



open-Source Media Interpretation by Large feature-space Extraction

Version 2.1, December 2015

Authors: Florian Eyben, Felix Weninger, Martin Wöllmer, Björn Schuller

E-mails: fe, fw, mw, bs at audeering.com

Copyright (C) 2013-2014 by
audEERING UG (haftungsbeschränkt)

Copyright (C) 2008-2013 by
TU München, MMK



audEERING UG (haftungsbeschränkt)
D-82205 Gilching, Germany
<http://www.audeering.com/>

The official openSMILE homepage can be found at: <http://opensmile.audeering.com/>

This documentation was written by Florian Eyben.

Contents

1	About openSMILE	5
1.1	What is openSMILE?	5
1.2	Who needs openSMILE?	6
1.3	Licensing	6
1.4	History	6
1.5	Capabilities - Overview	7
2	Using openSMILE	13
2.1	Obtaining and Installing openSMILE	13
2.2	Compiling the openSMILE source code	14
2.2.1	Build instructions for the impatient	14
2.2.2	Compiling on Linux/Mac	16
2.2.3	Compiling on Linux/Mac with PortAudio	19
2.2.4	Compiling on Linux with openCV and portaudio support.	19
2.2.5	Compiling on Windows	20
2.2.6	Compiling on Windows with PortAudio	21
2.2.7	Compiling on Windows with openCV support.	21
2.3	Extracting your first features	22
2.4	What is going on inside of openSMILE	26
2.4.1	Incremental processing	27
2.4.2	Smile messages	29
2.4.3	openSMILE terminology	29
2.5	Default feature sets	30
2.5.1	Chroma features	31
2.5.2	MFCC features	31
2.5.3	PLP features	32
2.5.4	Prosodic features	32
2.5.5	Extracting features for emotion recognition	33
2.6	Using Portaudio for live recording/playback	38
2.7	Extracting features with openCv	38
2.8	Visualising data with Gnuplot	39
3	Reference section	43
3.1	General usage - SMILExtract	43
3.2	Understanding configuration files	45
3.2.1	Enabling components	45
3.2.2	Configuring components	46
3.2.3	Including other configuration files	47
3.2.4	Linking to command-line options	47

3.2.5	Defining variables	47
3.2.6	Comments	47
3.3	Component description	48
3.3.1	cComponentManager	49
3.3.2	Basic data memory and interface components	49
3.3.3	Data sources	52
3.3.4	Data sinks	60
3.3.5	Live data sinks (classifiers)	69
3.3.6	Low-level features and signal processing	70
3.3.7	Functionals	152
3.4	Feature names	164
4	Developer's Documentation	167
5	Additional Support	169
6	Acknowledgement	171

Chapter 1

About openSMILE

We start introducing openSMILE by addressing two important questions for users who are new to openSMILE : *What is openSMILE ?* and *Who needs openSMILE ?*. If you want to start using openSMILE right away, then you should start reading section 2, or section 2.3 if you have already managed to install openSMILE.

1.1 What is openSMILE?

The Munich open-Source Media Interpretation by Large feature-space Extraction (openSMILE) toolkit is a modular and flexible feature extractor for signal processing and machine learning applications. The primary focus is clearly put on audio-signal features. However, due to their high degree of abstraction, openSMILE components can also be used to analyse signals from other modalities, such as physiological signals, visual signals, and other physical sensors, given suitable input components. It is written purely in C++, has a fast, efficient, and flexible architecture, and runs on various main-stream platforms such as Linux, Windows, and MacOS. openSMILE is designed for real-time online processing, but can also be used off-line in batch mode for processing of large data-sets. This is a feature rarely found in related feature extraction software. Most of related projects are designed for off-line extraction and require the whole input to be present. openSMILE can extract features incrementally as new data arrives. By using the PortAudio¹ library, openSMILE features platform independent live audio input and live audio playback, which enabled the extraction of audio features in real-time.

To facilitate interoperability, openSMILE supports reading and writing of various data formats commonly used in the field of data mining and machine learning. These formats include PCM WAVE for audio files, CSV (Comma Separated Value, spreadsheet format) and ARFF (Weka Data Mining) for text-based data files, HTK (Hidden-Markov Toolkit) parameter files, and a simple binary float matrix format for binary feature data.

Using the open-source software gnuplot², extracted features which are dumped to files can be visualised. A strength of openSMILE, due to its highly modular architecture is that almost all intermediate data which is generated during the feature extraction process (such as windowed audio data, spectra, etc.) can be accessed and saved to files for visualisation or further processing.

¹<http://www.portaudio.com>

²<http://www.gnuplot.info/>

1.2 Who needs openSMILE?

openSMILE is intended to be used for research applications, demonstrators, and prototypes in the first place. Thus, the target group of users is researchers and system developers. Due to its compact code and modular architecture, using openSMILE for the final product is also considerable. However, we would like to stress that openSMILE is distributed under a research only license (see the next section for details).

Currently, openSMILE is used by researchers and companies all around the world, which are working in the field of speech recognition (feature extraction front-end, keyword spotting, etc.), the area of affective computing (emotion recognition, affect sensitive virtual agents, etc.), and Music Information Retrieval (chord labelling, beat tracking, onset detection etc.). With the 2.0 open-source release we target the wider multi-media community by including the popular openCV library for video processing and video feature extraction.

1.3 Licensing

openSMILE follows a dual-licensing model. Since the main goal of the project is a widespread use of the software to facilitate research in the field of machine learning from audio-visual signals, the source code and the binaries are freely available for private, research, and educational use under an open-source license. It is not allowed to use the open-source version of openSMILE for or in any sort of commercial product. Fundamental research in companies, for example, is permitted, but if a product is the result of the research, we require you to buy a commercial development license. Contact us at info@audeering.com (or visit us at <http://www.audeering.com>) for further information.

1.4 History

openSMILE was originally created in the scope of the European EU-FP7 research project SEMAINE (<http://www.semaine-project.eu>) and is used there as the acoustic emotion recognition engine and keyword spotter in a real-time affective dialogue system. To serve the research community open-source releases of openSMILE were made independently of the main project's code releases.

The first publicly available version of openSMILE was contained in the first Emotion and Affect recognition toolkit openEAR as the feature extraction core. openEAR was introduced at the Affective Computing and Intelligent Interaction (ACII) conference in 2009. One year later the first independent release of openSMILE (version 1.0.0) was made, which aimed at reaching a wider community of audio analysis researchers. It was presented at ACM Multimedia 2010 in the Open-Source Software Challenge. This first release was followed by a small bugfix release (1.0.1) shortly. Since then development has taken place in the subversion repository on sourceforge. Since 2011 the development was continued in a private repository due to various internal and third party project licensing issues.

openSMILE 2.0 (rc1) is the next major release after the 1.0.1 version and contains the latest code of the core components with a long list of bugfixes, new components as well as improved old components, extended documentation, a restructured source tree and new major functionality such as a multi-pass mode and support for synchronised audio-visual feature extraction based on openCV.

Version 2.1 contains further fixes, improved backwards compatibility of the standard INTERSPEECH challenge parameter sets, support for reading JSON neural network files created

with the CURRENNT toolkit, a F0 harmonics component, and a fast fast linear SVM sink component for integrating models trained with WEKA SMO, as well as some other minor new components and features. It is the first version published and supported by audEERING.

1.5 Capabilities - Overview

This section gives a brief summary on openSMILE's capabilities. The capabilities are distinguished by the following categories: data input, signal processing, general data processing, low-level audio features, functionals, classifiers and other components, data output, and other capabilities.

Data input: openSMILE can read data from the following file formats:

- RIFF-WAVE (PCM) (for MP3, MP4, OGG, etc. a converter needs to be used)
- Comma Separated Value (CSV)
- HTK parameter files
- WEKA's ARFF format.
- Video streams via openCV.

Additionally, live recording of audio from any PC sound-card is supported via the PortAudio library. For generating white noise, sinusoidal tones, and constant values a signal Generator is provided.

Signal Processing: The following functionality is provided for general signal processing or signal pre-processing (prior to feature extraction):

- Windowing-functions (Rectangular, Hamming, Hann (raised cosine), Gauss, Sine, Triangular, Bartlett, Bartlett-Hann, Blackmann, Blackmann-Harris, Lanczos)
- Pre-/De-emphasis (i.e. 1st order high/low-pass)
- Re-sampling (spectral domain algorithm)
- FFT (magnitude, phase, complex) and inverse
- Scaling of spectral axis via spline interpolation (open-source version only)
- dbA weighting of magnitude spectrum
- Autocorrelation function (ACF) (via IFFT of power spectrum)
- Average magnitude difference function (AMDF)

Data Processing: openSMILE can perform a number of operations for feature normalisation, modification, and differentiation:

- Mean-Variance normalisation (off-line and on-line)
- Range normalisation (off-line and on-line)
- Delta-Regression coefficients (and simple differential)

- Weighted Differential as in [?]
- Various vector operations: length, element-wise addition, multiplication, logarithm, and power.
- Moving average filter for smoothing of contour over time.

Audio features (low-level): The following (audio specific) low-level descriptors can be computed by openSMILE :

- Frame Energy
- Frame Intensity / Loudness (approximation)
- Critical Band spectra (Mel/Bark/Octave, triangular masking filters)
- Mel-/Bark-Frequency-Cepstral Coefficients (MFCC)
- Auditory Spectra
- Loudness approximated from auditory spectra.
- Perceptual Linear Predictive (PLP) Coefficients
- Perceptual Linear Predictive Cepstral Coefficients (PLP-CC)
- Linear Predictive Coefficients (LPC)
- Line Spectral Pairs (LSP, aka. LSF)
- Fundamental Frequency (via ACF/Cepstrum method and via Subharmonic-Summation (SHS))
- Probability of Voicing from ACF and SHS spectrum peak
- Voice-Quality: Jitter and Shimmer
- Formant frequencies and bandwidths
- Zero- and Mean-Crossing rate
- Spectral features (arbitrary band energies, roll-off points, centroid, entropy, maxpos, minpos, variance (=spread), skewness, kurtosis, slope)
- Psychoacoustic sharpness, spectral harmonicity
- CHROMA (octave warped semitone spectra) and CENS features (energy normalised and smoothed CHROMA)
- CHROMA-derived Features for Chord and Key recognition
- F0 Harmonics ratios

Video features (low-level): The following video low-level descriptors can be currently computed by openSMILE , based on the openCV library:

- HSV colour histograms
- Local binary patterns (LBP)
- LBP histograms
- Optical flow and optical flow histograms
- Face detection: all these features can be extracted from an automatically detected facial region, or from the full image.

Functionals: In order to map contours of audio and video low-level descriptors onto a vector of fixed dimensionality, the following functionals can be applied:

- Extreme values and positions
- Means (arithmetic, quadratic, geometric)
- Moments (standard deviation, variance, kurtosis, skewness)
- Percentiles and percentile ranges
- Regression (linear and quadratic approximation, regression error)
- Centroid
- Peaks
- Segments
- Sample values
- Times/durations
- Onsets/Offsets
- Discrete Cosine Transformation (DCT)
- Zero-Crossings
- Linear Predictive Coding (LPC) coefficients and gain

Classifiers and other components: Live demonstrators for audio processing tasks often require segmentation of the audio stream. openSMILE provides voice activity detection algorithms for this purpose, and a turn detector. For incrementally classifying the features extracted from the segments, Support Vector Machines are implemented using the LibSVM library.

- Voice Activity Detection based on Fuzzy Logic
- Voice Activity Detection based on LSTM-RNN with pre-trained models
- Turn-/Speech-segment detector
- LibSVM (on-line)

- SVM sink (for loading linear kernel WEKA SMO models)
- GMM (experimental implementation from eNTERFACE’12 project, to be release soon)
- LSTM-RNN (Neural Network) classifier which can load RNNLIB and CURRENNT nets
- Speech Emotion recognition pre-trained models (openEAR)

Data output: For writing data data to files, the same formats as on the input side are supported, except for an additional binary matrix format:

- RIFF-WAVE (PCM uncompressed audio)
- Comma Separated Value (CSV)
- HTK parameter file
- WEKA ARFF file
- LibSVM feature file format
- Binary float matrix format

Additionally, live audio playback is supported via the Portaudio library.

Other capabilites : Besides input, signal processing, feature extraction, functionals, and output components, openSMILE comes with a few other capabilites (to avoid confusion, we do not use the term ‘features’ here), which make using openSMILE easy and versatile:

Multi-threading Independent components can be run in parallel to make use of multiple CPUs or CPU cores and thus speed up feature extraction where time is critical.

Plugin-support Additional components can be built as shared libraries (or DLLs on windows) linked against openSMILE’s core API library. Such plugins are automatically detected during initialisation of the program, if they are placed in the `plugins` subdirectory.

Extensive logging Log messages are handled by a `smileLogger` component, which currently is capable of saving the log messages to a file and printing them to the standard error console. The detail of the messages can be controlled by setting the log-level. For easier interpretation of the messages, the types *Message* (MSG), *Warning* (WRN), *Error* (ERR), and *Debug* (DBG) are distinguished.

Flexible configuration openSMILE can be fully configured via one single text based configuration file. This file is kept in a simple, yet very powerful, property file format. Thereby each component has it’s own section, and all components can be connected via their link to a central data memory component. The configuration file even allows for defining custom command-line options (e.g. for input and output files), and including other configuration files to build configurations with modular configuration blocks. The name of the configuration file to include can even be specified on the commandline, allowing maximum flexibility in scripting.

Incremental processing All components in openSMILE follow strict guidelines to meet the requirements of incremental processing. It is not allowed to require access to the full input sequence and seek back and forth within the sequence, for example. Principally each

component must be able to process its data frame by frame or at least as soon as possible. Some exceptions to this rule have been granted for components which are only used during off-line feature extraction, such as components for overall mean normalisation.

Multi-pass processing in batch mode For some tasks multi-pass processing is required, which obviously can only be applied in off-line (or buffered) mode. openSMILE since version 2.0 supports mutli-pass processing for all existing components.

TCP/IP network support In the commercial version a remote data I/O API is available, which allows to send and receive data (features and messages) from openSMILE, as well as remote control of openSMILE (pause/resume/restart and send config) via a TCP/IP network connection.

Chapter 2

Using openSMILE

Now we describe how to get started with using openSMILE . First, we will explain how to obtain and install openSMILE . If you already have a working installation of openSMILE you can skip directly to section 2.3, where we explain how to use openSMILE for your first feature extraction task. We then give an overview on openSMILE’s architecture in section 2.4. This will help you to understand what is going on inside openSMILE , and why certain things are easy, while others may be tricky to do. Next, to make full use of openSMILE’s capabilities it is important that you learn to write openSMILE configuration files. Section 3.2 will explain all necessary aspects. Finally, to make your life a bit simpler and to provide common feature sets to the research community for various tasks, some example configuration files are provided. These are explained in section 2.5. Included are all the baseline feature sets for the INTERSPEECH 2009–2013 affect and paralinguistics challenges. In section 2.6 we will teach you how to use the PortAudio interface components to set up a simple audio recorder and player as well as a full live feature extraction system. Section 2.7 will help you to get started with video feature extraction and synchronised audio-visual feature extraction. How you can plot the extracted features using the open-source tool gnuplot, is explained in section 2.8.

2.1 Obtaining and Installing openSMILE

Note for the impatient: If you have already downloaded openSMILE, and are an expert at compiling software on Windows and/or Linux, you may skip to section 2.2.1, which contains the quick-start compilation instructions.

The latest stable release of openSMILE can be found at <http://opensmile.audeering.com/>.

Major releases include binaries for Windows (32-bit) and Linux (64-bit), as well as Android ARM (Since 2.1, android-10). All releases contain the source code, which can be compiled on Linux, Mac OS, Windows, and for Android. This is the recommended way for Linux/Unix and Mac OS systems, and is mandatory if you require live audio recording/playback through portAudio. A release package contains the statically linked main executable `SMILEExtract` for Linux systems and a `SMILEExtract.Release.exe` and `openSmileLib.Release.dll` for Windows systems in the `bin/` folder, example configuration files in the `config/` folder, scripts for visualisation and other tasks such as model-building in the `scripts/` folder, and the source in the `src/` folder.

The binary releases are ready to use out-of-the-box. For Linux, a statically linked standalone binary is provided for 64-bit platforms. It is placed in the `bin` directory. Copy the executable

that matches your platform to a folder in your path (on Linux e.g. `/usr/local/bin`, on Windows e.g. `C:\Windows\`). Please be sure to also copy all DLL files to that path on Windows systems. Executables which are linked against PortAudio have a PA appended to their filenames (they are only provided for Windows). To test if your release works type

```
SMILEExtract -h
```

in the shell prompt on Unix systems or

```
SMILEExtract -h
```

in the Windows command-line prompt. If you see the usage information everything is working.

For Windows binaries with PortAudio support (PA suffix) and binaries without portaudio support are provided. A compiled portaudio DLL is also provided, which is linked against the Windows Media Extensions API. All these executables can be found in the `bin` subdirectory and your version of choice must be copied to `SMILEExtract.exe` in the top-level directory of the openSMILE distribution in order to be able to execute the example commands in this tutorial as they are printed (You must also copy the corresponding `.dll` to the top-level directory, however, without renaming it!).

