PRESTO - Preservation Technologies for European Broadcast Archives

IST-1999-20013

4.1 Audio Quality Monitor

DOCUMENT IDENTIFIER

DATE	January 29th, 2002
ABSTRACT	This document describes the audio quality monitors developed by ITC-irst in the PRESTO project. It also represents the user manual of the two software packages delivered.
AUTHOR, COMPANY	M. Cettolo, ITC-irst

INTERNAL REVIEWER

WORKPACKAGE / TASK task 4.1

DOCUMENT HISTORY

Release	Date	Reason of change	Status	Distribution
1.0	January 29th, 2002			

TABLE OF CONTENTS

1.Introduction

A labour-intensive task in preservation work is human monitoring of playback and recording equipment, to see if a signal is present and clear, and if the new digital signal matches the original signal. The goal of the task 4.1 was twofold:

1. to develop an automatic digitisation monitor - operating in software and on-line - able to provide the human operator with information useful for detecting possible problems of the digitisation process;
2. to implement an automatic quality monitor - operating in software and off-line - for the automatic evaluation of the quality of the signal, based on a set of parameters computed for the purpose. This "meta-data" can be exploited both during the ingestion stage - for example to take the decision that a restoration is needed - and during the future use of the archive - for example to select only items with a given quality.

The deliverable 4.1 consists of the present document and of the software prototypes of the two monitor stations: the two gzipped tar files online-AQM.tar.gz and offline-AQM.tar.gz contain, respectively, the on-line and the off-line audio quality monitors developed by ITC-irst within the WP4 of the Presto project.

The on-line monitor software package consists of a library with all the functionalities developed for audio monitoring. The library is compiled for the Linux OS, but all the source codes are provided, together with a Makefile, in such a way that compilations under other platforms are possible. A main program using the functions of the library is also given, both as source and executable code, in order to make easier the use of the library.

The off-line monitor software package is a stand-alone software station compiled for the Linux OS. Actually, it consists of a shell script running a set of executable codes whose data results are collected into a XML file. Only for the executable that uses the above audio library the source code is provided, since the rest of the package was not developed for the Presto project, but just re-used in it.

The two software packages are described in detail in the following sections.

2.On-line Quality Monitor

In this section, the software implementing the on-line quality monitor is described. First, instructions for installing and running it are provided. Then a brief overview of the classes is given, while the set of classes related to each algorithm is described in detail in the corresponding section. Finally, the description of the main program (file online.cpp) is given, since it represents a reference for the use of functionalities contained in the library.

2.1.Installation and Running

For opening the tar, put the online-AQM.tar.gz file into your working directory and, under a Unix/Linux OS, run the command:

tar -xvzf online-AQM.tar.gz

the online-AQM directory will be created, containing the following:

- Makefile: for the compilation
- src/*.(cpp|c): sources codes
- include/*.h: include files
- obj/linux/: for storing object codes
- bin/linux/online: executable code
- example/*.wav: sample wave test files
- lib/linux/libAudioInfo.a : ACS library for reading the header of wave audio files
- lib/include/*.h: header files of the library libAudioInfo.a
- lib/linux/libAudioQuality.a: library containing algorithms of the on-line audio quality monitor
- doc/man.(ps|pdf): this document in postscript and pdf formats

The executable online was built on a Pentium III Pc equipped with the Linux OS (version of the kernel 2.2.19; RedHat release 6.2.7). For whom interested in building their own version of the executable (on a different HW/OS platform, for example), the proper version of the library libAudioInfo.a (and its include files) are required. The Linux-compiled version of that library is stored in lib/linux/ (and include files in lib/include/*.h). Other versions are not available. However, for building the library libAudioQuality.a, containing all the functionalities of the audio quality monitor, libAudioInfo.a is not required.

The compilation of online can be performed through the following steps:

- check the value of the environment variable MACHINE; if needed, set it to the proper value:

  setenv MACHINE <machinename>

  (for C shell; for Bourne Shell, K Shell or Bash, use the export command)

- in case, create the proper directories for object and executable codes:

  mkdir bin/<machinename>

  mkdir obj/<machinename>

- run the compilation:

  make -f Makefile

The executable named online should have been created in bin/<machinename>/. For running it:

./bin/<machinename>/online -t wave -i ./example/GRR970404_0900.wav

If it properly works, it will print to the standard output and error some information about the processing, and will end after some seconds.

For building the library libAudioQuality.a (libAudioInfo.a is not required), you have to check if the directory

lib/<machinename>

already exists; in case, create it:

mkdir lib/<machinename>

and then run the compilation:

make -f Makefile audiolib

The library libAudioQuality.a, compiled under the current HW/OS platform, will be put into lib/<machinename>/.

2.2.Description of the Code

The code was developed by using the C++ language, then its description can be done by describing the classes implemented.

