transient value (between 0..1.5)

Returns

float : returns the current transient value

GetVersion() `static const char* ReTuneIf::GetVersion ( ) [static]`

ProcessData()
```
virtual int ReTuneIf::ProcessData (
    const float *const * ppfInputBlock,
    int iNumOfInputFrames,
    float ** ppfOutputBlock ) [pure virtual]
```

does the actual processing if the number of frames provided is as retrieved by ReTuneIf::GetFramesNeeded() this function always returns the number of frames as specified when calling ReTuneIf::CreateInstance(..)

Parameters

ppfInputBlock	: double pointer to the input buffer of samples [channels][samples]
iNumOfInputFrames	: the number of input frames
ppfOutputBlock	: double pointer to the output buffer of samples [channels][samples]

Returns

int : returns the number of frames returned

Reset() `virtual int ReTuneIf::Reset ( ) [pure virtual]`

clears the internal buffers of the reTune algorithm. you call this to avoid the remaining samples in the process buffer being audible when you e.g. stop playback and start it again at an other time position. other parameters (pitch offsets, sensitivity, etc.) are not reset.

Returns

int : currently returns 0

SetEnableEnvelopeCalculation() `virtual void ReTuneIf::SetEnableEnvelope↩ Calculation (`
`bool bEnable ) [pure virtual]`

enables envelope calculation and modification. if not needed it's better to disable as calculation takes a bit of time.

Parameters

bEnable	: enables envelope calculation and modification

SetEnablePitchCorrection() `virtual Error_t ReTuneIf::SetEnablePitchCorrection (`
`PitchClass ePitchClass,`
`bool bEnable ) [pure virtual]`

en-/disables the pitch correction for each pitch class

Parameters

ePitchClass	: the pitch class
bEnable	: enable or disable it

Returns

ReTuneIf::Error_t : returns an error or noError

SetEnvelopeShiftFactor() `virtual Error_t ReTuneIf::SetEnvelopeShiftFactor (`
`float fEnvelopeShiftFactor ) [pure virtual]`

sets the total shift factor for the envelope as factor. 1.o leaves the envelope in total untouched, 2.0 or 0.5 correspond to an octave up or down.

Parameters

fEnvelopeShiftFactor	: envelope shift factor

Returns

ReTuneIf::Error_t : returns an error or noError

SetInputTuningFreq() `virtual Error_t ReTuneIf::SetInputTuningFreq ( float fTuningFreq ) [pure virtual]`

sets the tuning frequency of the input audio, this is important to do a proper input frequency to pitch mapping

Parameters

fTuningFreq	: the input tuning frequency

Returns

ReTuneIf::Error_t : returns an error or noError

SetMaxEnvelopeEstimationFreq() `virtual Error_t ReTuneIf::SetMaxEnvelope↩EstimationFreq ( float fMaxEnvelopeEstimationFreq ) [pure virtual]`

sets the internal estimation order frequency, this one should be larger than the most prominent affected pitch. This should be well above the fundamental of the vocals or the main instrument. Needs some fiddling around.

Parameters

fMaxEnvelopeEstimationFreq	: estimation frequency (useful values 100.0-1800.0)

Returns

ReTuneIf::Error_t : returns an error or noError

SetOffset() `virtual Error_t ReTuneIf::SetOffset ( PitchClass ePitchClass, float fOffsetInSemitones ) [pure virtual]`

sets the pitch offset for each pitch class in semitones

Parameters

ePitchClass	: the pitch class
fOffsetInSemitones	: the offset in semitones

Returns

ReTuneIf::Error_t : returns an error or noError

SetOutputTuningFreq() `virtual Error_t ReTuneIf::SetOutputTuningFreq ( float fTuningFreq ) [pure virtual]`

sets the tuning frequency of the output audio

Parameters

fTuningFreq	: the output tuning frequency

Returns

ReTuneIf::Error_t : returns an error or noError

SetPitchCorrection() `virtual Error_t ReTuneIf::SetPitchCorrection ( float fPitchCorrectionFactor ) [pure virtual]`

sets the pitch correction for the enabled pitch classes. 0 means that no pitch correction is performed while 1 means that each pitch in an enabled pitch class is moved to the center pitch of the corresponding pitch class.

Parameters

fPitchCorrectionFactor	: pitch correction factor between (0..1)

Returns

ReTuneIf::Error_t : returns an error or noError

SetSensitivity() `virtual Error_t ReTuneIf::SetSensitivity ( float fSensitivity ) [pure virtual]`

sets how sensitive the algorithm should react to different f0 hypotheses (0..1). 0 means that it will be only take the strongest f0 hypothesis while higher values mean that more f0 hypotheses are taken as f0s - this may lead to too many f0s, usually 0.25 is a good value

default is 0.25

Parameters

fSensitivity	: the sensitivity (0..1)

Returns

ReTuneIf::Error_t : returns an error or noError

SetSmoothing() `virtual Error_t ReTuneIf::SetSmoothing ( float fSmooth ) [pure virtual]`

sets the smoothing speed for transitions between shifted notes. If a note is shifted the transition may be smoothed in order to avoid jumps - this is especially useful for vocals.

default is 46ms

Parameters

fSmooth	: smoothing speed in ms (0ms..300ms)

Returns

ReTuneIf::Error_t : returns an error or noError

SetTransients() `virtual Error_t ReTuneIf::SetTransients ( float fTransientVal ) [pure virtual]`

sets the transient behavior strength between (0..1.5). Values below 1.0 mean that the algorithm should attenuate transients while values above 1.0 mean that transients should be amplified. default is 1.0

Parameters

fTransientVal	: a value between 0..1.5

Returns

ReTuneIf::Error_t : returns an error or noError

The documentation for this class was generated from the following file:

- reTune/ReTuneIf.h

5 File Documentation

5.1 /work/project/docs/docugen.txt File Reference

5.1.1 Detailed Description

source documentation main file

5.2 reTune/ReTuneIf.h File Reference

Classes

- class ReTuneIf

Index