Note for Linux: The Linux binaries contained in the releases are statically linked binaries, i.e. the shared API *libopensmile* is linked into the binary. The binaries only depend on `libc6` and `pthread`s. The downside of this method is that you cannot use binary plugins with these binaries! In order to use plugins, you must compile the source code to obtain a binary linked dynamically to *libopensmile* (see section 2.2.2). As no binary release with PortAudio support is provided for Linux, in order to use PortAudio audio recording/playback, you must compile from the source code (see sections 2.2.2 and 2.2.3).

Binaries for ARM Android platforms are provided for version 2.1 and above in the `bin/android-*/.libs` folder. You can use the `.libs` folder directly in Android eclipse projects and link against it. More documentation for Android building will follow.

No binaries are provided for openSMILE with openCV support. In order to use video features, you must compile from source on a machine with openCV installed. Compilation on both Windows and Linux is supported. See sections 2.2.4 and 2.2.7. If you have obtained a source only release, read the next section on how to compile and install it.

2.2 Compiling the openSMILE source code

The core of openSMILE compiles without any third-party dependencies, except for *pthread*s on Unix systems. The core version is a command-line feature extractor only. You can not do live audio recording/playback with this version. In order to compile with live audio support, you need the PortAudio¹ library. This will add support for audio recording and playback on Linux, Windows, and Mac OS. Please refer to section 2.2.3 for instructions on how to compile with PortAudio support on Linux and section 2.2.6 for Windows. For openCV support, please refer to sections 2.2.4 and 2.2.7. However, be sure to read the compilation instructions for the standalone version in sections 2.2.2 and 2.2.5 first, as the following sections assume you are familiar with the basics.

2.2.1 Build instructions for the impatient

This section provides quick start build instructions, for people who are familiar with building applications from source on Unix and Windows. If these instructions don't work for you, if you

¹Available at: <http://www.portaudio.com/>

get build errors, or you require more detailed information, then please refer to the following sections for more detailed instructions, especially for Unix build environments.

We will always distinguish between building with PortAudio support for live audio playback and recording, building with openCV support for video features, and building the standalone version without any third-party dependencies. Building without PortAudio and openCV is easier, as you will get a single statically-linked executable, which is sufficient for all off-line command-line feature extraction tasks.

Unix The very short version: Build scripts are provided. Use `sh buildStandalone.sh` or `sh buildWithPortAudio.sh` (both take the `-h` option to show a usage). The main binary is installed in `inst/bin/` and called `SMILEExtract`. Run `inst/bin/SMILEExtract -h` to see an online help. Libraries are installed in `inst/lib`. For openCV support, pass the path of your openCV installation to the `buildWithPortAudio.sh` script using the `-o` option. The little longer version of the short build instructions: unpack the openSMILE archive by typing:

```
tar -zxvf openSMILE-2.x.x.tar.gz
```

This creates a folder called `openSMILE-2.x.x`. Change to this directory by typing:

```
cd openSMILE-2.x.x
```

Then (assuming you have a running build system installed (autotools, libtool, make, gcc and g++ compiler, ...) and have a bash compatible shell) all you need to do is type:

```
bash buildStandalone.sh
```

or, if the above doesn't work:

```
sh buildStandalone.sh
```

This will configure, build, and install the openSMILE binary `SMILEExtract` to the `inst/bin` subdirectory. Add this directory to your path, or copy `inst/bin/SMILEExtract` to a directory in your search path. Optionally you can pass an install prefix path to the script as a parameter:

```
sh buildStandalone.sh -p /my/path/to/install/to
```

To compile openSMILE with PortAudio support, *if PortAudio is NOT installed on your system* type (optionally specifying an installation prefix for portaudio and openSMILE as first parameter):

```
sh buildWithPortAudio.sh [-p install-prefix-path]
```

A PortAudio snapshot is included in the `thirdparty` subdirectory. This will be unpacked, configured, and installed into the `thirdparty` directory, which is on the same level as the `opensmile` main directory. openSMILE will then be configured to use this installation. The built executable is called `SMILEExtract` and is found in the `inst/bin` sub-directory. Please note, that in this case, it is a wrapper script, which sets up the library path and calls the actual binary `SMILEExtract.bin`. Thus, you can also use this method if you have a locally installed (different) PortAudio. Just be sure to always run openSMILE through the wrapper script, and not run the binary directly (this will use the system wide, possibly incompatible, PortAudio library).

Please also note that the recommended way of building openSMILE with PortAudio is to use the `portaudio.tgz` shipped with the openSMILE release. Other versions of portaudio might

be incompatible with openSMILE and might not work correctly (sometimes the PortAudio development snapshots also contain bugs). This is also the only way An obsolete (and currently unsupported) way of compiling with the locally installed portaudio library was provided by `buildWithInstalledPortaudio.sh`. As this script is outdated, we recommend using `buildWithPortAudio.sh` and removing the linker and include paths which reference the custom portaudio build. The configure script will then automatically detect the system wide installation of portaudio.

Windows

Important note for building on Windows: Some versions of Visual Studio always select the ‘Debug’ configuration by default instead of the ‘Release’ configuration. However, you always want to build the ‘Release’ configuration, unless you are an openSMILE developer. Thus, you must always select the ‘Release’ configuration from the drop-down menu, before clicking on ‘Build Solution’ !!

With version 2.0 we have switched to providing Visual Studio 2010 build files instead of Visual Studio 2005. We do not provide support for any version of visual studio below 2010 anymore.

The very short version of the compile instructions: Open `ide/vs10/openSmile.sln`. Select the configuration you want to build. Release and Debug refer to the standalone versions. The output binary is placed in the `msvcbuild` folder in the top-level of the distribution. For building with PortAudio support a few more steps are necessary to patch the PortAudio build files. These steps are described in section 2.2.6.

The little longer version of the short build instructions:

Assuming that you have a correctly set up Visual Studio 2010 (or newer) environment, you can open the file `ide/vs10/openSmile.sln`, select the ‘Release’ configuration, and choose ‘Build solution’ to build the standalone version of openSMILE for Windows. This will create the command-line utility `SMILEExtract.exe` in the `msvcbuild` directory in the top-level directory of the package, which you can copy to your favourite path or call it directly. For building with PortAudio support a few more steps are necessary to patch the PortAudio build files. These steps are described in section 2.2.6.

2.2.2 Compiling on Linux/Mac

This section describes how to compile and install openSMILE on Unix-like systems step by step (in case the build scripts mentioned in the previous section don’t work for you). You need to have the following packages installed: `autotools` (i.e. `automake`, `autoconf`, `libtool`, and `m4`), `make`, GNU C and C++ compiler `gcc` and `g++`. You will also want to install `perl5` and `gnuplot` in order to run the scripts for visualisation. Please refer to your distribution’s documentation on how to install packages. You will also need root privileges to install new packages. We recommend that you use the latest Ubuntu or Debian Linux, where packages can easily be installed using the command `sudo apt-get install package-name`. *Note:* Please be aware that the following instructions assume that you are working with the `bash` shell. If you use a different shell, you might encounter some problems if your shell’s syntax differs from `bash`’s.

Start by unpacking the openSMILE package to a directory to which you have write access:

```
tar -zxvf openSMILE-2.x.x.tar.gz
```

Then change to the newly created directory:

```
cd openSMILE-2.x.x/
```


Important: The following is the manual build instructions, which were relevant for version 1.0, but have been replaced by the build scripts (`buildStandalone` and `buildWithPortAudio`) in version 2.0. Thus, you should prefer the scripts, or read the scripts and modify them, if you need to customize the build process. The following text is only included as a reference and for historical reasons. Refer to the quick build instruction in Section 2.2.1 for using the build scripts.

Next, run the following script *twice* (you may get errors the first time, this is ok):

```
bash autogen.sh
```

Important: You must run `autogen.sh` a second time in order to have all necessary files created! If you do not do so, running `make` after `configure` will fail because `Makefile.in` is not found. If you see warnings in the `autogen.sh` output you can probably ignore them, if you get errors try to run `autogen.sh` a third time.

Note: if you cannot run `./autogen.sh` then run it either as `sh autogen.sh` or change the executable permission using the command `chmod +x autogen.sh`. If you get errors when running this script the second time, your version of autotools might be outdated. Please check that you have at least automake 1.10 and autoconf 2.61 installed (type `autoconf --version` and `automake --version` to obtain the version number).

Now configure openSMILE with

```
./configure
```

to have it installed in the default location `/usr` or `/usr/local` (depends on your system), or use the `--prefix` option to specify the installation directory (**important:** you need to use this, if you *don't have root privileges* on your machine):

```
./configure --prefix=/directory/prefix/to/install/to
```

Please make sure you have full write access to the directory you specify, otherwise the `make install` command will fail.

On modern CPUs you can create an optimised executable for your CPU by using the following compiler flags: `-O2 -mfpmath=sse -march=native`. You can pass those flags directly to `configure` (you may or may not combine this with the `--prefix` option):

```
./configure CXXFLAGS=''-O2 -mfpmath=sse -march=native'' CFLAGS=''-O2
-mfpmath=sse -march=native''
```

Please note that this option is not supported by all compilers.

The default setup will create a `SMILEExtract` binary and a `libopensmile.so` shared library. This is usually what you want, especially if you want to use plugins. However, in some cases a portable binary, without library dependencies may be preferred. To create such a statically linked binary pass the following option to the `configure` script:

```
./configure --enable-static --enable-shared=no
```

Warning: openSMILE plugins will not work with statically linked binaries.

After you have successfully configured openSMILE (i.e. if there were not error messages during configuring - warning messages are fine), you are now ready to compile openSMILE with this command:

```
make -j4 ; make
```

Note: `make -j4` runs 4 compile processes in parallel. This speeds up the compilation process on most machines (also single core). However, running only `make -j4` will result in an error, because `libopensmile` has not been built when `SMILEExtract` is built. Thus, you need to run a single `make` again. This should finish without error. If you have trouble with the `-j4` option, simply use `make` without options.

You are now ready to install openSMILE by typing:

```
make install
```

You have to have root privileges to install openSMILE in a standard location (i.e. if you have *not* specified an alternate path to `configure`). It is also possible to run openSMILE without installation directly from the top-level directory of the openSMILE distribution (this should be your current directory at the moment, if you have followed the above steps carefully). In this case you have to prefix the executable with `./` i.e. you have to run `./SMILEExtract` instead of `SMILEExtract`.

Please note that `make install` currently only installs the openSMILE feature extractor binary `SMILEExtract` and the feature extractor's library `libopensmile.so`. Configuration files still remain in the build directory. Therefore, the examples in the following sections will assume that all commands are entered in the top-level directory of the openSMILE distribution.

For splitting openSMILE into an executable and a dynamic library there have been primarily two reasons:

Reusability of source-code and binaries. The openSMILE library contains the API components with all the base classes and the standard set of components distributed with openSMILE. Custom components, or project specific implementations can be linked directly into the `SMILEExtract` executable. Thus, the library can be compiled without any additional third-party dependencies and can be maintained and distributed independently, while other projects using openSMILE can create a GUI frontend, for example, which depends on various GUI libraries, or add components which interface with a middleware, as in the SEMAINE project².

Support for linking binary plugins at run-time. Since binary plugins depend on the openSMILE API and various base classes, instances of these base classes may be present only once in the process memory during run-time. This can only be achieved by off-loading these classes to a separate library.

Note: If you have installed openSMILE to a non-default path, you must set your library path to include the newly installed `libopensmile` before running the `SMILEExtract` binary (replace `/directory/prefix/to/install/to` by the path you have passed to the `--prefix` option of the `configure` script):

```
export LD_LIBRARY_PATH=/directory/prefix/to/install/to/lib
```

You will also need to add the path to the binary to your current `PATH` variable:

```
export PATH="$PATH:/directory/prefix/to/install/to/lib"
```

Attention: You need to do this every time you reboot, log-on or start a new shell. To avoid this check your distribution's documentation on how to add environment variables to your

²See: <http://www.semaine-project.eu/>

shell's configuration files. For the bash shell usually a file called `.profile` or `.bashrc` exists in your home directory to which you can add the two export commands listed above. You can also have a look at the script `buildWithPortAudio.sh`, which creates a wrapper shell script for `SMILEExtract`.

2.2.3 Compiling on Linux/Mac with PortAudio

Important: The following is the manual build instructions, which were relevant for version 1.0, but have been replaced by the build scripts (`buildStandalone` and `buildWithPortAudio`) in version 2.0. Thus, you should prefer the scripts, or read the scripts and modify them, if you need to customize the build process. The following text is only included as a reference and for historical reasons. Refer to the quick build instruction in Section 2.2.1 for using the build scripts.

To compile openSMILE with PortAudio support, the easiest way is to install the latest version of PortAudio via your distribution's package manager (be sure to install a development package, which includes development header files). You can then run the same steps as in section 2.2.2, the configure script should automatically detect your installation of PortAudio.

If you cannot install packages on your system or do not have access to a PortAudio package, or the portaudio version installed on your system does not work with openSMILE, unpack the file `thirdparty/portaudio.tgz` in the `thirdparty` directory (`thirdparty/portaudio`). Then read the PortAudio compilation instructions and compile and install PortAudio according to these instructions. You can then continue with the steps listed in section 2.2.2. If you have installed PortAudio to a non-standard location (by passing the `--prefix` option to PortAudio's `configure`), you have to pass the path to your PortAudio installation to openSMILE's configure script:

```
./configure --with-portaudio=/path/to/your/portaudio
```

After successfully configuring with PortAudio support, type `make -j4; make; make install`, as described in the previous section.

2.2.4 Compiling on Linux with openCV and portaudio support.

This section briefly describes how to install OpenCV and compile openSMILE with openCV-based video feature support.

Installing OpenCV

You need OpenCV version 2.2 or higher as prerequisite (besides openSMILE version 2.0 or above). If you are using Ubuntu 12.04 or higher, you are lucky since Ubuntu provides OpenCV 2.2 or higher through the standard repositories. To install, just execute

```
sudo apt-get install libopencv*
```

in a shell. This will also take care of the dependencies. The installation path of OpenCV (`PATH_TO_OPENCV`) in this case will be `/usr/`.

If you are however using a different distribution or any older Ubuntu version, you might have to compile OpenCV yourself. Detailed instructions on this topic can be found here:

<http://opencv.willowgarage.com/wiki/InstallGuide>

Don't forget to execute `sudo make install` at the end of the installation to install OpenCV to the predefined path. You will need this path (`PATH_TO_OPENCV`) later in the build process. If you did not specify an alternate installation path, it will most likely be `/usr/local/`. After the installation you might need to update your library paths in `/etc/ld.so.conf` and add the line `/usr/local/lib`, if it is not already there.

Compiling openSMILE on Linux with openCV video support

After you have successfully installed OpenCV, openSMILE can be compiled with support for video input through OpenCV. You will use the standard openSMILE unix build scripts `build-Standalone.sh` and `buildWithPortaudio.sh` for this purpose, depending on whether you want to build the standalone version with openCV support or if you also need PortAudio support. Please build your version without openCV first, as described in section 2.2.1. If this succeeds, you can re-run the build script and append the `-o` option to specify the path to your openCV installation.

After the build process is complete, you can check with `./SMILEExtract -L`, whether `cOpenCVSource` appears in the component list. In case it does not appear, try to rebuild from one more time by running the build script with the openCV option, before asking for help for sending a bug report to the authors.

If you get an error message that some of the `libopencv*.so` libraries are not found when you run `SMILEExtract`, type this command in the shell before you run `SMILEExtract`:

```
export LD_LIBRARY_PATH="/usr/local/lib"
```

2.2.5 Compiling on Windows

For compiling openSMILE on Microsoft Windows (Vista, and Windows 7 are supported) there are two ways:

- Using Mingw32 and MSYS or Cygwin
- Using Visual Studio 2010 or above

The preferred way (and the only officially supported way) is to compile with Visual Studio 2010. If you want to use Mingw32, please refer to <http://www.mingw.org/wiki/msys> for how to correctly set up your Mingw32 and MSYS system with all necessary development tools (autoconf, automake, libtool, and m4 as included in the MSYS DTK). You should then be able to loosely follow the Unix installation instructions in sections 2.2.2 and 2.2.3.

To compile with Microsoft Visual Studio a single solution file is provided in `ide/vs10/openSmile.sln`. You can select several configurations from this solution which represent the various combinations of the standalone version (simply Release and Debug configurations) and support of OpenCV and PortAudio (named accordingly).

Due to some issues with Visual Studio not correctly recognizing the project build order in some configurations, the build process might fail if you just select the option "Build solution!". To solve this issue, you have to build the projects manually. First build `openSmileLib*`, then `openSmileLib`, then `SMILEExtract`, or try to build the entire solution multiple times until the number of error messages has converged (build failed, try building projects manually) or has reached zero (build succeeded!).

After successfully building the solution you should have an `openSmileLib*.dll` and a `SMILEExtract*.exe` in the `msvcbuild` directory in the top-level directory of unzipped source package (NOT in the `ide/vs10/Release*` or `Debug*` folder!). The `*` refers to a prefix that is appended depending on the configuration (PortAudio, OpenCV). After building you can now copy the openSmile dll (also the portaudio dll) and the executable to a directory in your path, e.g. `C:\Windows\system32`, or put everything in a single arbitrary directory and run the executable from this directory.