The classes can be split into two groups, those related to the Layer 1 (L1) of processing and those related to the Layer 2 (L2) of processing. The two layers correspond to different resolutions of the audio processing: at L1, processing is done for each input audio buffer coming from the acquisition stage; typically, the buffer size is of the order of fraction of second. The buffer size is usually defined by the user of the library; for example, the provided executable works on a buffer size set through the constant BUFFDIM (constants.h). At L2, processing is done on a queue of results from the short time computation done at L1; the size of the queue depends on the type of processing, however the number of elements of the queue typically ranges from an equivalent of tens of seconds to minutes of signal.

Figure 4.1. Processing flow through the class groups.

The classes are further grouped into data classes, that implement the structures for storing input and output data, and algo (algorithm) classes, that implement processing routines.

Figure 4.1 shows the processing flow through the class groups.

Broadly, the algorithms developed are

- Silence: for silence detection

- Saturation: for saturation detection
- Bandwidth: for bandwidth and spectrogram computation
- Features: for the computation of some features of the signal
- Click: for detection of clicks
- Correlation: for the computation of the time shift between channels

For each of them, the processing flow is similar to that depicted in Figure 4.1. In Tables 4.1 and 4.2, the classes involved in each algorithm are listed. They will be described in the following sections.

Algorithm	***DataInL1***	***AlgoL1***	***DataOutL1***
Silence	AudioDataSilenceInL1	AudioAlgoSilenceL1	AudioDataSilenceOut
Saturation	AudioDataPCM	AudioAlgoSaturationL1	AudioDataSaturationOut
Bandwidth	AudioDataPCM	AudioAlgoBandwidthL1	AudioDataBandwidthOutL1
Features	AudioDataFeaturesInL1	AudioAlgoFeaturesL1	AudioDataFeaturesOutL1
Click	AudioDataPCM	AudioAlgoClickL1	AudioDataClickOutL1
Correlation	AudioDataCorrInL1	AudioAlgoCorrelationL1	AudioDataCorrOutL1

Table 4.1 Classes of L1 algorithms.

Algorithm	***DataInL2***	***AlgoL2***	***DataOutL2***
Silence	AudioDataSilenceInL2	AudioAlgoSilenceL2	AudioDataSilenceOut
Saturation	AudioDataSaturationInL2	AudioAlgoSaturationL2	AudioDataSaturationOut
Bandwidth	AudioDataBandwidthInL2	AudioAlgoBandwidthL2	AudioDataBandwidthOutL2
Local Features	AudioDataFeatLocalInL2	AudioAlgoFeatLocalL2	AudioDataFeatLocalOutL2
Global Features	AudioDataFeatGlobalInL2	AudioAlgoFeatGlobalL2	AudioDataFeatGlobalOutL2
Click	AudioDataClickInL2	AudioAlgoClickL2	AudioDataClickOutL2
Correlation	AudioDataCorrInL2	AudioAlgoCorrelationL2	AudioDataCorrOutL2

Table 4.2 Classes of L2 algorithms.

2.3.Algorithm Parameters

Each algorithm needs some parameter to be set. They are defined in the classes AudioAlgoParams*, where also methods for setting and reading them are provided.

2.4.AudioDataPCM class

The class AudioDataPCM is used to store the portion of signal to be processed. The method:

```
void setSignal(int *buffer, size_t n, size_t offset);
```

allows to store the pointer to the buffer of n samples (integers). The offset gives the position of the first sample of the buffer with respect to the beginning of the processing (that is, how many samples have been processed so far).

Samples are stored as integers since the typical bit resolution is 24 bit. If the original signal contains more than one channel (e.g. a stereo signal), each channel has to be processed separately. The user of the library is in charge of the split.

The class represents the input of many algorithms at L1.

2.5.Silence

The set of classes named silence aims at individuating within the signal significant portions of silence. The detection algorithm is based on the computation of the variance of the signal, and on the value of the bandwidth. The meaning of "significant" portion, and the variance and bandwidth values are all referenced to threshold values, that have to be properly set. In the main program, reasonable threshold values are proposed.

2.5.1.AudioAlgoParamsSilence

In addition to the usual constructor and destructor class methods, the following methods are provided:

- void minSil(float min);
- void silThresh(double thresh);
- void sampleFrequency(int freq);

  for setting the parameter values of the silence processing

- float minSil() const;
- double silThresh() const;
- int sampleFrequency() const;

  for reading the parameter values of the silence processing

The first two parameters have the following meaning:

- minSil represents the minimum duration in seconds of silence that can be detected. If it equals the dimension of the input buffer, only one label (silence or not-silence) will be associated to the whole input buffer, that is no more refined segmentation will be done
- silThresh is the threshold value for silence/non-silence discrimination. The higher the silThresh is, the more care the function will be in classifying a given portion of signal as silence

The sampleFrequency is used only for number-of-samples/time conversions.

2.5.2.AudioDataSilenceInL1

In addition to the usual constructor and destructor class methods, the following methods are provided:

- void setSignal(AudioDataPCM *in);

  for setting the instance of the class with the current buffer of signal;

- void setSignal(int *bfr, size_t n, size_t os);

  for setting the instance of the class with the current buffer of signal (overloading of the previous method);

- void setBandwidth(AudioData* band);

for setting the instance of the class with the result of the bandwidth (L1) computation

2.5.3.AudioAlgoSilenceL1

In addition to the usual constructor and destructor class methods, the following methods are provided:

- void init(AudioAlgoParams* params);

  for communicating the parameter values (set through the AudioAlgoParamsSilence class) to the AudioAlgoSilenceL1 class objects

- int compute(AudioData* dataInput, AudioData* dataOutput);

  for detection of silences of the signal in dataInput; result stored in dataOutput

2.5.4.AudioDataSilenceOut

In addition to the usual constructor and destructor class methods, the following methods are provided:

- AudioDataSilenceOut&

  AudioDataSilenceOut::operator=(const AudioDataSilenceOut&);

  the copy operator (it copies the contents of the right operand into the left operand, allocating memory if necessary)

- std::string toString() const;

  for building a string with the contents of the instance (it can be then printed on any stream)

Moreover, for storing the computation results, the following (public) class members are provided:

- Segmentation *seg;
- int segN;
- size_t offset;

The members contain information about the segmentation in silence/non-silence segments of the audio buffer. Each element x=0...segN-1 of seg contains three useful data:

- seg[x].start: time index (int) corresponding to the start of the segment
- seg[x].end: time index (int) corresponding to the end of the segment
- seg[x].label: label (char) for silence/non-silence ('l'/'h').

Time indexes are computed by summing the value offset.

2.5.5.AudioDataSilenceInL2

In addition to the usual constructor and destructor class methods, the following methods are provided:

- std::string toString() const;

  for building a string with the contents of the instance (it can be then printed on any stream)

- std::deque<AudioDataSilenceOut*> buffer;

  a queue (http://www.sgi.com/tech/stl/Deque.html) for containing a sequence of instances of AudioDataSilenceOut

2.5.6.AudioAlgoSilenceL2

In addition to the usual constructor and destructor class methods, the following method is provided:

- int compute(AudioData* dataInput, AudioData* dataOutput);

  for merging all silences detected at L1 and stored in the queue dataInput; result stored in dataOutput

2.6.Saturation

The set of classes named saturation aims at individuating portions of saturated signal. The detection algorithm works in the time domain, for computational efficiency, and is based on the detection of long flat and high energy sequence of samples. The meaning of "long" and "high" is relative to threshold values. In the main program, reasonable threshold values for defining a signal to be flat are proposed.

2.6.1.AudioAlgoParamsSaturation

In addition to the usual constructor and destructor class methods, the following methods are provided:

- void minSat(float min);
- void flatInterval(int interval);
- void flatDelta(float delta);
- void sampleFrequency(int freq);

  for setting the parameter values of the saturation processing

- float minSat() const;
- int flatInterval() const;
- float flatDelta() const;
- int sampleFrequency() const;

  for reading the parameter values of the saturation processing

The first three parameters have the following meaning:

- minSat: if two consecutive saturation intervals are closer than minSat seconds, they are merged
- flatInterval: if the signal is flat for at least flatInterval samples, there it is considered saturated
- flatDelta: it is the maximum relative difference between the values of two consecutive samples for which the signal is considered flat

The sampleFrequency is used only for number-of-samples/time conversions.

2.6.2.AudioAlgoSaturationL1

In addition to the usual constructor and destructor class methods, the following methods are provided:

- void init(AudioAlgoParams* params);

  for communicating the parameter values (set through the AudioAlgoParamsSaturation class) to the AudioAlgoSaturationL1 class objects

- int compute(AudioData* dataInput, AudioData* dataOutput);

  for detection of saturations of the signal in dataInput; result stored in dataOutput

2.6.3.AudioDataSaturationOut

In addition to the usual constructor and destructor class methods, the following methods are provided:

- AudioDataSaturationOut&

  AudioDataSaturationOut::operator=(const AudioDataSaturationOut&);

  the copy operator (it copies the contents of the right operand into the left operand, allocating memory if necessary)

- std::string toString() const;

  for building a string with the contents of the instance (it can be then printed on any stream)

Moreover, for storing the computation results, the following (public) class members are provided:

- Segmentation *seg;
- int segN;
- size_t offset;

The members contain information about the segmentation in saturation/non-saturation segments of the audio buffer. Each element x=0...segN-1 of seg contains three useful data:

- seg[x].start : time index (int) corresponding to the start of the segment
- seg[x].end: time index (int) corresponding to the end of the segment
- seg[x].label: label (char) for saturation/non-saturation ('s'/'r').

Time indexes are computed by summing the value offset.

2.6.4.AudioDataSaturationInL2

In addition to the usual constructor and destructor class methods, the following methods are provided:

- std::string toString() const;

  for building a string with the contents of the instance (it can be then printed on any stream)

- std::deque<AudioDataSaturationOut*> buffer;

  a queue (http://www.sgi.com/tech/stl/Deque.html) for containing a sequence of instances of AudioDataSaturationOut

2.6.5.AudioAlgoSaturationL2

In addition to the usual constructor and destructor class methods, the following method is provided:

- int compute(AudioData* dataInput, AudioData* dataOutput);

  for merging all saturations detected at L1 and stored in the queue dataInput; result stored in dataOutput

2.7.Bandwidth

The set of classes named bandwidth aims at computing the band of the signal stored in the input buffer and the energy of sub-bands (histogram). All the processing is based on the output of the Fast Fourier Transform (FFT) of the input signal. Parameters of the FFT has to be set.