2.2.6 Compiling on Windows with PortAudio

A PortAudio snapshot known to work with openSMILE is provided in the `thirdparty` subdirectory. Alternatively you can download the latest PortAudio SVN snapshot from <http://www.portaudio.com/>. It is a good idea (however not actually necessary) to read the PortAudio compilation instructions for windows before compiling openSMILE .

Now, unpack the Windows PortAudio source tree to the `thirdparty` subdirectory of the openSMILE distribution (top-level in unpacked zip file), which should create a directory called `portaudio` there. If you don't unpack PortAudio to this location, then you need to modify the Visual Studio project files mentioned in the next paragraph and adjust the Include and Linker paths for PortAudio. By default PortAudio will be built supporting all possible media APIs on a Windows system. However, in most cases only the default Windows Media Extensions (WME) are available and absolutely sufficient. Thus, we provide modified build files for PortAudio in the directory `ide/vs10`. To use them (after unpacking PortAudio to the `thirdparty/portaudio` subdirectory), copy the following files from `ide/vs10` to `thirdparty/portaudio/build/msvc`:

`portaudio.vcxproj`, and `portaudio.def`.

The modified build files basically disable the DirectX, ASIO, and wasapi APIs. They add `PA_NO_DS` and `PA_NO_ASIO` to the preprocessor defines (C/C++ settings tab, preprocessor) and disable all the `.cpp` files in the related hostapi project folders. Moreover, the output path is adjusted to the `msvcbuild` directory in the top-level directory and the filename of the output dll is set to `portaudio_x86.dll`.

Now, to compile openSMILE with PortAudio support, select the `ReleasePortAudio` configuration from the solution `ide/vs10/openSmile.sln` and build it. Due to some issues with Visual Studio not correctly recognizing the project build order in some configurations, the build process might fail if you just select the option "Build solution!". To solve this issue, you have to build the projects manually. First build `portaudio`, then `openSmileLib*`, then `openSmileLib`, then `SMILEExtract`, or try to build the entire solution multiple times until the number of error messages has converged (build failed, try building projects manually) or has reached zero (build succeeded!).

Please note: the PortAudio versions of the openSMILE Visual Studio projects assume that the dll is called `portaudio_x86.dll` and the import library `portaudio_x86.lib` and both are found in the `msvcbuild` directory in the top-level. This name, however, might be different, depending on your architecture. Thus, you should check this and change the name of the import library in the Linker advanced settings tab.

2.2.7 Compiling on Windows with openCV support.

You first need to download a Windows binary release of openCV version 2.4.5 from <http://opencv.org/downloads.html>

If you have another version (≥2.2), you will need to modify the names of the `.lib` files in the solution configuration manually (in `opencv.props` in `ide/vs10`). The following instructions assume you have version 2.4.5.

Unpack the openCV distribution to a directory you choose by running the self-extracting `.exe`. From that directory copy `build/<yourplatform>/vc10/lib` and `build/include/` to `thirdparty/compiled/include` and `thirdparty/compiled/lib`. Make sure to replace `<yourplatform>` by the correct platform (x64 or x86) to match the platform you are building openSMILE for (drop down menu in Visual Studio next to the solution configuration). Please note that currently in the release candidates the build only works for the Win32 (x86) platform. The proper

configuration for the x64 platform will follow shortly. If you need to have an x64 version, you will have to copy the configuration from the Win32 platform.

Then build the `ReleaseOpenCV` configuration. If you also want PortAudio support, pick the according configuration (`ReleasePortaudioOpenCV`) and additionally follow the instructions mentioned in section 2.2.6. Due to some issues with Visual Studio not correctly recognizing the project build order in some configurations, the build process might fail if you just select the option "Build solution!". To solve this issue, you have to build the projects manually. First build portaudio (if building with portaudio support), then `openSmileLib*`, then `openSmileLib`, then `SMILEExtract`, or try to build the entire solution multiple times until the number of error messages has converged (build failed, try building projects manually) or has reached zero (build succeeded!).

The built executable will be located in `msvcbuild/`. You will find `openSmileLib_<configuration>.dll` and `SMILEExtract_<configuration>.exe` there, which you will both need. The .dll should be in the same path as the .exe to run, or copied to a system path such as `Windows/system32`. You will also need to copy the `opencv` dlls to a system path or to the same path that contains the `SMILEExtract` binary. The `opencv` executables are contained in the unpacked `openCV` distribution from which you copied the lib files. Look in the folder `bin/` which is on the same level as the `lib/` folder.

2.3 Extracting your first features

Now, that you have either successfully downloaded and installed the binary version of `openSMILE` or have compiled the source code yourself, you are ready to test the program and extract your first features. To check if you can run `SMILEExtract`, type:

```
SMILEExtract -h
```

If you see the usage information and version number of `openSMILE`, then everything is set up correctly. You will see some lines starting with (MSG) at the end of the output, which you can safely ignore. To check if your `SMILEExtract` binary supports live audio recording and playback, type:

```
SMILEExtract -H cPortaudio
```

If you see various configuration option of the `cPortaudio` components, then your binary supports live audio I/O. If you see only three lines with messages, then you do not have live audio support. To check if your `SMILEExtract` binary supports video features via `OpenCV`, type:

```
SMILEExtract -H cOpenCV
```

If you see various configuration option of the `cPortaudio` components, then your binary supports live audio I/O.

Please note: You may have to prefix a `“./”` on Unix like systems, if `SMILEExtract` is not in your path but in the current directory instead.

Now we will start using `SMILEExtract` to extract very simple audio features from a wave file. You can use your own wave files if you like, or use the files provided in the `wav-samples` directory.

For a quick start, we will use an example configuration file provided with the `openSMILE` distribution. Type the following command in the top-level directory of the `openSMILE` package (if you start `openSMILE` in a different directory you must adjust the paths to the config file and the wave file):

```
SMILExtract -C config/demo/demo1\_energy.conf -I wav\_samples/speech01.
wav -O speech01.energy.csv
```

If you get only (MSG) and (WARN) type messages, and you see **Processing finished!** in the last output line, then openSMILE ran successfully. If something fails, you will get an (ERROR) message.

Note for windows users: Due to faulty exception handling, if an exception indicating an error is thrown in the DLL and caught in the main executable, Windows will display a program crash dialogue. In most cases openSMILE will have displayed the error message beforehand, so can just close the dialogue. In some cases however, Windows kills the program before it can display the error message. If this is the case, please use Linux, or contact the authors and provide some details on your problem.

Now, if openSMILE ran successfully, open the file **speech01.energy.csv** in a text editor to see the result. You can also plot the result graphically using gnuplot. This is discussed in section 2.8.

Next, we will generate the configuration file from the above simple example ourselves, to learn how openSMILE configuration files are written. openSMILE can generate configuration file templates for simple scenarios. We will use this function to generate our first configuration file, which will be capable of reading a wave file, compute frame energy, and saving the output to a CSV file. First, create a directory **myconfig** which will hold your configuration files. Now type the following (without newlines) to generate the first configuration file:

```
SMILExtract -cfgFileTemplate -configDflt cWaveSource,cFramer,cEnergy,
cCsvSink -l 1 2> myconfig/demo1.conf
```

The **-cfgFileTemplate** option instructs openSMILE to generate a configuration file template, while the **-configDflt** option is used to specify a comma separated list of components which shall be part of the generated configuration. The **-l 1** option sets the log-level to one, to suppress any messages, which should not be in the configuration file (you will still get ERROR messages on log-level one, e.g. messages informing you that components you have specified do not exist, etc.). The template text is printed to standard error, thus we use **2>** to dump it to the file **myconfig/demo1.conf**. If you want to add comments describing the individual option lines in the generated configuration file, add the option **-cfgFileDescriptions** to the above command-line.

The newly generated file consists of two logical parts. The first part looks like this (please note, that comments in the examples are started by **;** or **//** and may only start at the beginning of a line):

```

;= component manager configuration (= list of enabled components!) =

[componentInstances:cComponentManager]
// this line configures the default data memory:
instance[dataMemory].type = cDataMemory
instance[waveSource].type = cWaveSource
instance[framer].type = cFramer
instance[energy].type = cEnergy
instance[csvSink].type = cCsvSink
// Here you can control the amount of detail displayed for the
// data memory level configuration. 0 is no information at all,
// 5 is maximum detail.
printLevelStats = 1
// You can set the number of parallel threads (experimental):
nThreads = 1
```

It contains the configuration of the component manager, which determines what components are instantiated when you call SMILEExtract. There always has to be one `cDataMemory` component, followed by other components. The name given in `[]` specifies the name of the component instance, which must be unique within one configuration.

The next part contains the component configuration sections, where each begins with a section header:

```
[ waveSource : cWaveSource ]
...
[ framer : cFramer ]
...
[ energy : cEnergy ]
...
[ csvSink : cCsvSink ]
...
```

The section header follows this format: `[instanceName:componentType]`. The template component configuration sections are generated with all available values set to their default values. This functionality currently is still experimental, because some values might override other values, or have a different meaning if explicitly specified. Thus, you should carefully check all the available options, and list only those in the configuration file which you require. Even if in some cases you might use the default values (such as the number of spectral bands, etc.) it is considered good practice to include these in the configuration file. This will ensure compatibility with future versions, in case the defaults - for whatever reason - might change. Moreover, it will increase the readability of your configuration files because all parameters can be viewed in one place without looking up the defaults in this manual.

Next, you have to configure the component connections. This can be done by assigning so called data memory “levels” to the `dataReader` and `dataWriter` components which are always contained in each source, sink, or processing component by modifying the `reader.dmLevel` and `writer.dmLevel` lines. You can choose arbitrary names for the writer levels here, since the `dataWriters` register and create the level you specify as `writer.dmLevel` in the data memory. You then connect the components by assigning the desired read level to `reader.dmLevel`. Thereby the following rules apply: for one level only **one** writer may exist, i.e. only one component can write to a level; however, there is no limit to the number of components that read from a level, and one component can read from more than one level if you specify multiple level names separated by a `;`, such as `reader.dmLevel = energy;loudness` to read data from the levels `energy` and `loudness`. Data is thereby concatenated column wise.

For our example configuration we want the `cFramer` component to read from the input PCM stream, which is provided by the `cWaveSource` component, create frames of 25 ms length every 10 ms and write these frames to a new level we call “energy”), thus we change the following lines:

```
[ waveSource : cWaveSource ]
writer.dmLevel = <<XXXX>>
```

to

```
[ waveSource : cWaveSource ]
writer.dmLevel = wave
```

and the `framer` section

```
[ framer : cFramer ]
reader.dmLevel = <<XXXX>>
writer.dmLevel = <<XXXX>>
```


...

to (note, that we removed a few superfluous `frameSize*` options and changed `frameStep` to 0.010):

```
[ framer : cFramer ]
reader.dmLevel = wave
writer.dmLevel = waveframes
copyInputName = 1
frameMode = fixed
frameSize = 0.025000
frameStep = 0.010000
frameCenterSpecial = left
noPostEOIprocessing = 1
```

Next, the `cEnergy` component shall read the audio frames and compute the signal log energy, and the `cCsvSink` shall write them to a CSV format file. Thus, we change the corresponding lines to:

```
[ energy : cEnergy ]
reader.dmLevel = waveframes
writer.dmLevel = energy
...
rms = 0
log = 1
...
[ csvSink : cCsvSink ]
reader.dmLevel = energy
filename = myenergy.csv
...
```

We are now ready to run `SMILEExtract` with our own configuration file:

```
SMILEExtract -C myconfig/demo1.conf
```

This will open the file “input.wav” in the current directory (be sure to copy a suitable wave file and rename it to “input.wav”), do the feature extraction, and save the result to “myenergy.csv”. The result should be the same as with the example configuration file.

If you want to be able to pass the input file name and the output file name on the `SMILEExtract` command-line, you have to add a command to the configuration file to define a custom command-line option. To do this, change the filename lines of the wave source and the csv sink to:

```
[ waveSource : cWaveSource ]
...
filename = \cm[inputfile(I):file name of the input wave file]
...
[ csvSink : cCsvSink ]
...
filename = \cm[outputfile(O):file name of the output CSV file]
...
```

You can now run:

```
SMILEExtract -C myconfig/demo1.conf -I wav\_samples/speech01.wav -O
speech01.energy.csv
```

This concludes the introductory section. We hope that you now understand the basics of how to use and configure `openSMILE`, and are ready to take a look at the more complex examples, which are explained in section 2.5. To explore the full potential of `openSMILE` configuration

files, please read section 3.2, which provides description of the format, and section 3.3, which describes the function and configuration options of all components in detail. If you are interested what is going on inside openSMILE, which components exist besides those which are instantiable and connectable via the configuration files, and to learn more about the terminology used, then you should read section 2.4 which describes the program architecture in detail.

2.4 What is going on inside of openSMILE

The SMILExtract binary is the main application which can run all configuration files. If you take a look at the source code of it (which is found in `SMILExtract.cpp`), you will see that it is fairly short. It uses the classes from the openSMILE API to create the components and run the configurations. These API functions can be used in custom applications, such as GUI front-ends etc. Therefore, they will be described in more detail in the developer's documentation in section 4. However, to obtain a general understanding what components make openSMILE run, how they interact, and in what phases the program execution is split, a brief overview is given in this section.

openSMILE's application flow can be split into three general phases:

Pre-config phase Command-line options are read and the configuration file is parsed. Also, usage information is displayed, if requested, and a list of built-in components is generated.

Configuration phase The component manager is created and instantiates all components listed in its `instances` configuration array. The configuration process is then split into 3 phases, where components first register with the component manager and the data memory, then perform the main configuration steps such as opening of input/output files, allocation of memory, etc., and finally finalise their configuration (e.g. set the names and dimensions of their output fields, etc.). Each of the 3 phases is passed through several times, since some components may depend on other components having finished their configuration (e.g. components that read the output from another component and need to know the dimensionality of the output and the names of the fields in the output). Errors, due to mis-configurations, bogus input values, or inaccessible files, are likely to happen during this phase.

Execution phase When all components have been initialised successfully, the component manager starts the main execution loop (also referred to as tick-loop). Every component has a `tick()` method, which implements the main incremental processing functionality and reports on the status of the processing via its return value.

In one iteration of the execution loop, the component manager calls all `tick()` functions in series (*Note:* the behaviour is different, when components are run in multiple threads). The loop is continued as long as at least one component's `tick()` method returns a non-zero value (which indicates that data was processed by this component).

If all components indicate that they did not process data, it can be safely assumed that no more data will arrive and the end of the input has been reached (this may be slightly different for on-line settings, however, it is up to the source components to return a positive return value or pause the execution loop, while they are waiting for data).

When the end of the input is reached, the component manager signals the end-of-input condition to the components by running one final iteration of the execution loop. After that the execution loop will be ran a new, until all components report a failure status. This second phase is referred to end-of-input processing. It is mainly used for off-line processing,

e.g. to compute features from the last (but incomplete) frames, to mean normalise a complete sequence, or to compute functionals from a complete sequence.

openSMILE contains three classes which cannot be instantiated from the configuration files. These are the commandline parser (`cCommandlineParser`), the configuration manager (`cConfigManager`), and the component manager (`cComponentManager`). We will now briefly describe the role of each of these in a short paragraph. The order of the paragraph corresponds to the order the classes are created during execution of the `SMILEExtract` program.

The commandline parser This class parses the command-line and provides options in an easily accessible format to the calling application. Simple command-line syntax checks are also performed. After the configuration manager has been initialised and the configuration has been parsed, the command-line is parsed a second time, to also get the user-defined command-line options set in the current configuration file.

The configuration manager The configuration manager loads the configuration file, which was specified on the `SMILEExtract` command-line. Thereby, configuration sections are split and then parsed individually. The configuration sections are stored in an abstract representation as `ConfigInstance` classes (the structure of these classes is described by a `ConfigType` class). Thus, it is easy to add additional parsers for formats other than the currently implemented ini-style format.

The component manager The component manager is responsible of instantiating, configuring, and executing the components. The details have already been described in the above section on openSMILE's application flow. Moreover, the component manager is responsible of enumerating and registering components in plugins. Therefore, a directory called `plugins` is scanned for binary plugins. The plugins found are registered, and become useable exactly in the same way as built-in components. A single plugin binary thereby can contain multiple openSMILE components.

The components instantiated by the component manager are all descendants of the `cSmileComponent` class. They have two basic means of standardised communication: a) directly and asynchronously, via smile messages, and b) indirectly and synchronously via the data memory.

Method a) is used to send out-of-line data, such as events and configuration changes directly from one smile component to another. Classifier components, for example, send a 'classification-Result' message, which can be caught by other components (esp. custom plug-ins), to change their behaviour or send the message to external sources.

Method b) is the standard method for handling of data in openSMILE. The basic principle is that of a data source producing a frame of data and writing it to the data memory. A data processor reads this frame, applies some fancy algorithm to it, and writes a modified output frame back to a different location in the data memory. This step can be repeated for multiple data processors. Finally, a data sink reads the frame and passes it to an external source or interprets (classifies) it in some way. The advantage of passing data indirectly is that multiple components can read the same data, and data from past frames can be stored efficiently in a central location for later use.