2.7.1.AudioAlgoParamsBandwidth

In addition to the usual constructor and destructor class methods, the following methods are provided:

- void fftWindow(int win);
- void fftStep(int step);
- void histoBandBlockN(int blockN);
- void sampleFrequency(int freq);

  for setting the parameter values of the bandwidth computation

- int fftWindow() const;
- int fftStep() const;
- int histoBandBlockN() const;
- int sampleFrequency() const;

  for reading the parameter values of the bandwidth computation

The first three parameters have the following meaning:

- fftWindow: number of samples of the signal on which the FFT is computed
- fftStep: since the FFT is computed on a fixed number of point (fftWindow), for covering all the input signal buffer it is necessary to compute a set of FFTs, by moving the analysis window from left to right. fftStep is the number of samples between the starting points of two consecutive analysis windows
- histoBandBlockN: it is the number of sub-bands for which the energy is computed
- sampleFrequency: it is the sample frequency of the digital audio signal

2.7.2.AudioAlgoBandwidthL1

In addition to the usual constructor and destructor class methods, the following methods are provided:

- void init(AudioAlgoParams* params);

  for communicating the parameter values (set through the AudioAlgoParamsBandwidth class) to the AudioAlgoBandwidthL1 class objects
- int compute(AudioData* dataInput, AudioData* dataOutput);

  for computing bandwidth and energy of sub-bands of the signal in dataInput; result stored in dataOutput

2.7.3.AudioDataBandwidthOutL1

In addition to the usual constructor and destructor class methods, the following methods are provided:

- AudioDataBandwidthOutL1&

  AudioDataBandwidthOutL1::operator=(const AudioDataBandwidthOutL1&);

  the copy operator (it copies the contents of the right operand into the left operand, allocating memory if necessary)
- std::string toString() const;

  for building a string with the contents of the instance (it can be then printed on any stream)

Moreover, for storing the computation results, the following (public) class members are provided:

- float *histoBand;
- int histoBandN;
- int bandwidth;
- size_t offset;

The members contain information about the bandwidth and energy of sub-bands:

- histoBand: it is a histoBandN-size vector; each element gives the energy (in dB) of one sub-band

- histoBandN: it is the number (set by the user through the method histoBandBlockN() of the class AudioAlgoParamsBandwidth; see section 2.7.1) of sub-bands
- bandwidth: it is the bandwidth of the input signal

Time indexes are computed by summing the value offset.

2.7.4.AudioDataBandwidthInL2

In addition to the usual constructor and destructor class methods, the following methods are provided:

- std::string toString() const;

  for building a string with the contents of the instance (it can be then printed on any stream)

- std::deque<AudioDataBandwidthOutL1*> buffer;

  a queue (http://www.sgi.com/tech/stl/Deque.html) for containing a sequence of instances of AudioDataBandwidthOutL1

2.7.5.AudioAlgoBandwidthL2

In addition to the usual constructor and destructor class methods, the following method is provided:

- int compute(AudioData* dataInput, AudioData* dataOutput);

  for computing bandwidth of the signal on the last n (user defined) signal buffers; result stored in dataOutput

2.7.6.AudioDataBandwidthOutL2

In addition to the usual constructor and destructor class methods, the following method is provided:

- std::string toString() const;

  for building a string with the contents of the instance (it can be then printed on any stream)

Moreover, for storing the computation result, the following (public) class member is provided:

- int bandwidth;

That member contains the value of the bandwidth of the signal under consideration.

2.8.Features

The set of classes named features aims at computing some important features of the signal, namely:

- power: estimated through the mean squared amplitude (dB)
- max peak: maximum sample value (dB)
- DC-offset: estimated through the mean of sample values
- dynamic: difference between the max peak and the noise floor (dB)
- SNR: difference between the current signal and the noise floor (dB)

At L1, a set of values is computed that allows either to give some of the above features directly at L1 or to compute them at L2 with different history lengths. Since some features are meaningful only if they are computed on long histories, at L2 two sets have been defined: the FeatLocal, which includes the power and the peak, and FeatGlobal, which includes DC-offset, dynamic and SNR. In particular, dynamic and SNR require the computation of the energy of the background noise: this is done by computing the energy of the silence segments individuated through the silence class, whose output is then used by algorithms for feature computation.

2.8.1.AudioDataFeaturesInL1

In addition to the usual constructor and destructor class methods, the following methods are provided:

- void setSignal(AudioDataPCM *in);

  for setting the instance of the class with the current buffer of signal;

- void setSignal(int *bfr, size_t n, size_t os);

  for setting the instance of the class with the current buffer of signal (overloading of the previous method);

- void setSilSegmentation(AudioData* segSil);

  for setting the instance of the class with the result of the silence (L1) computation

2.8.2.AudioAlgoFeaturesL1

In addition to the usual constructor and destructor class methods, the following method is provided:

- int compute(AudioData* dataInput, AudioData* dataOutput);

  for computing the features of the signal stored in dataInput, where silence segmentation is stored too; result given in dataOutput

2.8.3.AudioDataFeaturesOutL1

In addition to the usual constructor and destructor class methods, the following method is provided:

- std::string toString() const;

  for building a string with the contents of the instance (it can be then printed on any stream)

Moreover, for storing the computation results, the following (public) class members are provided:

- float sqrAmplitudeSignal;
- int sampleSignalN;
- float sqrAmplitudeNoise;
- int sampleNoiseN;
- float sampleSum;
- int sampleN;
- float samplePeak;

Those members contain information about the signal features:

- sqrAmplitudeSignal: sum of the square of sample values for the portion of non-silence signal
- sampleSignalN: number of samples of the previous sum
- sqrAmplitudeNoise: sum of the square of sample values for the portion of silence signal
- sampleNoiseN: number of samples of the previous sum
- sampleSum: sum of the square of sample values of the whole signal
- sampleN: number of samples of the previous sum
- samplePeak: max sample value

2.8.4.AudioDataFeatLocalInL2

In addition to the usual constructor and destructor class methods, the following methods are provided:

- std::string toString() const;

  for building a string with the contents of the instance (it can be then printed on any stream)

- std::deque<AudioDataFeaturesOutL1*> buffer;

  a queue (http://www.sgi.com/tech/stl/Deque.html) for containing a sequence of instances of AudioDataFeaturesOutL1

2.8.5.AudioAlgoFeatLocalL2

In addition to the usual constructor and destructor class methods, the following method is provided:

- int compute(AudioData* dataInput, AudioData* dataOutput);

  for computing the peak and the energy of the signal on the last n (user defined) signal buffers; result stored in dataOutput

2.8.6.AudioDataFeatLocalOutL2

In addition to the usual constructor and destructor class methods, the following method is provided:

- std::string toString() const;

  for building a string with the contents of the instance (it can be then printed on any stream)

Moreover, for storing the computation results, the following (public) class members are provided:

- float maxPeak;
- float meanLevel;

Those members contain information about the signal features:

- maxPeak: maximum sample value (dB)
- meanLevel: energy (mean squared amplitude) of the signal (dB)

2.8.7.AudioDataFeatGlobalInL2

In addition to the usual constructor and destructor class methods, the following methods are provided:

- std::string toString() const;

  for building a string with the contents of the instance (it can be then printed on any stream)

- std::deque<AudioDataFeaturesOutL1*> buffer;

  a queue (http://www.sgi.com/tech/stl/Deque.html) for containing a sequence of instances of AudioDataFeaturesOutL1

2.8.8.AudioAlgoFeatGlobalL2

In addition to the usual constructor and destructor class methods, the following method is provided:

- int compute(AudioData* dataInput, AudioData* dataOutput);

  for computing dynamic, SNR and DC-offset of the signal on the last n (user defined) signal buffers; result stored in dataOutput

2.8.9.AudioDataFeatGlobalOutL2

In addition to the usual constructor and destructor class methods, the following method is provided:

- std::string toString() const;

  for building a string with the contents of the instance (it can be then printed on any stream)

Moreover, for storing the computation results, the following (public) class members are provided:

- float dynamic;
- float SNR;
- float DCoffset;

Those members contain information about the signal features:

- DC-offset: mean of sample values
- dynamic: difference between the max peak and the energy (mean squared amplitude) of silences (dB)
- SNR: difference between the energy (mean squared amplitude) of non-silence signal and that of silence (dB)

2.9.Click

The set of classes named click aims at detecting within the signal significant clicks. The detection algorithm is based on the linear prediction of each sample on the basis of the previous samples. If the predicted value differs from the actual sample for more than a given threshold, the actual sample is considered an outlier, that is a click. The linear prediction is based on a set of coefficients computed through a maximum entropy method. This is the heaviest algorithm of the package. A number of parameters has to be set. In the main program, reasonable values are proposed.

2.9.1.AudioAlgoParamsClick

In addition to the usual constructor and destructor class methods, the following methods are provided:

- void clickNdata(int Ndata);
- void clickNpoles(int Npoles);
- void clickThresh(int Thresh);
- void minClickDelta(int Delta);

  for setting the parameter values of the click detection

- int clickNdata() const;
- int clickNpoles() const;
- int clickThresh() const;
- int minClickDelta() const;

  for reading the parameter values of the click detection

The parameters have the following meaning:

- clickNdata: number of samples used for computing coefficients
- clickNpoles: number of coefficients
- clickThresh: threshold for considering a sample an outlier or not

- minClickDelta: if two consecutive clicks are closer than minClickDelta samples, one is discarded (deleted)

2.9.2.AudioAlgoClickL1

In addition to the usual constructor and destructor class methods, the following methods are provided:

- void init(AudioAlgoParams* params);

  for communicating the parameter values (set through the AudioAlgoParamsClick class) to the AudioAlgoClickL1 class objects