2.4.1 Incremental processing

The data-flow in openSMILE is handled by the `cDataMemory` component. This component manages multiple data memory 'levels' internally. These levels are independent data storage

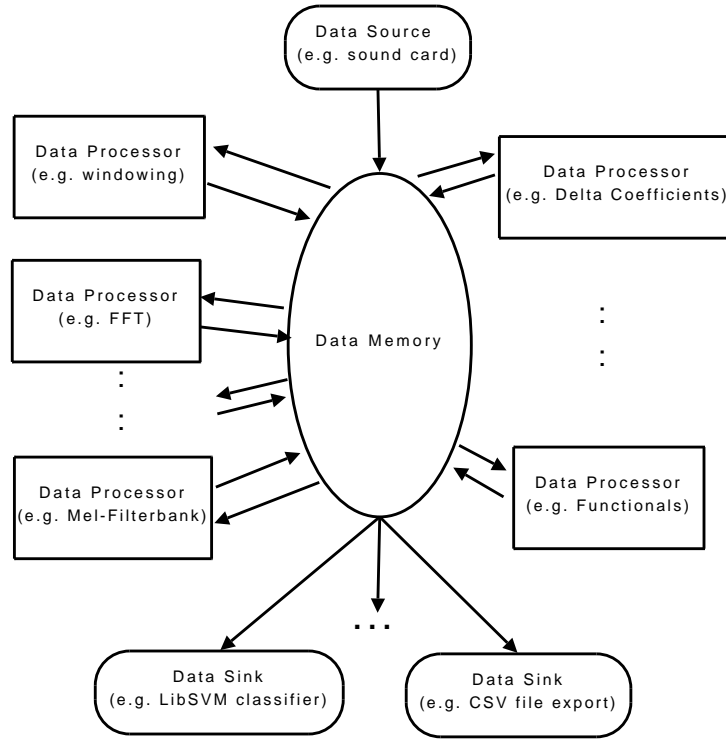


Figure 2.1: Overview on openSMILE's component types and openSMILE's basic architecture.

locations, which can be written to by exactly one component and read by an arbitrary number of components. From the outside (the component side) the levels appear to be a $N \times \infty$ matrix, with N rows, whereby N is the frame size. Components can read / write frames (=columns) at / to any location in this virtual matrix. If this matrix is internally represented by a ring-buffer, a write operation only succeeds if there are empty frames in the buffer (frames that have not been written to, or frames that have been read by all components reading from the level), and a read operation only succeeds if the referred frame index lies no more than the ring buffer size in the past. The matrices can also be internally represented by a non-ring buffer of fixed size ($nT=size$, $growDyn=0$, $isRb=0$), or variable size ($nT=initial\ size$, $growDyn=1$, $isRb=0$). In the case of the variable size a write will always succeed, except when there is no memory left; for a fixed frame size a write will succeed until the buffer is full, after that the write will always fail. For fixed buffers, reads from 0 to the current write position will succeed.

Figure 2.1 shows the overall data-flow architecture of openSMILE, where the data memory is the central link between all `dataSource`, `dataProcessor`, and `dataSink` components.

The ring-buffer based incremental processing is illustrated in figure 2.2. Three levels are present in this setup: wave, frames, and pitch. A `cWaveSource` component writes samples to the 'wave' level. The write positions in the levels are indicated by a red arrow. A `cFramer` produces frames of size 3 from the wave samples (non-overlapping), and writes these frames to the 'frames' level. A `cPitch` (a component with this name does not exist, it has been chosen here only for illustration purposes) component extracts pitch features from the frames and writes them to the 'pitch' level. In figure 2.2 (right) the buffers have been filled, and the write pointers have been warped. Data that lies more than 'buffersize' frames in the past has been overwritten.

Figure 2.3 shows the incremental processing of higher order features. Functionals (max and min) over two frames (overlapping) of the pitch features are extracted and saved to the level 'func'.

The size of the buffers must be set correctly to ensure smooth processing for all block sizes. A

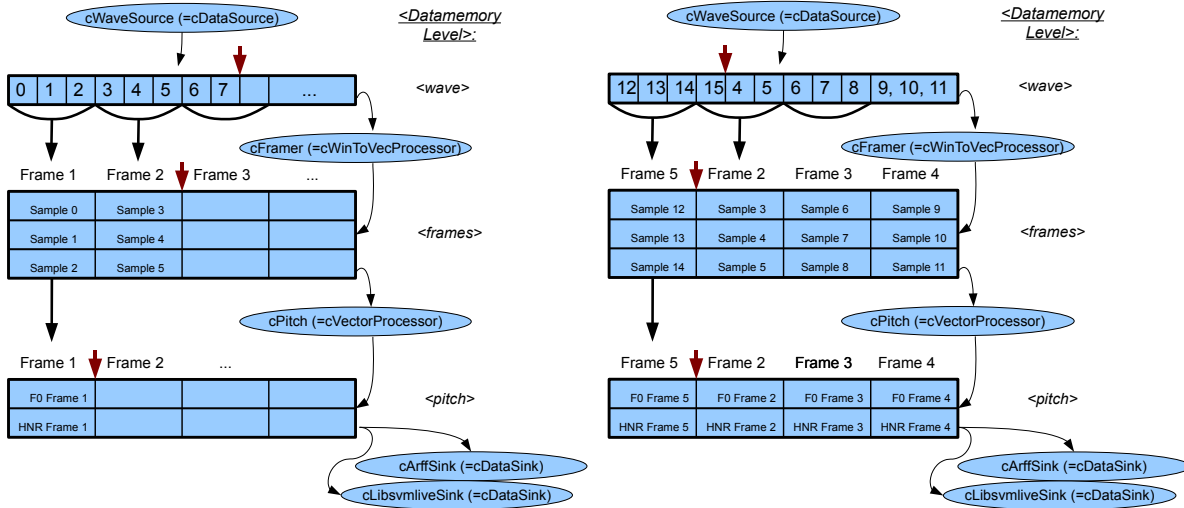


Figure 2.2: Incremental processing with ring-buffers. Partially filled buffers (left) and filled buffers with warped read/write pointers (right).

‘blocksize’ thereby is the size of the block a reader or writer reads/writes from/to the dataMemory at once. In the above example the read blocksize of the functionals component would be 2 because it reads two pitch frames at once. The input level buffer of ‘pitch’ must be at least 2 frames long, otherwise the functionals component will never be able to read a complete window from this level.

openSMILE handles automatic adjustment of the buffersizes. Therefore, readers and writers must register with the data memory during the configuration phase and publish their read and write blocksizes. The minimal buffersize is computed based on these values. If the buffersize of a level is set smaller than the minimal size, the size will be increased to the minimum possible size. If the specified size (via configuration options) is larger than the minimal size, the larger size will be used. *Note:* this automatic buffersize setting only applies to ring-buffers. If you use non-ring buffers, or if you want to process the full input (e.g. for functionals of the complete input, or mean normalisation) it is always recommended to configure a dynamically growing non-ring buffer level (see the `cDataWriter` configuration for details, section 3.3.2).

2.4.2 Smile messages

This section has yet to be written. In the meantime, please refer to the file `doc/developer/messages.txt` for a minimal documentation of currently available smile messages. See also the `smileComponent.hpp` source file, which contains the structural definitions of smile messages.

2.4.3 openSMILE terminology

In the context of the openSMILE data memory various terms are used which require clarification and a precise definition, such as ‘field’, ‘element’, ‘frame’, and ‘window’.

You have learnt about the internal structure of the dataMemory in section 2.4.1. Thereby a level in the data memory represents a unit which contains numeric data, frame meta data, and temporal meta data. Temporal meta data is present on the one hand for each frame, thereby describing frame timestamps and custom per frame meta information, and on the other hand globally, describing the global frame period and timing mode of the level.

If we view the numeric contents of the data memory level as a 2D `<nFields x nTimestamps>`

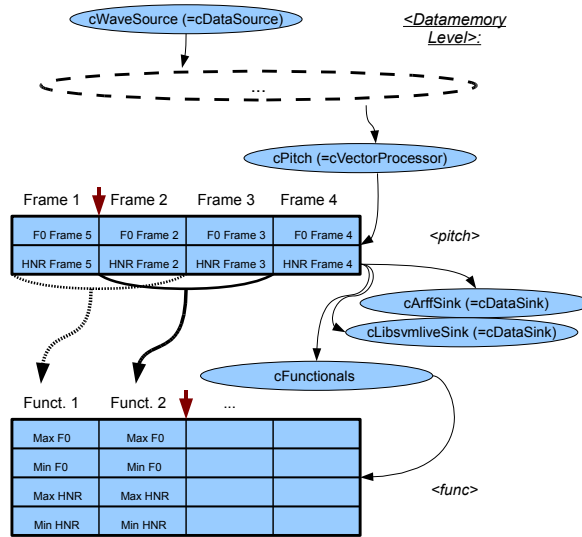


Figure 2.3: Incremental computation of high-level features such as statistical functionals.

matrix, ‘frames’ correspond to the columns of this matrix, and ‘windows’ or ‘contours’ correspond the rows of this matrix. The frames are also referred to as (column-)‘vectors’ in some places. (*Note:* when exporting data to files, the data – viewed as matrix – is transposed, i.e. for text-based files (CSV, ARFF), the rows of the file correspond to the frames.) The term ‘elements’ – as used in openSMILE – refers to the actual elements of the frames/vectors. The term ‘field’ refers to a group of elements that belongs together logically and where all elements have the same name. This principle shall be illustrated by an example: A feature frame containing the features ‘energy’, ‘F0’, and MFCC 1-6, will have $1 + 1 + 6 = 8$ elements, but only 3 fields: the field ‘energy’ with a single element, the field ‘F0’ with a single element, and the (array-) field ‘MFCC’ with 6 elements (called ‘MFCC[0]’ – ‘MFCC[1]’).

2.5 Default feature sets

For common tasks from the Music Information Retrieval and Speech Processing fields we provide some example configuration files in the `config/` directory for the following frequently used feature sets. These also contain the baseline acoustic feature sets of the 2009–2013 INTERSPEECH challenges on affect and paralinguistics:

- Chroma features for key and chord recognition
- MFCC for speech recognition
- PLP for speech recognition
- Prosody (Pitch and loudness)
- The INTERSPEECH 2009 Emotion Challenge feature set
- The INTERSPEECH 2010 Paralinguistic Challenge feature set
- The INTERSPEECH 2011 Speaker State Challenge feature set
- The INTERSPEECH 2012 Speaker Trait Challenge feature set
- The INTERSPEECH 2013 ComParE feature set

- The MediaEval 2012 TUM feature set for violent scenes detection.
- Three reference sets of features for emotion recognition
- Audio-visual features based on INTERSPEECH 2010 audio features.

These configuration files can be used as they are, or as a basis for your own feature files.

Note: If you publish results with features extracted by openSMILE, we would appreciate it if you share your configuration files with the research community, by uploading them to your personal web-pages and providing the URL in the paper, for example.

2.5.1 Chroma features

The configuration file `config/chroma_fft.conf` computes musical Chroma features (for 12 semi-tones) from a short-time FFT spectrogram (window-size 50 ms, rate 10 ms, Gauss-window). The spectrogram is scaled to a semi-tone frequency axis scaling using triangular filters. To use this configuration, type:

```
SMILExtract -C config/chroma\_fft.conf -I input.wav -O chroma.csv
```

The resulting CSV file contains the Chroma features as ascii float values separated by ‘;’, one frame per line. This configuration uses the ‘cTonespec’ (section 3.3.6) component to compute the semitone spectrum. We also provide a configuration using the experimental ‘cTonefilt’ (section 3.3.6) as a replacement for ‘cTonespec’ in the file `config/chroma_filt.conf`.

We also provide an example configuration for computing a single vector which contains the mean value of the Chroma features computed over the complete input sequence. Such a vector can be used for recognising the musical key of a song. The configuration is provided in `config/chroma_fft.sum.conf`. It uses the ‘cFunctionals’ component (section 3.3.7) to compute the mean values of the Chroma contours. Use it with the following command-line:

```
SMILExtract -C config/chroma\_fft.sum.conf -I input.wav -O chroma.csv
```

`chroma.csv` will contain a single line with 12 values separated by ‘;’, representing the mean Chroma values.

2.5.2 MFCC features

For extracting MFCC features (HTK compatible) the following four files are provided (they are named after the corresponding HTK parameter kinds they represent):

MFCC12_0_D_A.conf This configuration extracts Mel-frequency Cepstral Coefficients from 25 ms audio frames (sampled at a rate of 10 ms) (Hamming window). It computes 13 MFCC (0-12) from 26 Mel-frequency bands, and applies a cepstral liftering filter with a weight parameter of 22. 13 delta and 13 acceleration coefficients are appended to the MFCC.

MFCC12_E_D_A.conf This configuration is the same as `MFCC12_0_D_A.conf`, except that the log-energy is appended to the MFCC 1-12 instead of the 0-th MFCC.

MFCC12_0_D_A_Z.conf This configuration is the same as `MFCC12_0_D_A.conf`, except that the features are mean normalised with respect to the full input sequence (usually a turn or sub-turn segment).

MFCC12_E_D_A_Z.conf This configuration is the same as MFCC12_E_D_A.conf, except that the features are mean normalised with respect to the full input sequence (usually a turn or sub-turn segment).

The frame size is set to 25 ms at a rate of 10 ms. A Hamming function is used to window the frames and a pre-emphasis with $k = 0.97$ is applied. The MFCC 0/1-12 are computed from 26 Mel-bands computed from the FFT power spectrum. The frequency range of the Mel-spectrum is set from 0 to 8 kHz. These configuration files provide the `-I` and `-O` options. The output file format is the HTK parameter file format. For other file formats you must change the ‘cHtkSink’ component type in the configuration file to the type you want. An example command-line is given here:

```
SMILEExtract -C config/MFCC12\E\D\A.conf -I input.wav -O output.mfcc.
             htk
```

2.5.3 PLP features

For extracting PLP cepstral coefficients (PLP-CC) (HTK compatible) the following four files are provided (they are named after the corresponding HTK parameter kinds they represent):

PLP_0_D_A.conf This configuration extracts Mel-frequency Cepstral Coefficients from 25 ms audio frames (sampled at a rate of 10 ms) (Hamming window). It computes 6 PLP (0-5) from 26 Mel-frequency bands using a predictor order of 5, and applies a cepstral liftering filter with a weight parameter of 22. 6 delta and 6 acceleration coefficients are appended to the PLP-CC.

PLP_E_D_A.conf This configuration is the same as PLP_0_D_A.conf, except that the log-energy is appended to the PLP 1-5 instead of the 0-th PLP.

PLP_0_D_A_Z.conf This configuration is the same as PLP_0_D_A.conf, except that the features are mean normalised with respect to the full input sequence (usually a turn or sub-turn segment).

PLP_E_D_A_Z.conf This configuration is the same as PLP_E_D_A.conf, except that the features are mean normalised with respect to the full input sequence (usually a turn or sub-turn segment).

The frame size is set to 25 ms at a rate of 10 ms. A Hamming function is used to window the frames and a pre-emphasis with $k = 0.97$ is applied. The PLP 0/1-5 are computed from 26 auditory Mel-bands (compression factor 0.33) computed from the FFT power spectrum. The predictor order of the linear predictor is 5. The frequency range of the Mel-spectrum is set from 0 to 8 kHz. These configuration files provide the `-I` and `-O` options. The output file format is the HTK parameter file format. For other file formats you must change the ‘cHtkSink’ component type in the configuration file to the type you want. An example command-line is given here:

```
SMILEExtract -C config/PLP\E\D\A.conf -I input.wav -O output.plp.htk
```

2.5.4 Prosodic features

Example configuration files for extracting prosodic features are provided in the files

`config/prosodyAcf.conf`, and `config/prosodyShs.conf`.

These files extract the fundamental frequency (F0), the voicing probability, and the loudness contours. The file `prosodyAcf.conf` uses the ‘cPitchACF’ component (section 3.3.6) to extract the fundamental frequency via an autocorrelation and cepstrum based method. The file `prosodyShs.conf` uses the ‘cPitchShs’ component (section 3.3.6) to extract the fundamental frequency via the sub-harmonic sampling algorithm (SHS). Both configurations set the CSV format as output format. An example command-line is given here:

```
SMILEExtract -C config/prosodyShs.conf -I input.wav -O prosody.csv
```

2.5.5 Extracting features for emotion recognition

Since openSMILE is used by the openEAR project [?] for emotion recognition, various standard feature sets for emotion recognition are available as openSMILE configuration files.

The INTERSPEECH 2009 Emotion Challenge feature set The INTERSPEECH 2009 Emotion Challenge feature set (see [?]) is represented by the configuration file `config/emo_IS09.conf`. It contains 384 features as statistical functionals applied to low-level descriptor contours. The features are saved in Arff format (for WEKA), whereby new instances are appended to an existing file (this is used for batch processing, where openSMILE is repeatedly called to extract features from multiple files to a single feature file). The names of the 16 low-level descriptors, as they appear in the Arff file, are documented in the following list:

pcm_RMSenergy Root-mean-square signal frame energy

mfcc Mel-Frequency cepstral coefficients 1-12

pcm_zcr Zero-crossing rate of time signal (frame-based)

voiceProb The voicing probability computed from the ACF.

F0 The fundamental frequency computed from the Cepstrum.

The suffix `_sma` appended to the names of the low-level descriptors indicates that they were smoothed by a moving average filter with window length 3. The suffix `_de` appended to `_sma` suffix indicates that the current feature is a 1st order delta coefficient (differential) of the smoothed low-level descriptor. The names of the 12 functionals, as they appear in the Arff file, are documented in the following list:

max The maximum value of the contour

min The minimum value of the contour

range = max-min

maxPos The absolute position of the maximum value (in frames)

minPos The absolute position of the minimum value (in frames)

amean The arithmetic mean of the contour

linregc1 The slope (m) of a linear approximation of the contour

linregc2 The offset (t) of a linear approximation of the contour

linregerrQ The quadratic error computed as the difference of the linear approximation and the actual contour

stddev The standard deviation of the values in the contour

skewness The skewness (3rd order moment).

kurtosis The kurtosis (4th order moment).

The INTERSPEECH 2010 Paralinguistic Challenge feature set The INTERSPEECH 2010 Paralinguistic Challenge feature set (see Proceedings of INTERSPEECH 2010) is represented by the configuration file `config/IS10.paraling.conf`. The set contains 1 582 features which result from a base of 34 low-level descriptors (LLD) with 34 corresponding delta coefficients appended, and 21 functionals applied to each of these 68 LLD contours (1 428 features). In addition, 19 functionals are applied to the 4 pitch-based LLD and their four delta coefficient contours (152 features). Finally the number of pitch onsets (pseudo syllables) and the total duration of the input are appended (2 features).

The features are saved in Arff format (for WEKA), whereby new instances are appended to an existing file (this is used for batch processing, where openSMILE is repeatedly called to extract features from multiple files to a single feature file). The names of the 34 low-level descriptors, as they appear in the Arff file, are documented in the following list:

pcm_loudness The loudness as the normalised intensity raised to a power of 0.3.

mfcc Mel-Frequency cepstral coefficients 0-14

logMelFreqBand logarithmic power of Mel-frequency bands 0 - 7 (distributed over a range from 0 to 8 kHz)

lspFreq The 8 line spectral pair frequencies computed from 8 LPC coefficients.

F0finEnv The envelope of the smoothed fundamental frequency contour.

voicingFinalUnclipped The voicing probability of the final fundamental frequency candidate. Unclipped means, that it was not set to zero when it falls below the voicing threshold.

The suffix **_sma** appended to the names of the low-level descriptors indicates that they were smoothed by a moving average filter with window length 3. The suffix **_de** appended to **_sma** suffix indicates that the current feature is a 1st order delta coefficient (differential) of the smoothed low-level descriptor. The names of the 21 functionals, as they appear in the Arff file, are documented in the following list:

maxPos The absolute position of the maximum value (in frames)

minPos The absolute position of the minimum value (in frames)

amean The arithmetic mean of the contour

linregc1 The slope (m) of a linear approximation of the contour

linregc2 The offset (t) of a linear approximation of the contour

linregerrA The linear error computed as the difference of the linear approximation and the actual contour

linregerrQ The quadratic error computed as the difference of the linear approximation and the actual contour

stddev The standard deviation of the values in the contour

skewness The skewness (3rd order moment).

kurtosis The kurtosis (4th order moment).

quartile1 The first quartile (25% percentile)

quartile2 The first quartile (50% percentile)

quartile3 The first quartile (75% percentile)

iqr1-2 The inter-quartile range: quartile2-quartile1

iqr2-3 The inter-quartile range: quartile3-quartile2

iqr1-3 The inter-quartile range: quartile3-quartile1

percentile1.0 The outlier-robust minimum value of the contour, represented by the 1% percentile.

percentile99.0 The outlier-robust maximum value of the contour, represented by the 99% percentile.

pctlrage0-1 The outlier robust signal range ‘max-min’ represented by the range of the 1% and the 99% percentile.

upleveltime75 The percentage of time the signal is above $(75\% * \text{range} + \text{min})$.

upleveltime90 The percentage of time the signal is above $(90\% * \text{range} + \text{min})$.

The four pitch related LLD (and corresponding delta coefficients) are as follows (all are 0 for unvoiced regions, thus functionals are only applied to voiced regions of these contours):

F0final The smoothed fundamental frequency contour

jitterLocal The local (frame-to-frame) Jitter (pitch period length deviations)

jitterDDP The differential frame-to-frame Jitter (the ‘Jitter of the Jitter’)

shimmerLocal The local (frame-to-frame) Shimmer (amplitude deviations between pitch periods)

19 functionals are applied to these 4+4 LLD, i.e. the set of 21 functionals mentioned above without the minimum value (the 1% percentile) and the range.

The INTERSPEECH 2011 Speaker State Challenge feature set The configuration file for this set can be found in `config/IS11_speaker_state.conf`.

Details on the feature set will be added to the openSMILE book soon. Meanwhile, we refer to the Challenge paper:

Bjrn Schuller, Anton Batliner, Stefan Steidl, Florian Schiel, Jarek Krajewski:
 ”The INTERSPEECH 2011 Speaker State Challenge”, Proc. INTERSPEECH 2011,
 ISCA, Florence, Italy, pp. 3201-3204, 28.-31.08.2011.

The INTERSPEECH 2012 Speaker Trait Challenge feature set The configuration file for this set can be found in `config/IS12_speaker_trait.conf`.

Details on the feature set will be added to the openSMILE book soon. Meanwhile, we refer to the Challenge paper:

Bjrn Schuller, Stefan Steidl, Anton Batliner, Elmar Nth, Alessandro Vinciarelli, Felix Burkhardt, Rob van Son, Felix Weninger, Florian Eyben, Tobias Bocklet, Gelareh Mohammadi, Benjamin Weiss: "The INTERSPEECH 2012 Speaker Trait Challenge", Proc. INTERSPEECH 2012, ISCA, Portland, OR, USA, 09.-13.09.2012.

The INTERSPEECH 2013 ComParE Challenge feature set The configuration file for this set can be found in `config/IS13_ComParE.conf`. A configuration that extracts only the low-level descriptors of the ComParE feature set is provided in `config/IS13_ComParE_lld.conf`. The configuration for the vocaliations (laughter, etc.) sub-challenge is also included in `config/IS13_ComParE`.

Details on the feature set will be added to the openSMILE book soon. Meanwhile, we refer to the Challenge paper:

Bjrn Schuller, Stefan Steidl, Anton Batliner, Alessandro Vinciarelli, Klaus Scherer, Fabien Ringeval, Mohamed Chetouani, Felix Weninger, Florian Eyben, Erik Marchi, Marcello Mortillaro, Hugues Salamin, Anna Polychroniou, Fabio Valente, Samuel Kim: "The INTERSPEECH 2013 Computational Paralinguistics Challenge: Social Signals, Conflict, Emotion, Autism", to appear in Proc. INTERSPEECH 2013, ISCA, Lyon, France, 2013.

The MediaEval 2012 TUM feature set for violent video scenes detection The feature set for the work on violent scenes detection in popular Hollywood style movies as presented in:

Florian Eyben, Felix Weninger, Nicolas Lehment, Gerhard Rigoll, Bjrn Schuller: "Violent Scenes Detection with Large, Brute-forced Acoustic and Visual Feature Sets", Proc. MediaEval 2012 Workshop, Pisa, Italy, 04.-05.10.2012.

can be found for various settings in `config/mediaeval2012_tum_affect`.

The file `MediaEval_Audio_IS12based_subwin2.conf` contains the configuration which extracts the static audio features from 2 second sub-windows. `MediaEval_Audio_IS12based_subwin2_step0.5` extracts the same features, but for overlapping 2 second windows with a shift of 0.5 seconds. For the video features the file `MediaEval_VideoFunctionals.conf` is provided, which requires a CSV file containing the low-level descriptors (can be extracted with openCV) as input and outputs and ARFF file with the video features as used in the paper.

The openSMILE/openEAR 'emobase' set The old baseline set (see the 'emobase2' set for the new baseline set) of 988 acoustic features for emotion recognition can be extracted using the following command:

```
SMILEExtract -C config/emobase.conf -I input.wav -O output.arff
```

This will produce an ARFF file with a header containing all the feature names and one instance, containing a feature vector for the given input file. To append more instances to the same ARFF file, simply run the above command again for different (or the same) input files. The ARFF file will have a dummy class label called emotion, containing one class *unknown* by default. To change this behaviour and assign custom classes and class labels to an individual instance, use a command-line like the following:

```
SMILEExtract -C config/emobase.conf -I inputN.wav -O output.arff -instname
inputN -classes {anger,fear,disgust} -classlabel anger
```

Thereby the parameter `-classes` specifies the list of nominal classes including the `{}` characters, or can be set to *numeric* for a numeric (regression) class. The parameter `-classlabel` specifies the class label/value of the instance computed from the currently given input (`-I`). For further information on these parameters, please take a look at the configuration file `emobase.conf` where these command-line parameters are defined.

The feature set specified by `emobase.conf` contains the following low-level descriptors (LLD): Intensity, Loudness, 12 MFCC, Pitch (F_0), Probability of voicing, F_0 envelope, 8 LSF (Line Spectral Frequencies), Zero-Crossing Rate. Delta regression coefficients are computed from these LLD, and the following functionals are applied to the LLD and the delta coefficients: Max./Min. value and respective relative position within input, range, arithmetic mean, 2 linear regression coefficients and linear and quadratic error, standard deviation, skewness, kurtosis, quartile 1–3, and 3 inter-quartile ranges.

The large openSMILE emotion feature set For extracting a larger feature set with more functionals and more LLD enabled (total 6 552 features), use the configuration file

```
config/emo_large.conf.
```

Please read the configuration file and the header of the generated arff file in conjunction with the matching parts in the component reference section (3.3) for details on the contained feature set. A documentation has to be yet written, volunteers are welcome!

The openSMILE ‘emobase2010’ reference set This feature set is based on the INTERSPEECH 2010 Paralinguistic Challenge feature set. It is represented by the file

```
config/emobase2010.conf.
```

A few tweaks have been made regarding the normalisation of duration and positional features. This feature set contains a greatly enhanced set of low-level descriptors, as well as a carefully selected list of functionals compared to the older ‘emobase’ set. This feature set is recommended as a reference for comparing new emotion recognition feature sets and approaches to, since it represents a current state-of-the-art feature set for affect and paralinguistic recognition.

The set contains 1 582 features (same as the INTERSPEECH 2010 Paralinguistic Challenge set) which result from a base of 34 low-level descriptors (LLD) with 34 corresponding delta coefficients appended, and 21 functionals applied to each of these 68 LLD contours (1 428 features). In addition, 19 functionals are applied to the 4 pitch-based LLD and their four delta coefficient contours (152 features). Finally the number of pitch onsets (pseudo syllables) and the total duration of the input are appended (2 features). The only difference to the INTERSPEECH 2010 Paralinguistic Challenge set is the normalisation of the ‘maxPos’ and ‘minPos’ features which are normalised to the segment length in the present set.

Audio-visual features based on INTERSPEECH 2010 audio features. The folder `config/audiovisual` contains two configuration files for video features (`video.conf`) and synchronised audio-visual feature extraction (`audiovideo.conf`). These files are used for the examples in section 2.7. The audio features and the set of functionals which is applied to both the audio and the video low-level features is taken from the INTERSPEECH 2010 Paralinguistic Challenge feature set (section 2.5.5 for details).

The video features contain RGB and HSV colour histograms, local binary patterns (LBP), and an optical flow histogram. They can either be extracted from the complete image or only the facial region. The latter is automatically detected via the OpenCV face detector. The face detection can be controlled in the configuration file `audiovideo.conf` in the section

```
[openCVSource:cOpenCVSource]
```

with the option `extract_face`. The number of histogram bins can also be changed in this section.

2.6 Using Portaudio for live recording/playback

The components `cPortaudioSource` and `cPortaudioSink` can be used as replacements for `cWaveSource` and `cWaveSink`. They produce/expect data in the same format as the wave components. See section 3.3.3 and section 3.3.4 for details on those two components.

Two example configuration files are provided which illustrate the basic use of PortAudio for recording live audio to file (`config/demo/audiorecorder.conf`) and for playing live audio from a file (`config/demo/audioplayer.conf`).

Using these configurations is very simple. To record audio to a file, type:

```
SMILEExtract -C config/demo/audiorecorder.conf -sampleRate 44100 -
channels 2 -O output.wave
```

To stop the recording, quit the program with Ctrl+C. To play the recorded audio use this command:

```
SMILEExtract -C config/demo/audioplayer.conf -I output.wave
```

On top of these two simple examples, a live feature extraction example is provided, which captures live audio and extracts prosodic features (pitch and loudness contours) from the input. The features are saved to a CSV file. To use this configuration, type:

```
SMILEExtract -C config/liveProsodyAcf.conf
```

The recording has started once you see the message

```
(MSG) [2] in cComponentManager : starting single thread processing loop
```

You can now speak an example sentence or play some music in front of your microphone. When you are done, press Ctrl+C to terminate openSMILE. A CSV file called `prosody.csv` has now been created in the current directory (use the `-O` command-line option to change the file name). You can now plot the loudness and pitch contours using gnuplot, for example, as is described in the next section.

2.7 Extracting features with openCv

openSMILE can extract audio and video features simultaneously and time synchronised. An example is provided in the configuration file `config/audiovisual/audiovideo.conf`.

For this example to work, you need:

- a video file in a supported format (rule of thumb: if FFMPEG can open it, openCV/openSMILE can too)
- the audio track of the video file in a separate file (.wav format)

You can use `mplayer` or `ffmpeg` for example to extract the audio track of the video. These tools often create wave files with a WAVEext header, which unfortunately is not (yet) supported by `openSMILE`. Thus, `openSMILE` will complain that the wave file is not in the correct format. If this happens you can convert the wave files with the commandline tool `sox` (available both for Linux and Windows) to a supported format:

```
sox input.wav -c 1 -2 -s output.wav
```

With some (older) versions of `sox` simply

```
sox input.wav output.wav
```

will also work. However, recent versions tend to do a simple copy operation when the source parameters match the input parameters. We thus have to change the parameters, by e.g., reducing the number of channels to 1 and possibly converting to 16-bit signed integer sample format.

Once `openSMILE` accepts the wave file, the analysis can be started by executing (all on a single line):

```
./SMILEExtract -C config/av_fusion/audiovideo.conf \  
-V VIDEO_FILE -A AUDIO_FILE -N NAME -a AGE \  
-g GENDER -e ETHNICITY -O VIDEO_ARFF -P AUDIO_ARFF
```

in a shell, whereas the following replacements should be done:

- `AUDIO_FILE` and `VIDEO_FILE` should be replaced by the path to the respective audio (.wav) and video input files (can contain the audio track, it is ignored by `OpenCV`)
- `NAME` denotes the title for the `arff` instance and can be freely chosen from alphanumeric characters and `_`.
- `AGE`, `GENDER` and `ETHNICITY` represent the ground-truth class labels for this particular pair of audio/video, if you want them to be included in an `ARFF` file, which you use to train a classifier.
- `VIDEO_ARFF` and `AUDIO_ARFF` should be replaced by the desired filename for the respective output `arff`s.

After execution, two new files will have been created: `VIDEO_ARFF` and `AUDIO_ARFF` which contain the audio and video descriptors respectively, time synchronised. If those files already exist, the content is appended accordingly.

2.8 Visualising data with Gnuplot

In order to visualise feature contours with `gnuplot`, you must have `perl5` and `gnuplot` installed. On Linux `perl` should be installed by default (if not, check your distributions documentation on how to install `perl`), and `gnuplot` can be either installed via your distribution's package manager (On Ubuntu: `sudo apt-get install gnuplot-nox`), or compiled from the source (<http://www.gnuplot.info>). For windows, `gnuplot` binaries are available from the project webpage (<http://www.gnuplot.info>). For `perl5`, download the ActiveState Perl distribution from <http://www.activestate.com/> and install it. Moreover, you will need the `bash` shell to execute the `.sh` scripts (if you don't have the `bash` shell, you must type these commands manually on the windows command-prompt).

A set of scripts, which are included with openSMILE in the directory `scripts/gnuplot`, simplifies the file conversion process and simple plotting tasks. The set of these scripts is by far not complete, but we feel this is not necessary. These script serve as templates which you can easily adjust for you own tasks. They convert (transpose) the CSV files generated by openSMILE to a representation which can be read by gnuplot directly and call gnuplot with some default plot scripts as argument.

For some ready-to-use examples, see the scripts `plotchroma.sh` to plot Chroma features as a ‘Chromagram’ and `plotaudspec.sh` to plot an auditory spectrum. The following commands give a step-by-step guide on how to plot Chroma features and an auditory spectrum (*Note*: we assume that you execute these commands in the top-level directory of the openSMILE distribution, otherwise you may need to adjust the paths):

First, you must extract chroma features from some music example file, e.g.:

```
SMILExtract -C config/chroma_fft.conf -I wav_samples/music01.wav -O
chroma.csv
```

Then you can plot them with

```
cd scripts/gnuplot
sh plotchroma.sh ../../chroma.csv
```

If your gnuplot installation is set up correctly you will now see a window with a ‘Chromagram’ plot.

For the auditory spectrum, follow these steps:

```
SMILExtract -C config/audspec.conf -I wav_samples/music01.wav -O
audspec.csv
```

Then you can plot them with

```
cd scripts/gnuplot
sh plotaudspec.sh ../../audspec.csv
```

If your gnuplot installation is set up correctly you will now see a window with a auditory spectrogram plot.