- int compute(AudioData* dataInput, AudioData* dataOutput);

  for detection of clicks in the signal stored in dataInput; result in dataOutput

2.9.3.AudioDataClickOutL1

In addition to the usual constructor and destructor class methods, the following methods are provided:

- AudioDataClickOutL1&

  AudioDataClickOutL1::operator=(const AudioDataClickOutL1&);

  the copy operator (it copies the contents of the right operand into the left operand, allocating memory if necessary)

- std::string toString() const;

  for building a string with the contents of the instance (it can be then printed on any stream)

Moreover, for storing the computation results, the following (public) class members are provided:

- ClickType clickIdx;
- size_t offset;

The structure clickIdx is of type ClickType and contains information about the click detection:

- clickIdx.clickN: it is an int indicating how many clicks have been detected in the input buffer; it represents the size of the two following vectors too
- clickIdx.timeidx: pointer to a vector of integers; each element of the vector represents the time index of one click, with respect to offset. In other words, clickIdx.timeidx[k] is the absolute time of the (k+1)_th click detected inside the input buffer
- clickIdx.deleted: pointer to a vector of chars; each element of the vector represents a flag indicating if the click stored at the same position in the vector clickIdx.timeidx is to be onsidered kept or deleted; the values of the flag in the two cases are FALSE (0) and TRUE (1), respectively

Time indexes are computed by summing the value offset.

2.9.4.AudioDataClickInL2

In addition to the usual constructor and destructor class methods, the following methods are provided:

- std::string toString() const;

  for building a string with the contents of the instance (it can be then printed on any stream)

- std::deque<AudioDataClickOutL1*> buffer;

  a queue (http://www.sgi.com/tech/stl/Deque.html) for containing a sequence of instances of AudioDataClickOutL1

2.9.5.AudioAlgoClickL2

In addition to the usual constructor and destructor class methods, the following method is provided:

- int compute(AudioData* dataInput, AudioData* dataOutput);

  for computing the click rate on the basis of the queue in dataInput; result stored in dataOutput

2.9.6.AudioDataClickOutL2

In addition to the usual constructor and destructor class methods, the following method is provided:

- std::string toString() const;

for building a string with the contents of the instance (it can be then printed on any stream)

Moreover, for storing the computation results, the following (public) class member is provided:

- float clickRate;

It represents the rate (per million of samples) of (kept) clicks (outlier samples) with respect to the total number of samples.

2.10.Correlation

The set of classes named correlation aims at computing the time shift between the left and right channel. The algorithm is based on the discrete correlation theorem. Since the requirement was to find shifts with a resolution higher than one sample, an oversampling is necessary. The main steps of the algorithm are:

- FFT the two data sets (channels)
- one resulting transform times the complex conjugate of the other
- inverse transform the product => correlation function (max)
- oversampling correlation function

➓find the new maximum

The position of the final maximum allows to estimate the time shift between the channels. A number of parameters has to be set. In the main program, reasonable values are proposed.

In order to avoid the computation of correlation between silence signals, the algorithms exploit the segmentation in terms of silence/non-silence individuated through the silence class.

2.10.1.AudioAlgoParamsCorrelation

In addition to the usual constructor and destructor class methods, the following methods are provided:

➓void corrPointN(int pointN);

for setting the parameter value of the correlation computation

➓int corrPointN() const;

for reading the parameter value of the correlation computation

The parameter sets the number of samples actually used to compute the channel correlation. It has to be not greater than the size of the input buffer.

2.10.2.AudioDataCorrInL1

In addition to the usual constructor and destructor class methods, the following methods are provided:

➓void setSignal(int *bufferL, int *bufferR, size_t n, size_t offset);

for setting the instance of the class with the current buffers (left and right channels) of signal;

➓void setSilSegmentation(AudioData* segSilL, AudioData* segSilR);

for setting the instance of the class with the result of the silence computation on the two channels

2.10.3.AudioAlgoCorrelationL1

In addition to the usual constructor and destructor class methods, the following methods are provided:

➓void init(AudioAlgoParams* params);

for communicating the parameter values (set through the AudioAlgoParamsCorrelation class) to the AudioAlgoCorrelationL1 class objects

➓int compute(AudioData* dataInput, AudioData* dataOutput);

for computing the correlation between the two channels; result in dataOutput

2.10.4.AudioDataCorrOutL1

In addition to the usual constructor and destructor class methods, the following method is provided:

➓std::string toString() const;

for building a string with the contents of the instance (it can be then printed on any stream)

Moreover, for storing the computation results, the following (public) class members are provided:

➓int phase;

➓int offset;

The member phase gives the time shift between the two channels. The value is given in degrees, with respect to time shifts between two 10KHz tones: since the sample frequency is 48KHz, 1-sample-shift corresponds to 75 degrees.

The member offset has the usual meaning.