Two more universally usable scripts are also included. First we describe the `plotmatrix.sh` script. Its usage is the same as for the `plotaudspec.sh` and `plotchroma.sh` scripts:

```
cd scripts/gnuplot
sh plotmatrix.sh <filename of csv-file saved by openSMILE>
```

This script plots the matrix as a 2D-surface map with the gnuplot script `plotmatrix.gp`, thereby using a grey-scale color-map, displaying time in frames on the x-axis and the feature index on the y-axis.

To plot feature contours such as pitch or energy, first extract some example contours using the `prosodyAcf.conf` configuration, for example (You can also use the live feature extractor configuration, mentioned in the previous section):

```
SMILExtract -C config/prosodyAcf.conf -I wav_samples/speech02.wav -O
prosody.csv
```

Next, you can plot the pitch contour with

```
cd scripts/gnuplot
sh plotcontour.sh 3 ../../prosody.csv
```

The general syntax of the `plotcontour.sh` script is the following:

```
sh plotcontour.sh <index of feature to plot> <filename of CSV file
containing features>
```


The index of the feature to plot can be determined by opening the CSV file in a text editor (gedit, kate, vim, or Notepad++ on Windows, for example). The first two features in the file are always ‘frameIndex’ and ‘frameTime’ with indices 0 and 1.

Plotting of features in real-time when performing on-line feature extraction is currently not supported. However, since features are extracted incrementally anyways, it is possible to write a custom output plugin, which passes the data to some plotting application in real-time, or plots the data directly using some GUI API.

Chapter 3

Reference section

This section includes a list of components included with openSMILE and detailed documentation for each component (section 3.3). The components are grouped in logical groups on behalf of their functionality. The following section (3.1) documents available command-line options and describes the general usage of the **SMILEExtract** command-line tool. A documentation of the configuration file format can be found in section 3.2.

3.1 General usage - SMILEExtract

The **SMILEExtract** binary is a very powerful command-line utility which includes all the built-in openSMILE components. Using a single ini-style configuration file, various modes of operation can be configured. This section describes the command-line options available when calling **SMILEExtract**. Some options take an optional parameter, denoted by [parameter-type], while some require a mandatory parameter, denoted by <parameter-type>.

Usage: **SMILEExtract** [-option (value)] ...

-C, -configfile	<string> Path to openSMILE config file. <i>Default:</i> 'smile.conf'
-l, -loglevel	<int> Verbosity level of log messages (MSG, WRN, ERR, DBG) (0-9). 1: only important messages, 2,3: more detailed messages, 4,5: very detailed debug messages (if -debug is enabled), 6+: currently unused. <i>Default:</i> 2
-d, -debug	Show debug log-messages (DBG) (this is only available if the binary was compiled with the <code>_DEBUG</code> preprocessor flag) <i>Default:</i> off
-h	Show usage information and exit.
-L, -components	Show full component list (this list includes plugins, if they are detected correctly), and exit.
-H, -configHelp	[componentName:string]

Show the documentation of configuration options of all available components - including plugins - and exit. If the optional string parameter is given, then only documentation of components beginning with the given string will be shown.

-configDflt	<p>[<i>string</i>]</p> <p>Show default configuration file section templates for all components available (empty parameter), or selected components beginning with the string given as parameter. In conjunction with the 'cfgFileTemplate' option a comma separated list of components can be passed as parameter to 'configDflt', to generate a template configuration file with the listed components.</p>
-cfgFileTemplate	<p>Experimental functionality to print a configuration file template containing the components specified in a comma separated string as argument to the 'configDflt' option.</p>
-cfgFileDescriptions	<p>If this option is set, then option descriptions will be included in the generated template configuration files.</p> <p><i>Default:</i> off</p>
-c, -ccmdHelp	<p>Show user defined command-line option help in addition to the standard usage information (as printed by '-h'). Since openSMILE provides means to define additional command-line options in the configuration file, which are available only after parsing the configuration file, and additional command-line option has been introduced to show a help on these options. A typical command-line to show this help would be <code>SMILEextract -c -C myconfigfile.conf</code>.</p>
-logfile	<p><<i>string</i>></p> <p>Specifies the path and filename of the log file to use. Make sure the path of the log-file is writeable.</p> <p><i>Default:</i> 'smile.log'</p>
-appendLogfile	<p>If this option is specified, openSMILE will append log messages to an existing log-file instead of overwriting the log-file at every program start (which is the default behaviour).</p>
-nologfile	<p>If this option is specified, openSMILE does not write to a log file (use this on a read-only filesystems, for example).</p>
-noconsoleoutput	<p>If this option is specified, no log-output is displayed in the console. Logging to the log file is not affected by this option, see 'nologfile' for disabling the log-file.</p>
-t, -nticks	<p><<i>int</i>></p> <p>Number of ticks (=component loop iterations) to process (-1 = infinite) (Note: this only works for single thread processing, i.e. nThreads=1 set in the config file). This option is not intended for normal use. It is for debugging component execution code only.</p> <p><i>Default:</i> -1</p>

3.2 Understanding configuration files

openSMILE configuration files follow an INI-style file format. The file is divided into sections, which are introduced by a section header:

```
[sectionName:sectionType]
```

The section header, opposed to standard INI-format, always contains two parts, the section name (first part) and the section type (second part). The two parts of the section header are separated by a colon (:). The section body (the part after the header line up to the next header line or the end of the file) contains attributes (which are defined by the section type; a description of the available types can be seen using the `-H` command-line option as well as in section 3.3). Attributes are given as name = value pairs. An example of a generic configuration file section is given here:

```
[instancename:configType]      <— this specifies the header
variable1 = value               <— example of a string variable
variable2 = 7.8                 <— example of a "numeric" variable
variable3 = X                   <— example of a "char" variable
subconf.var1 = myname           <— example of a variable in a sub type
myarr[0] = value0               <— example of an array
myarr[1] = value1
anotherarr = value0;value1      <— example of an implicit array
noarray = value0\;value1        <— use \; to quote the separator ';'
strArr[name1] = value1          <— associative arrays, name=value pairs
strArr[name2] = value2
; line-comments may be expressed by ; // or # at the beginning
```

Principally the config type names can be any arbitrary names. However, for consistency the names of the components and their corresponding configuration type names are identical. Thus, to configure a component `cWaveSource` you need a configuration section of type `cWaveSource`.

In every openSMILE configuration file there is one mandatory section, which configures the component manager. This is the component, which instantiates and runs all other components. The following sub-section describes this section in detail.

3.2.1 Enabling components

The components which will be run, can be specified by configuring the `cComponentManager` component, as shown in the following listing (the section always has to be called `componentInstances`):

```
[componentInstances:cComponentManager]    <— don't change this
; one data memory component must always be specified!
; the default name is 'dataMemory'
; if you call your data memory instance 'dataMemory',
; you will not have to specify the reader.dmInstance variables
; for all other components!
; NOTE: you may specify more than one data memory component
; configure the default data memory:
instance[dataMemory].type=cDataMemory
; configure an example data source (name = source1):
instance[source1].type=cExampleSource
```

The associative array `instance` is used to configure the list of components. The component instance names are specified as the array keys and are freely definable. They can contain all characters except for `]`, however, it is recommended to only use alphanumeric characters, `_`, and

-. The component types (i.e. which component to instantiate), are given as value to the **type** option.

Note: for each component instance specified in the **instance** array a configuration section in the file *must* exist (*except for the data memory components!*), even if it is empty (e.g. if you want to use default values only). In this case you need to specify only the header line **[name:type]**.

3.2.2 Configuring components

The parameters of each component can be set in the configuration section corresponding to the specific component. For a wave source, for example, (which you instantiate with the line

```
instance[source1].type = cWaveSource
```

in the component manager configuration) you would add the following section (note that the name of the configuration section must match the name of the component instance, and the name of the configuration type must match the component's type name):

```
[source1:cWaveSource]
; the following sets the level this component writes to
; the level will be created by this component
; no other components may write to a level having the same name
writer.dmLevel = wave
filename = input.wav
```

This sets the file name of the wave source to **input.wav**. Further, it specifies that this wave source component should write to a data memory level called **wave**. Each openSMILE component, which processes data has at least a data reader (of type **cDataReader**), a data writer (of type **cDataWriter**), or both. These sub-components handle the interface to the data memory component(s). The most important option, which is mandatory, is **dmLevel**, which specifies the level to write to or to read from. Writing is only possible to one level and only one component may write to each level. We would like to note at this point that the levels do not have to be specified implicitly by configuring the data memory – in fact, the data memory is the only component which does not have and does not require a section in the configuration file – rather, the levels are created implicitly through **writer.dmLevel = newlevel**. Reading is possible from more than one level. Thereby, the input data will be concatenated frame-wise to one single frame containing data from all input levels. To specify reading from multiple levels, separate the level names with the array separator **;**, e.g.:

```
reader.dmLevel = level1;level2
```

The next example shows the configuration of a **cFramer** component **frame**, which creates (overlapping) frames from raw wave input, as read by the wave source:

```
[frame:cFramer]
reader.dmLevel=wave
writer.dmLevel=frames
frameSize = 0.0250
frameStep = 0.010
```

The component reads from the level **wave**, and writes to the level **frames**. It will create frames of 25 ms length at a rate of 10 ms. The actual frame length in samples depends on the sampling rate, which will be read from meta-information contained in the **wave** level. For more examples please see section 2.5.

3.2.3 Including other configuration files

To include other configuration files into the main configuration file use the following command on a separate line at the location where you want to include the other file:

```
\{path/to/config.file.to.include\}
```

This include command can be used anywhere in the configuration file (as long it is on a separate line). It simply copies the lines of the included file into the main file while loading the configuration file into openSMILE .

3.2.4 Linking to command-line options

openSMILE allows for defining of new command-line options for the **SMILEExtract** binary in the configuration file. To do so, use the `\cm` command as value, which has the following syntax:

```
\cm[longoption(shortoption){default value}:description text]
```

The command may be used as illustrated in the following example:

```
[exampleSection:exampleType]
myAttrib1 = \cm[longoption(shortopt){default}:descr. text]
myAttrib2 = \cm[longoption{default}:descr. text]
```

The **shortopt** argument and the **default value** are optional. Note that, however, either **default** and/or **descr. text** are required to define a *new* option. If neither of the two is specified, the option will not be added to the command-line parser. You can use this mode to reference options that were already added, i.e. if you want to use the value of an already existing option which has been defined at a prior location in the config file:

```
[exampleSection2:exampleType]
myAttrib2 = \cm[longoption]
```

An example for making a filename configurable via the command-line, is given here:

```
filename = \cm[filename(F){default.file}:use this option to specify the
filename for the XYZ component]
```

You can call **SMILEExtract -c -C yourconfigfile.conf** to see your command-line options appended to the general usage output.

Please note: When specifying command-line options as a value to an option, the `\cm` command is the only text allowed at the right side of the equal sign! Something like **key = value** `\cm[...]` is currently not allowed. We understand that this may be a useful feature, thus it may appear in one of the following releases. The `\cm` command may also only appear in the value field of an assignment *and (since version 2.0) also instead of a filename in the config file include command*.

3.2.5 Defining variables

This feature is not yet supported, but will be added soon. This should help avoid duplicate values and increase maintainability of configuration files.

3.2.6 Comments

Single line comments may be initiated by the following characters at the beginning of the line (only whitespaces may follow the characters): `;` `#` `//` `%`

If you want to comment out a partial line, please use `//`. Everything following the double slash on this line (and the double slash itself) will be considered a comment and will be ignored.

Multi-line comments are now supported via the C-style sequences `/*` and `*/`. In order to avoid parser problems here, please make sure these sequences are on a separate line, e.g.

```
/*
[exampleSection2:exampleType]
myAttrib2 = \cm[longoption]
*/
```

and not:

```
/*[exampleSection2:exampleType]
myAttrib2 = \cm[longoption]*/
```

The latter case is supported, however, you must ensure that the closing `*/` is *not* followed by *any* whitespaces.

3.3 Component description

This section contains detailed descriptions of the openSMILE built-in components and their configuration options. First, the abstract API base classes are listed and briefly documented. These cannot be instantiated directly, however, it is relevant to know about their existence and moreover, they play an important role if you are developing your own components (in this case, please do also consult the developer's documentation in section 4).

Please note, that some components are included with openSMILE which are not documented in this section. These components are experimental components or preliminary test components, which are not (yet) officially supported. Thus, you are advised not to spend you time on figuring out how to use them, if you do not know what the purpose of these components is. If you, on the other hand, happen to be a developer or for any other reasons you decide to work with these unfinished components, you might get some information by calling `SMILEExtract -L` to list all components compiled into your `SMILEExtract` binary and then calling `SMILEExtract -H component-name` to see the configuration parameters this component requires.

Note: The documentation provided here is from version 1.0.1. Many options have changed or have been added in version 2.0. Since openSMILE contains an excellent on-line help for each component, we decided not to further maintain this section. It will be replaced soon by a description of the algorithms implemented by each component, removing the reference parts and referring to the online help for these. Thus, generally it is a good practice to use `SMILEExtract -H component-name` to get the most up-to-date documentation of the component's capabilities. This is especially true if you are working with the latest SVN snapshot, since openSMILE's development process is quite active and things might change very quickly without this documentation being updated every time.

The following reference part of this documentation lists the stable components and describes their functionality in detail. Not listed are base classes and parent components, which cannot be instantiated directly. For those components, please refer to the developer's documentation in section 4.

The documentation syntax uses some naming conventions. A `[]` appended to an option name indicates that the option is an array option. I.e. multiple options can be separated by `;` or the elements can be addressed directly via `option[n]`, or `option[name]`, if the field is an associative array (see the textual description of the option to find out whether it is a standard array or an associative array). openSMILE has no boolean options, thus all boolean flags are represented as

integer options where a 1 indicates **true**, **enable**, or **on**, etc. and a 0 indicates **false**, **disabled**, or **off**, etc.

3.3.1 cComponentManager

Description: The component manager controls the instantiation of all components in openSMILE. Thus, it is inherently present in all configurations. Its configuration section must always be present as the first section in an openSMILE configuration file, the configuration section must be called **componentInstances**. You must use the associative array **instance[]** to configure the list of components which will be instantiated in your configuration. The configuration options are described in detail in the following:

instance[] = <object of type 'cComponentManagerInst'> Associative array storing the component list. Array indices are the instance names. Array elements must specify the type of the component to instantiate (see **type** option below).

instance[].type = <string> [Default: '(null)']
The component type to create, i.e. the name of the component class to create an instance of. All component classes begin with 'c' followed by a capital letter.

instance[].configInstance = <string> [Default: '(null)']
You will rarely have to use this option. This can be used to connect a configuration instance (a section in the configuration file) to this component instance. By default a configuration instance with the same name as the component instance will be attached to each component.

instance[].threadId = <numeric> [Default: -1]
This specifies the thread ID this component should be run in. The default is -1, which indicates that the component will be run in the first (or only) thread, or – if nThreads=0 – each component will be automatically run in a separate thread (this is untested and very experimental).

printLevelStats = <numeric> [Default: 1]
This specifies the level of detail to display the data memory configuration for debugging purposes with. Valid values 0-6. 0 disables the data memory configuration output, 6 is maximum detail.

nThreads = <numeric> [Default: 1]
This specifies the number of threads to create. A 1 configures single thread mode, a 0 configures the fully automatic multi-threading mode, where each component is run in a separate thread. Please note, that due to the overhead of creating and managing the threads, the use of multi-threading only makes sense for live demonstrator systems, or computationally complex tasks combined with large input files. Batch processing of short files does not benefit from multi-threading (on the contrary, it degrades performance!).

Class hierarchy: cComponentManager (top-level)

3.3.2 Basic data memory and interface components

This section describes the data memory (from a configuration option point of view), and the data reader and writer configuration options.

cDataMemory

Description: This is the central data memory component. It needs to be included in the component manager's instance list for every configuration file. It is the only component that has no own configuration section in the configuration file. The data memory is completely configured by the other components via their cDataWriter (section 3.3.2) and cDataReader (section 3.3.2) sub-components.

Class hierarchy: cSmileComponent → cDataMemory

Configuration options:

isRb = <numeric> [Default: 1]
The default for the isRb option for all levels.

nT = <numeric> [Default: 100]
The default level buffer size in frames for all levels.

level[] = <object of type 'cDataMemoryLevel'> See the documentation of 'cDataMemoryLevel' for more information (section 3.3.2). An associative array containing the level configuration (obsolete, you should use the cDataWriter configuration in the components that write to the dataMemory to properly configure the dataMemory!)

cDataMemoryLevel

Description: This structure specifies an optional configuration of a data memory level. If this is given, it will overwrite any defaults and inherited values from input levels. This configuration structure is part of the cDataWriter (section 3.3.2) components and cDataMemory (section 3.3.2). Normally, you should only change this configuration via the data writer that creates the level (levelconf in cDataWriter, thus all options here are prefixed with 'levelconf').

Class hierarchy: cDataMemoryLevel (top-level, sub-component of cDataMemory)

Configuration options:

levelconf.name = <string> [Default: '(null)']
The name of this data memory level, must be unique within one data memory instance.

levelconf.type = <string> [Default: 'float']
The data type of the level [can be: 'float' or 'int'(eger) , currently only float is supported by the majority of processing components]

levelconf.isRb = <numeric> [Default: 1]
Flag that indicates whether this level is a ring-buffer level (1) or not (0). I.e. this level stores the last 'nT' frames, and discards old data as new data comes in (if the old data has already been read by all registered readers; if this is not the case, the level will report that it is full to the writer attempting the write operation)

levelconf.nT = <numeric> [Default: 100]
The size of the level buffer in frames (this overwrites the 'lenSec' option)

levelconf.T = <numeric> [Default: 0]
The frame period of this level in seconds. Use a frame period of 0 for a-periodic levels (i.e. data that does not occur periodically)

levelconf.lenSec = <numeric> [Default: 0]

The size of the level buffer in seconds

levelconf.frameSizeSec = <numeric> [Default: 0]

The size of one frame in seconds. (This is generally NOT equal to 1/T, because frames may overlap)

levelconf.growDyn = <numeric> [Default: 0]

Supported currently only if 'ringbuffer=0'. If this option is set to 1, the level grows dynamically, if more memory is needed. You can use this to store live input of arbitrary length in memory. However, be aware that if openSMILE runs for a long time, it will allocate more and more memory!

levelconf.noHang = <numeric> [Default: 1]

This option controls the 'hang' behaviour for ring-buffer levels, i.e. the behaviour exhibited, when the level is full because data from the ring-buffer has not been marked as read because not all readers registered to read from this level have read data. Valid options are, 0, 1, and 2:

- 0 = always wait for readers, mark the level as full and make writes fail until all readers have read at least the next frame from this level.
- 1 = don't wait for readers, if no readers are registered, i.e. this level is a dead-end (this is the default).
- 2 = never wait for readers, writes to this level will always succeed (reads to non existing data regions might then fail), use this option with a bit of caution.

cDataReader

Description: This is the dataMemory interface component that reads data as vector or matrix from a dataMemory (section 3.3.2) component. It is used internally by all dataProcessor, dataSource, and dataSink components. A cDataReader can read from one or more data memory levels (cannot be changed during the program's run-time phase). In the latter case a single vector is returned which consists of all individual vectors concatenated. Reading from multiple levels implies waiting for data on the 'slowest' level, since only completely concatenated frames are read.

Class hierarchy: cSmileComponent → cDataReader

Configuration options:

dmInstance = <string> [Default: 'dataMemory']

The name of the cDataMemory instance this reader should connect to. This allows for complex configurations with multiple, independent data memories. For most applications the default 'dataMemory' should be reasonable. This is also the assumed default when automatically generating a configuration file.

dmLevel[] = <string> [Default: '(null)']

The level in the data memory instance specified by 'dmInstance' which to read from. If this array element contains more than one element, this reader will read data from multiple input levels, and concatenate the data to generate a single frame/vector. It is a good practice to have unique field names in all levels that you wish to concatenate. *Note:* If reading from multiple levels, the reader can only return a successfully read frame, if data

is available for reading on all input levels. If data is missing on one level, the reader cannot output data, even if data is present on the other levels.

forceAsyncMerge = **<numeric>** [Default: 0]

1/0 = yes/no : force framewise merging of levels with differing frame period, if multiple levels are specified in ‘dmLevel’.

errorOnFullInputIncomplete = **<numeric>** [Default: 1]

1/0 = yes/no : 1 = abort with an error if full input matrix reading is activated (frameSize=0 and frameStep=0 =i frameMode=full) and beginning of matrix (curR) is not 0 (if this option is set to 0, only a warning is shown).

cDataWriter

Description: This is the dataMemory (section 3.3.2) interface component that writes vector or matrix data to a dataMemory level. A writer can write only to a single level in the dataMemory, this level cannot be changed during the run-time phase.

Class hierarchy: cSmileComponent → cDataWriter

Configuration options:

dmInstance = **<string>** [Default: ‘dataMemory’]

The cDataMemory instance this writer shall connect to. This allows for complex configurations with multiple, independent data memories. For most applications the default ‘dataMemory’ should be reasonable. This is also the assumed default when automatically generating a configuration file.

dmLevel = **<string>** [Default: ‘(null)’]

The data memory level this writer will write data to. You can specify any name here, this writer will register and create a level of this name in the dataMemory during initialisation of openSMILE. Please be aware of the fact that only one writer can write to a data memory level, therefore you are not allowed to use the same name again in a ‘dmLevel’ option of any other component in the same config.

levelconf = **<object of type ‘cDataMemoryLevel’>** See the documentation of ‘cDataMemoryLevel’ for more information (section 3.3.2). This structure specifies an optional configuration of this data memory level. If this is set, it will overwrite any defaults or inherited values from input levels. For details see the help on the configuration type ‘cDataMemoryLevel’ (section 3.3.2).

3.3.3 Data sources

This section contains all data source components.

cArffSource

Description: This component reads WEKA ARFF files. The full ARFF format is not yet supported, but a simplified form, such as the files generated by the ‘cArffSink’ (section 3.3.4) component can be parsed and read. This component reads all (and only!!) ‘numeric’ or ‘real’ attributes from an ARFF file (WEKA file format) into the specified data memory level. Thereby each instance (i.e. one line in the arff file’s data section) corresponds to one frame. The frame period is 0 by default (aperiodic level), use the ‘period’ option to change this and use a fixed

period for each frame/instance. Automatic generation of frame timestamps from a ‘timestamp’ field in the ARFF file is not yet supported.

Class hierarchy: `cSmileComponent` \rightarrow `cDataSource` \rightarrow `cArffSource`

Configuration options:

writer = **<object of type ‘cDataWriter’>** The configuration of the `cDataWriter` sub-component, which handles the `dataMemory` interface for data output. See the documentation of ‘`cDataWriter`’ for more information (section 3.3.2).

bufferSize = **<numeric>** [Default: 0]
The buffer size for the output level in frames (this overwrites `bufferSize_sec`).

bufferSize_sec = **<numeric>** [Default: 0]
The buffer size for the output level in seconds.

blocksize = **<numeric>** [Default: 0]
A size of data blocks to write at once, in frames (same as `blocksizeW` for source only components, this overwrites `blocksize_sec`, if set).

blocksizeW = **<numeric>** [Default: 0]
The size of data blocks to write in frames (this overwrites `blocksize` and `blocksize_sec`, if it is set) (this option is provided for compatibility only, it is exactly the same as ‘`blocksize`’).

blocksize_sec = **<numeric>** [Default: 0]
The size of data blocks to write at once, in seconds.

blocksizeW_sec = **<numeric>** [Default: 0]
The size of data blocks to write at once, in seconds (this option overwrites `blocksize_sec`!) (this option is provided for compatibility only, it is exactly the same as ‘`blocksize`’).

period = **<numeric>** [Default: 0]
(optional) The period of the input frames, if it cannot be determined from the input file format. (if set and $\neq 0$, this will overwrite any automatically set values, from the parent level or the input file (samplerate, frame period etc.).)

filename = **<string>** [Default: ‘input.arff’]
The filename of the ARFF file to read.

skipClasses = **<numeric>** [Default: 0]
The number of numeric(!) (or real) attributes (values) at end of each instance to skip (Note: nominal and string attributes are ignored anyway, this option only applies to the last numeric attributes, even if they are followed by string or nominal attributes). To have more fine grained control over selecting attributes, please use the component `cDataSelector` (section 3.3.6)!

cCsvSource

Description: This component reads CSV (Comma separated value) files. It reads all columns as attributes into the data memory. One line represents one frame. The first line may contain a header with the feature names (see the ‘header’ option).

Class hierarchy: cSmileComponent → cDataSource → cCsvSource

Configuration options:

writer = <object of type 'cDataWriter'> The configuration of the cDataWriter sub-component, which handles the dataMemory interface for data output. See the documentation of 'cDataWriter' for more information (section 3.3.2).

bufferSize = <numeric> [Default: 0]
The buffer size for the output level in frames (this overwrites bufferSize_sec).

bufferSize_sec = <numeric> [Default: 0]
The buffer size for the output level in seconds.

blockSize = <numeric> [Default: 0]
A size of data blocks to write at once, in frames (same as blockSizeW for source only components, this overwrites blockSize_sec, if set).

blockSizeW = <numeric> [Default: 0]
The size of data blocks to write in frames (this overwrites blockSize and blockSize_sec, if it is set) (this option is provided for compatibility only, it is exactly the same as 'blockSize').

blockSize_sec = <numeric> [Default: 0]
The size of data blocks to write at once, in seconds.

blockSizeW_sec = <numeric> [Default: 0]
The size of data blocks to write at once, in seconds (this option overwrites blockSize_sec!) (this option is provided for compatibility only, it is exactly the same as 'blockSize').

period = <numeric> [Default: 0]
(optional) The period of the input frames, if it cannot be determined from the input file format. (if set and != 0, this will overwrite any automatically set values, from the parent level or the input file (samplerate, frame period etc.).)

filename = <string> [Default: 'input.csv']
The CSV file to read

delimChar = <char> [Default: ',']
The CSV delimiter character to use. Usually ',' or ';'.

header = <string> [Default: 'auto']
yes/no/auto : whether to read the first line of the CSV file as header (yes), or treat it as numeric data (no), or automatically determine from the first field in the first line whether to read the header or not (auto).

cHtkSource

Description: This component reads data from binary HTK parameter files. Each frame of the HTK file is read a frame into the openSMILE data memory. The read frame has a single array field which has 'frameSize' elements. The 'frameSize' is determined by the header of the HTK parameter file. The paramKind header field of the HTK file is ignored. The frame period is also read from the HTK file.

Class hierarchy: `cSmileComponent` \rightarrow `cDataSource` \rightarrow `cHtkSource`

Configuration options:

writer = `<object of type 'cDataWriter'>` The configuration of the `cDataWriter` sub-component, which handles the `dataMemory` interface for data output. See the documentation of 'cDataWriter' for more information (section 3.3.2).

bufferSize = `<numeric>` [Default: 0]
The buffer size for the output level in frames (this overwrites `bufferSize_sec`).

bufferSize_sec = `<numeric>` [Default: 0]
The buffer size for the output level in seconds.

blockSize = `<numeric>` [Default: 0]
A size of data blocks to write at once, in frames (same as `blockSizeW` for source only components, this overwrites `blockSize_sec`, if set).

blockSizeW = `<numeric>` [Default: 0]
The size of data blocks to write in frames (this overwrites `blockSize` and `blockSize_sec`, if it is set) (this option is provided for compatibility only, it is exactly the same as 'blockSize').

blockSize_sec = `<numeric>` [Default: 0]
The size of data blocks to write at once, in seconds.

blockSizeW_sec = `<numeric>` [Default: 0]
The size of data blocks to write at once, in seconds (this option overwrites `blockSize_sec`!) (this option is provided for compatibility only, it is exactly the same as 'blockSize').

period = `<numeric>` [Default: 0]
(optional) The period of the input frames, if it cannot be determined from the input file format. (if set and $\neq 0$, this will overwrite any automatically set values, from the parent level or the input file (samplerate, frame period etc.).)

filename = `<string>` [Default: 'input.htk']
Filename of HTK parameter file to read.

featureName = `<string>` [Default: 'htkpara']
The name of the array-field which is to be created in the data memory output level for the data array (frame) read from the HTK file.

cPortaudioSource

Description: This component handles live audio recording from the sound-card via the PortAudio library. In order to use this component, you must have a `SMILEExtract` binary linked against the PortAudio library. You can verify this, if you run `SMILEExtract -H cPortaudioSource`. If you see help on the configuration options of this component, your binary is linked against PortAudio. If not, then please refer to section 2.2 for instructions on how to build from the source with PortAudio support.

Class hierarchy: `cSmileComponent` \rightarrow `cDataSource` \rightarrow `cPortaudioSource`

Configuration options:

writer = <object of type ‘cDataWriter’> The configuration of the cDataWriter sub-component, which handles the dataMemory interface for data output. See the documentation of ‘cDataWriter’ for more information (section 3.3.2).

bufferSize = <numeric> [Default: 0]
The buffer size for the output level in frames (this overwrites bufferSize_sec).

bufferSize_sec = <numeric> [Default: 0]
The buffer size for the output level in seconds.

blockSize = <numeric> [Default: 0]
A size of data blocks to write at once, in frames (same as blockSizeW for source only components, this overwrites blockSize_sec, if set).

blockSizeW = <numeric> [Default: 0]
The size of data blocks to write in frames (this overwrites blockSize and blockSize_sec, if it is set) (this option is provided for compatibility only, it is exactly the same as ‘blockSize’).

blockSize_sec = <numeric> [Default: 0]
The size of data blocks to write at once, in seconds.

blockSizeW_sec = <numeric> [Default: 0]
The size of data blocks to write at once, in seconds (this option overwrites blockSize_sec!) (this option is provided for compatibility only, it is exactly the same as ‘blockSize’).

period = <numeric> [Default: 0]
(optional) The period of the input frames, if it cannot be determined from the input file format. (if set and != 0, this will overwrite any automatically set values, from the parent level or the input file (samplerate, frame period etc.).)

monoMixdown = <numeric> [Default: 0]
1 = mix down all recorded channels to 1 mono channel.

device = <numeric> [Default: -1]
The ID of the PortAudio device to use (device number, see the option ‘listDevices’ to get information on device numbers).

listDevices = <numeric> [Default: 0]
If set to 1, openSMILE will list available portaudio devices during initialisation phase and exit immediately after that (you might get an error message on windows, which you can ignore).

sampleRate = <numeric> [Default: 16000]
The sampling rate to use for audio recording (if supported by device!).

channels = <numeric> [Default: 1]
The number of channels to record (check your device’s capabilities!).

nBits = <numeric> [Default: 16]
The number of bits per sample and channel.

nBPS = <numeric> [Default: 0]
The number of bytes per sample and channel (0=determine automatically from nBits).

audioBuffersize = <numeric> [Default: 0]

The size of the portaudio recording buffer in samples (overwrites `audioBuffersize_sec`, if set).

audioBuffersize_sec = <numeric> [Default: 0.05]

The size of the portaudio recording buffer in seconds. This value has influence on the system latency. Setting it too high might introduce a high latency. A too low value might lead to dropped samples and reduced performance.

byteSwap = <numeric> [Default: 0]

1 = swap bytes, big endian <-> little endian (usually not required).

cSignalGenerator

Description: This component provides a signal source. This source generates various noise types and pre-defined signals and value patterns. See the configuration documentation for a list of currently implemented types.

Class hierarchy: `cSmileComponent` → `cDataSource` → `cSignalGenerator`

Configuration options:

writer = <object of type ‘cDataWriter’> The configuration of the `cDataWriter` sub-component, which handles the `dataMemory` interface for data output. See the documentation of ‘`cDataWriter`’ for more information (section 3.3.2).

buffersize = <numeric> [Default: 0]

The buffer size for the output level in frames (this overwrites `buffersize_sec`).

buffersize_sec = <numeric> [Default: 0]

The buffer size for the output level in seconds.

blocksize = <numeric> [Default: 0]

A size of data blocks to write at once, in frames (same as `blocksizeW` for source only components, this overwrites `blocksize_sec`, if set).

blocksizeW = <numeric> [Default: 0]

The size of data blocks to write in frames (this overwrites `blocksize` and `blocksize_sec`, if it is set) (this option is provided for compatibility only, it is exactly the same as ‘`blocksize`’).

blocksize_sec = <numeric> [Default: 0]

The size of data blocks to write at once, in seconds.

blocksizeW_sec = <numeric> [Default: 0]

The size of data blocks to write at once, in seconds (this option overwrites `blocksize_sec`!) (this option is provided for compatibility only, it is exactly the same as ‘`blocksize`’).

period = <numeric> [Default: 0]

The period of the output frames. You must specify this option for the signal generator component!

nFields = <numeric> [Default: 1]

The number of fields in the output vector, set to -1 to use the size of the ‘`nElements`’ array or the ‘`fieldNames`’ array, if no ‘`nElements`’ array is given.

nElements[] = <numeric> [Default: 1]

An array of number of values/elements for each field in the output vector (default is 1).

fieldNames[] = <string> [Default: '(null)']

An array of names of fields in the output vector (default for all: 'noiseN', where N is the field number).

signalType = <string> [Default: 'white']

The type of signal/noise to generate for ALL(!) output fields. If you want different types of signals for the individual fields, use multiple signalGenerator sources and combine the output vectors. Currently implemented noise and signal types are:

white pseudo-random white Gaussian noise.

const constant value output, use 'constant' parameter to set this value.

sine single sine wave (default range -1 to +1), see 'signalPeriod' or 'frequency' option and the 'phase' option.

rect rectangular periodic signal (default range -1 to +1), see 'signalPeriod' or 'frequency' option and 'phase' option.

randSeed = <numeric> [Default: 1]

Random seed, for pseudo random noise which is transformed into Gaussian white noise.

scale = <numeric> [Default: 1.0]

A scaling factor by which the generated signal is multiplied by.

const = <numeric> [Default: 0]

The constant value for the 'constant' signal type.

signalPeriod = <numeric> [Default: 1.0]

The period T in seconds of periodic signals (sine, rect, etc.) ($T = 1/f$). Don't forget to set the sample (=frame) period via 'writer.levelconf.T' or the 'period' option. Don't confuse this option with the 'period' option!

frequency = <numeric> [Default: 1.0]

The frequency f in Hz of periodic signals (sine, rect, etc.) ($f = 1/T$). This overrides the 'signalPeriod' option, if both are set. Don't forget to set the sample (=frame) period via 'writer.levelconf.T' or the 'period' option. Don't confuse this option with the 'period' option!

phase = <numeric> [Default: 0.0]

The initial phase (offset in time) of periodic signals, in seconds.

length = <numeric> [Default: 3]

The length of the signal to generate (in seconds), -1.0 for infinite.

lengthFrames = <numeric> [Default: -1]

The length of signal to generate (in frames), -1 for infinite (overwrites 'length', if set).

cWaveSource

Description: This component reads an uncompressed RIFF (PCM-WAVE) file and saves it as a stream to the data memory. For most feature extraction tasks you will require a 'cFramer' (section 3.3.6) component which chunks the input stream into short term frames.