2.10.5.AudioDataCorrInL2

In addition to the usual constructor and destructor class methods, the following methods are provided:

➓std::string toString() const;

for building a string with the contents of the instance (it can be then printed on any stream)

➓std::deque<AudioDataCorrOutL1*> buffer;

a queue (http://www.sgi.com/tech/stl/Deque.html) for containing a sequence of instances of AudioDataCorrOutL1

2.10.6.AudioAlgoCorrelationL2

In addition to the usual constructor and destructor class methods, the following method is provided:

➓int compute(AudioData* dataInput, AudioData* dataOutput);

for computing the mean of the time shift between channels of the signal on the last n (user defined) signal buffers; result stored in dataOutput

2.10.7.AudioDataCorrOutL2

In addition to the usual constructor and destructor class methods, the following method is provided:

➓std::string toString() const;

for building a string with the contents of the instance (it can be then printed on any stream)

Moreover, for storing the computation result, the following (public) class member is provided:

➓int phase;

that gives the mean of the time shift between the two channels computed on the given history.

2.11.The Main program

The main program is included in the file online.cpp. Since it represents a reference for the use of functionalities contained in the library, here it is described in detail.

After the reading of the wave input audio file header and some checks, the flow of the program is:

1. for each algorithm (silence, saturation...) do
    2. set algorithm parameters
    3. declare objects used by the computation
    4. initialize the algorithm class with the algorithm parameters
    5. for each input audio buffer do
        6. for each algorithm (silence, saturation...) do
            7. prepare input to L1 algorithm
            8. run L1 algorithm
            9. print result to standard output
            10. put result into the queue
            11. check the length of the queue: if necessary, pop the oldest element
            12. run the L2 algorithm on the queue
            13. print result to standard output

Let focus our attention to the silence algorithm. The use of the other algorithms is very similar to that, then the comments in the source code should be enough for understanding their use.

step 2.

```
AudioAlgoParamsSilence parSil;
parSil.minSil(0.2);
parSil.silThresh(1e4);
parSil.sampleFrequency(header.frequency);
```

Here, the object parSil of the class AudioAlgoParamsSilence is declared, and then the parameters for silence detection are set through the proper methods.

step 3.

```
AudioAlgoSilenceL1 silL1;
AudioAlgoSilenceL2 silL2;
AudioDataSilenceInL1 inSilL1;
AudioDataSilenceOut outSilL1_L, outSilL1_R, outSilL2;
AudioDataSilenceInL2 inSilL2;
outSilL1_L.seg=new Segmentation[BUFSIZ];
outSilL1_R.seg=new Segmentation[BUFSIZ];
```

outSilL2.seg=new Segmentation[BUFSIZ];

Here, a set of objects of the Data and Algo silence classes are declared. Moreover, the allocation of memory for storing results is done. It is assumed that the value BUFSIZ will be greater than the maximum number of silence/non-silence segments that will be detected. Note that the input buffer size (in number of samples) is fixed and known, hence the maximum number of segments could be calculated.

step 4.

silL1.init(&parSil);

Here, the parameters for silence detection (stored into the parSil object) are communicated to the object silL1 of the AudioAlgoSilenceL1 through the proper method.

step 5.

inDataL1.setSignal(InData[channel], sampleN, offset);

Here, it is assumed that an object inDataL1 of the class AudioDataPCM has been declared, that InData[channel] is a pointer to an array of integers containing the portion of the signal to be processed, and that the variables sampleN and offset are set to the proper values. The method setSignal() store the input audio buffer into the object inDataL1.

step 7.

outSilL1_L.segN=0; (outSilL1_R.segN=0;)

inSilL1.setSignal(&inDataL1);

inSilL1.setBandwidth(&outBandL1);

Here, the input for the L1 algorithm is prepared: first, the number of segments of the left (right) channel is reset, then the audio data and the result of the bandwidth computation at L1 are communicated to the object inSilL1 through the proper methods.

step 8.

silL1.compute(&inSilL1, &outSilL1_L); (silL1.compute(&inSilL1, &outSilL1_R);)

Here, the L1 algorithm is run on the left (right) data; result is stored into the object outSilL1_L (outSilL1_R).

step 9.

printf("%s", outSilL1_L.toString().c_str()); (printf("%s", outSilL1_R.toString().c_str());)

Here, result of L1 algorithm is printed to the standard output stream. However, the method toString() of the AudioDataSilenceOut puts the result into a string that then can be sent to any stream.

step 10.

AudioDataSilenceOut* tmpSil;

tmpSil = new AudioDataSilenceOut;

(*tmpSil) = outSilL1_L;

inSilL2.buffer.push_back(tmpSil);

Here, result of L1 algorithm is put into the queue. First, a pointer to the class AudioDataSilenceOut is declared and an object allocated. Then, the result of L1 algorithm is copied into the new element which is inserted into the queue.

step 11.

```
if (inSilL2.buffer.size()>history4L2)
  tmpSil = inSilL2.buffer.front();
  inSilL2.buffer.pop_front();
  delete tmpSil;
```

Here, the length of the queue is checked: if it has more elements than a given value (history4L2), the oldest element is removed (and memory deallocated).

step 12.

```
silL2.compute(&inSilL2,  &outSilL2);
```

Here, the L2 algorithm is run on the current queue of L1 results. Result is stored into the object outSilL2.

step 13.

```
printf("%s", outSilL2.toString().c_str());
```

Here, result of L2 algorithm is printed to the standard output stream through the method toString() of the AudioDataSilenceOut (see step 9).

3.Off-line Quality Monitor

In this section, the off-line quality monitor is described. First, instructions for installing and running it are provided. Then, the metadata stored into the XML output file are described.

3.1.Installation and Running

For opening the tar, put the offline-AQM.tar.gz file into your working directory and, under a Unix/Linux OS, run the command:

tar -xvzf offline-AQM.tar.gz

the offline-AQM directory will be created, containing, among the other things, the shell script

AQmonitor-offline.sh

that can be used to run the monitor. In the same directory, the README file gives essential instructions for the running.

All the executables (in the sub directory bin/linux/) were built on a Pentium III Pc equipped with the Linux OS (version of the kernel 2.2.19; RedHat release 6.2.7). Only for the executable offline all the source codes are provided in the online-AQM.tar.gz (see Section 2). There, you can run the compilation by the command:

make -f Makefile offline

that puts into the subdirectory bin/<machinename>/ the executable offline. All the other executables have not been developed within the Presto project, hence source codes are not provided.

The executable bin/linux/buildOfflineXMLrep, that writes the XML file starting from the metadata computed by the set of signal processors, uses the library lib/linux/ibxerces-c1_6_0.so developed by the Apache Software Foundation. The Apache Software License is given in the LICENSE.txt file.

Before running the monitor, some environment variables have to be properly set:

- MACHINE: be sure that its value is linux (and that the OS on which the monitor is going to run is a Linux OS)

- LOCALPATH: it has to be set to the directory where the monitor has been installed, that is where the shell script AQmonitor-offline.sh is stored. Example:

  setenv LOCALPATH /rockette0/ssi/cettolo/projects/presto/system/toRAI_1.0/

- WORKPATH: it is the working directory; in it, the result (an XML file) will be stored too. It has to have a subdirectory named preprocessing. Example:

  setenv WORKPATH

  /rockette0/ssi/cettolo/projects/presto/system/toRAI_1.0/tmp/

  ($WORKPATH/preprocessing must exist!)

- LD_LIBRARY_PATH. Since the program that builds the final XML file is based on a dynamic library (developed by the Apache Software Foundation), it is necessary to add

to the variable LD_LIBRARY_PATH (on Linux and Solaris; LIBPATH on AIX, SHLIB_PATH on HP-UX) the path of the library, that is stored in $LOCALPATH/lib/linux/

```
setenv LD_LIBRARY_PATH "$LOCALPATH/lib/$MACHINE/:$LD_LIBRARY_PATH"
(export LD_LIBRARY_PATH=$LOCALPATH/lib/$MACHINE/:$LD_LIBRARY_PATH)
```

The monitor is executed by the command:

```
sh AQmonitor-offline.sh $WORKPATH/<filename>.xml
```

At the end of the processing, the result will be stored in

```
$WORKPATH/<filename>.xml
```

It is possible to specify an optional parameter "-initfile":

```
sh AQmonitor-offline.sh $WORKPATH/<filename>.xml -initfile <float>
```

It represents the number of seconds of initial silence/noise that can be hypothesized in the audio document. The interval is used in order to estimate the energy of the background silence/noise, instead of using defalut values for that computation.

3.2.Metadata Description

An example of a XML file written by the off-line monitor station is:

```
<audioMetadata>
  <summary>
    <azimuth degrees="3"/>
    <sampleFrequency Hz="48000"/>
    <leftChannel>
        <bandwidth Hz="22095"/>
        <dynamic dB="42.8"/>
        <SNR dB="28.2"/>
        <DC-offset sampleMean="-141.3"/>
        <peak dB="-15.0"/>
        <energy dB="-29.7"/>
        <clicks perMillionOfSamples="1.0"/>
        <silence percentage="1.5"/>
        <saturation percentage="0.0"/>
    </leftChannel>
    <rightChannel>
        (same metadata of the left channel)
    </rightChannel>
  </summary>
```

```
    <segmentation>
      <leftChannel>
        <segment type="silence" endTime="14560" startTime="0"/>
        <segment type="music" endTime="185760" startTime="14560"/>
        <segment type="noise" endTime="260960" startTime="185760"/>
        <segment type="speech" gender="male" endTime="515360"
                    bandwidth="wide" startTime="260960"/>
        <segment type="noise" endTime="960000" startTime="515360"/>
       </leftChannel>
       <rightChannel>
            (same metadata of the left channel>
      </rightChannel>
    </segmentation>
</audioMetadata>
```

There are two main sections, summary and segmentation. The summary section contains the values computed by the L2 algorithms (described in Section 2) on the whole audio document, that is on the queues with all the elements. For what concern the silences and saturations, in this section the percentages of those phenomena computed over all the input audio audio are given. The segmentation section contains the time segmentation (in samples) and segment classification of the audio stream; homogeneous (from an acoustic point of view) segments are classified in terms of the following audio classes:

- silence
- noise
- music
- speech:
    - gender (female/male)
    - bandwidth (narrow/wide)

In each audio segment, saturation intervals are also specified as subsegments.

Each time index corresponds to a sample number.