Class hierarchy: `cSmileComponent` \rightarrow `cDataSource` \rightarrow `cWaveSource`

Configuration options:

writer = **<object of type ‘cDataWriter’>** The configuration of the `cDataWriter` subcomponent, which handles the `dataMemory` interface for data output. See the documentation of ‘`cDataWriter`’ for more information (section 3.3.2).

bufferize = **<numeric>** [Default: 0]
The buffer size for the output level in frames (this overwrites `bufferize_sec`).

bufferize_sec = **<numeric>** [Default: 0]
The buffer size for the output level in seconds.

blocksize = **<numeric>** [Default: 0]
A size of data blocks to write at once, in frames (same as `blocksizeW` for source only components, this overwrites `blocksize_sec`, if set).

blocksizeW = **<numeric>** [Default: 0]
The size of data blocks to write in frames (this overwrites `blocksize` and `blocksize_sec`, if it is set) (this option is provided for compatibility only, it is exactly the same as ‘`blocksize`’).

blocksize_sec = **<numeric>** [Default: 0]
The size of data blocks to write at once, in seconds.

blocksizeW_sec = **<numeric>** [Default: 0]
The size of data blocks to write at once, in seconds (this option overwrites `blocksize_sec`!) (this option is provided for compatibility only, it is exactly the same as ‘`blocksize`’).

period = **<numeric>** [Default: 0]
(optional) The period of the input frames, if it cannot be determined from the input file format. (if set and $\neq 0$, this will overwrite any automatically set values, from the parent level or the input file (samplerate, frame period etc.).)

filename = **<string>** [Default: ‘input.wav’]
The filename of the PCM wave file to load. Only uncompressed RIFF files are supported. Use a suitable converter (`mpplayer`, for example) to convert other formats to wave.

monoMixdown = **<numeric>** [Default: 1]
Mix down all channels to 1 mono channel (1=on, 0=off).

start = **<numeric>** [Default: 0]
The read start point in seconds from the beginning of the file.

end = **<numeric>** [Default: -1]
The read end point in seconds from the beginning of file (-1 = read to EoF).

endrel = **<numeric>** [Default: 0]
The read end point in seconds from the END of file (only if ‘end’ = -1, or not set).

startSamples = **<numeric>** [Default: 0]
The read start in samples from the beginning of the file (this overwrites ‘start’).

endSamples = <numeric> [Default: -1]

The read end in samples from the beginning of the file (this overwrites ‘end’ and ‘endrelSamples’).

endrelSamples = <numeric> [Default: 0]

The read end in samples from the END of file (this overwrites ‘endrel’).

noHeader = <numeric> [Default: 0]

1 = treat the input file as ‘raw’ format, i.e. don’t read the RIFF header. You must specify the parameters ‘sampleRate’, ‘channels’, and possibly ‘sampleSize’ if the defaults don’t match your file format.

sampleRate = <numeric> [Default: 16000]

Set/force the sampling rate that is assigned to the input data (required for reading raw files).

sampleSize = <numeric> [Default: 2]

Set/force the size of one sample (in bytes) (required for reading raw files).

channels = <numeric> [Default: 1]

Set/force the number of channels (required for reading raw files).

3.3.4 Data sinks

This section contains all data sink components, except the (on-line) classifier components.

cArffSink

Description: This component writes dataMemory data to an ARFF file (WEKA). Depending on your configuration, an instance name field, a frame index, and a frame time field can be added as well as multiple class/target attributes. See the configuration type documentation for more details.

Class hierarchy: cSmileComponent → cDataSink → cArffSink

Configuration options:

reader = <object of type ‘cDataReader’> See the documentation of ‘cDataReader’ for more information (section 3.3.2). The configuration of the cDataReader sub-component, which handles the dataMemory interface for reading of input

blocksize = <numeric> [Default: 0]

The size of the data blocks to read at once, in frames (overwrites blocksize_sec, if set).

blocksizeR = <numeric> [Default: 0]

The size of the data blocks to read at once, in frames (this overwrites blocksize and blocksize_sec!) (this option is provided for compatibility only... it is exactly the same as ‘blocksize’).

blocksize_sec = <numeric> [Default: 0]

The size of the data blocks to read at once, in seconds.

blocksizeR_sec = <numeric> [Default: 0]

The size of the data blocks to read at once, in seconds (this overwrites blocksize_sec!) (this option is provided for compatibility only... it is exactly the same as ‘blocksize’).

filename = <string> [Default: 'smileoutput.arff']

The filename of the ARFF file to write to.

append = <numeric> [Default: 0]

1 = append to an existing file, or create a new file; 0 = overwrite an existing file, or create a new file.

relation = <string> [Default: 'smile']

The name of the ARFF relation (@relation tag)

instanceBase = <string> [Default: '']

If this string is not empty, cArffSink prints an instance name attribute <instanceBase_Nr>, where *Nr* is the number (continuous index) of the current instance.

instanceName = <string> [Default: '']

If this string is not empty, cArffSink prints an instance name attribute <instanceName> for all instances.

number = <numeric> [Default: 1]

'Synonym' for the 'frameIndex' option: 1 = print an instance number (= frameIndex) attribute (continuous index) (1/0 = yes/no).

timestamp = <numeric> [Default: 1]

'Synonym' for the 'frameTime' option: 1 = print a timestamp (=frameTime) attribute (1/0 = yes/no)

frameIndex = <numeric> [Default: 1]

1 = print an instance number (= frameIndex) attribute (continuous index) (1/0 = yes/no) (same as 'number' option)

frameTime = <numeric> [Default: 1]

1 = print a timestamp (=frameTime) attribute (1/0 = yes/no) (same as 'timestamp' option)

class[] = <object of type 'arffClass'> This is an array defining the optional class target attributes (grund-truths that you want to have included in your arff file along with your features). It is an array for multiple targets/classes. See also the 'target' array.

class[].name = <string> [Default: 'class']

The name of the target attribute.

class[].type = <string> [Default: 'numeric']

The type of the target attribute: 'numeric', 'string', or nominal (= list of classes, enclosed in).

target[] = <object of type 'arffTarget'> The ground truth targets (classes) for each target (class) attribute.

target[].instance[] = <string> [Default: '']

An array containing a target for each instance.

target[].all = <string> [Default: '']

Assigns this one target to all processed instances. You can use this option if you pass only one instance to cArffSink when openSMILE is run. (This option is used by a lot batch feature extraction scripts)

cCsvSink

Description: This component exports data in CSV (comma-separated-value) format used in many spreadsheet applications. As the first line of the CSV file a header line may be printed, which contains a delimiter separated list of field names of the output values. You can also use this to output files which can be read by gnuplot, if you use the value ‘<space>’ for the ‘delimChar’ option.

Class hierarchy: cSmileComponent → cDataSink → cCsvSink

Configuration options:

reader = <object of type ‘cDataReader’> See the documentation of ‘cDataReader’ for more information (section 3.3.2). The configuration of the cDataReader sub-component, which handles the dataMemory interface for reading of input

blocksize = <numeric> [Default: 0]

The size of the data blocks to read at once, in frames (overwrites blocksize_sec, if set).

blocksizeR = <numeric> [Default: 0]

The size of the data blocks to read at once, in frames (this overwrites blocksize and blocksize_sec!) (this option is provided for compatibility only... it is exactly the same as ‘blocksize’).

blocksize_sec = <numeric> [Default: 0]

The size of the data blocks to read at once, in seconds.

blocksizeR_sec = <numeric> [Default: 0]

The size of the data blocks to read at once, in seconds (this overwrites blocksize_sec!) (this option is provided for compatibility only... it is exactly the same as ‘blocksize’).

filename = <string> [Default: ‘smileoutput.csv’]

The CSV file to write to

delimChar = <char> [Default: ‘;’]

The column delimiter character to use (usually ‘,’ or ‘;’), only 1 character is allowed. Special characters are represented by the following strings: <space> (a single space), and <tab> (a single tab).

append = <numeric> [Default: 0]

1 = append to an existing file, or create a new file; 0 = overwrite an existing file, or create a new file

timestamp = <numeric> [Default: 1]

1 = print a timestamp attribute for each output frame (1/0 = yes/no)

number = <numeric> [Default: 1]

1 = print an instance number (= frameIndex) attribute for each output frame (1/0 = yes/no)

printHeader = <numeric> [Default: 1]

1 = print a header line as the first line in the CSV file. This line contains the attribute names separated by the delimiter character.

cDatadumpSink

Description: This component writes data to a raw binary file (e.g. for Matlab import). The binary file consists of 32-bit float values representing the data values, concatenated frame by frame along the time axis. The first two float values in the file resemble the file header, and thus indicate the dimension of the matrix (1: size of frames, 2: number of frames in file). The total file size in bytes is thus $\langle \text{size of frames} \rangle \times \langle \text{number of frames} \rangle \times 4 + 2$.

Class hierarchy: cSmileComponent \rightarrow cDataSink \rightarrow cDatadumpSink

Configuration options:

reader = **<object of type ‘cDataReader’>** See the documentation of ‘cDataReader’ for more information (section 3.3.2). The configuration of the cDataReader sub-component, which handles the dataMemory interface for reading of input

blocksize = **<numeric>** [Default: 0]
The size of the data blocks to read at once, in frames (overwrites blocksize_sec, if set).

blocksizeR = **<numeric>** [Default: 0]
The size of the data blocks to read at once, in frames (this overwrites blocksize and blocksize_sec!) (this option is provided for compatibility only... it is exactly the same as ‘blocksize’).

blocksize_sec = **<numeric>** [Default: 0]
The size of the data blocks to read at once, in seconds.

blocksizeR_sec = **<numeric>** [Default: 0]
The size of the data blocks to read at once, in seconds (this overwrites blocksize_sec!) (this option is provided for compatibility only... it is exactly the same as ‘blocksize’).

filename = **<string>** [Default: ‘datadump.dat’]
The filename of the output file (if it doesn’t exist it will be created).

append = **<numeric>** [Default: 0]
1 = append to an existing file, or create a new file; 0 = overwrite an existing file, or create a new file.

cHtkSink

Description: This component writes data to a binary HTK parameter file.

Class hierarchy: cSmileComponent \rightarrow cDataSink \rightarrow cHtkSink

Configuration options:

reader = **<object of type ‘cDataReader’>** See the documentation of ‘cDataReader’ for more information (section 3.3.2). The configuration of the cDataReader sub-component, which handles the dataMemory interface for reading of input

blocksize = **<numeric>** [Default: 0]
The size of the data blocks to read at once, in frames (overwrites blocksize_sec, if set).

blocksizeR = <numeric> [Default: 0]

The size of the data blocks to read at once, in frames (this overwrites blocksize and blocksize_sec!) (this option is provided for compatibility only... it is exactly the same as 'blocksize').

blocksize_sec = <numeric> [Default: 0]

The size of the data blocks to read at once, in seconds.

blocksizeR_sec = <numeric> [Default: 0]

The size of the data blocks to read at once, in seconds (this overwrites blocksize_sec!) (this option is provided for compatibility only... it is exactly the same as 'blocksize').

filename = <string> [Default: 'smileoutput.htk']

HTK parameter file to write to.

append = <numeric> [Default: 0]

1 = append to existing file (0 = don't append).

parmKind = <numeric> [Default: 9]

HTK parmKind header field. Available target kinds (for details see the HTK book): 0=WAVEFORM, 1=LPC, 2=LPREFC, 3=LPCEPSTRA, 4=LPDELCEP, 5=IREFC, 6=MFCC, 7=FBANK (log), 8=MELSPEC (linear), 9=USER, 10=DISCRETE, 11=PLPCC). Available qualifiers (they are added to the target kind value): 64=_E (log energy), 128=_N, 256=_D, 512=_A, 1024=_C, 2048=_Z, 4096=_K, 8192=_0 (zero'th coefficient appended)

cLibsvmSink

Description: This component writes data to a text file in LibSVM feature file format. For the 'on-the-fly' classification component see 'cLibsvmLiveSink' (section 3.3.5).

Class hierarchy: cSmileComponent → cDataSink → cLibsvmSink

Configuration options:

reader = <object of type 'cDataReader'> See the documentation of 'cDataReader' for more information (section 3.3.2). The configuration of the cDataReader sub-component, which handles the dataMemory interface for reading of input

blocksize = <numeric> [Default: 0]

The size of the data blocks to read at once, in frames (overwrites blocksize_sec, if set).

blocksizeR = <numeric> [Default: 0]

The size of the data blocks to read at once, in frames (this overwrites blocksize and blocksize_sec!) (this option is provided for compatibility only... it is exactly the same as 'blocksize').

blocksize_sec = <numeric> [Default: 0]

The size of the data blocks to read at once, in seconds.

blocksizeR_sec = <numeric> [Default: 0]

The size of the data blocks to read at once, in seconds (this overwrites blocksize_sec!) (this option is provided for compatibility only... it is exactly the same as 'blocksize').

filename = <string> [Default: 'smileoutput.lsvm']

Output filename.

append = <numeric> [Default: 0]

Whether to append to an existing file or overwrite it (1/0 = append/overwrite).

timestamp = <numeric> [Default: 1]

1 = print a timestamp attribute (0 = don't print it).

instanceBase = <string> [Default: '']

If not empty, print instance name attribute <instanceBase_Nr>

instanceName = <string> [Default: '']

If not empty, print instance name attribute <instanceName>

class[] = <string> [Default: 'classX']

Optional definition of class-name strings (array for multiple classes; the array size defines the number of class columns which are appended to the CSV file). This allows for passing ground-truth information directly on the command-line and makes post-editing of feature files obsolete.

targetNum[] = <numeric> [Default: 0]

Targets/Ground truths (as numbers/indices) for each instance.

targetStr[] = <string> [Default: 'classX']

Targets/Ground truths (as strings) for each instance.

targetNumAll = <numeric> [Default: 0]

Target/Ground truth (as numbers/indices) for all instances.

targetStrAll = <string> [Default: 'classX']

Target/Ground truth (as strings) for all instances.

cNullSink

Description: This component reads data vectors from the data memory and 'destroys' them, i.e. does not write them anywhere.

Class hierarchy: cSmileComponent → cDataSink → cNullSink

Configuration options:

reader = <object of type 'cDataReader'> See the documentation of 'cDataReader' for more information (section 3.3.2). The configuration of the cDataReader sub-component, which handles the dataMemory interface for reading of input

blocksize = <numeric> [Default: 0]

The size of the data blocks to read at once, in frames (overwrites blocksize_sec, if set).

blocksizeR = <numeric> [Default: 0]

The size of the data blocks to read at once, in frames (this overwrites blocksize and blocksize_sec!) (this option is provided for compatibility only... it is exactly the same as 'blocksize').

blocksize_sec = <numeric> [Default: 0]

The size of the data blocks to read at once, in seconds.

blocksizeR_sec = <numeric> [Default: 0]

The size of the data blocks to read at once, in seconds (this overwrites blocksize_sec!) (this option is provided for compatibility only... it is exactly the same as 'blocksize').

cPortaudioSink

Description: This component handles live audio playback via the PortAudio library and the system sound-card. In order to use this component, you must have a SMILExtract binary linked against the PortAudio library. You can verify this, if you run **SMILExtract -H cPortaudioSink**. If you see help on the configuration options of this component, your binary is linked against PortAudio. If not, then please refer to section 2.2 for instructions on how to build from the source with PortAudio support.

Class hierarchy: cSmileComponent → cDataSource → cPortaudioSink

Configuration options:

reader = <object of type 'cDataReader'> See the documentation of 'cDataReader' for more information (section 3.3.2). The configuration of the cDataReader sub-component, which handles the dataMemory interface for reading of input

blocksize = <numeric> [Default: 0]

The size of the data blocks to read at once, in frames (overwrites blocksize_sec, if set).

blocksizeR = <numeric> [Default: 0]

The size of the data blocks to read at once, in frames (this overwrites blocksize and blocksize_sec!) (this option is provided for compatibility only... it is exactly the same as 'blocksize').

blocksize_sec = <numeric> [Default: 0]

The size of the data blocks to read at once, in seconds.

blocksizeR_sec = <numeric> [Default: 0]

The size of the data blocks to read at once, in seconds (this overwrites blocksize_sec!) (this option is provided for compatibility only... it is exactly the same as 'blocksize').

monoMixdown = <numeric> [Default: 0]

1 = mix down all channels to 1 mono channel.

device = <numeric> [Default: -1]

The ID of the PortAudio device to use (device number).

listDevices = <numeric> [Default: 0]

(1/0=yes/no) list available portaudio devices during initialisation phase.

sampleRate = <numeric> [Default: 0]

Force output sample rate to the given value (0=determine sample rate from input level).

audioBufferSize = <numeric> [Default: 1000]

Set the size of port the audio recording buffer in samples (overwrites audioBufferSize_sec, if set).

audioBufferSize_sec = <numeric> [Default: 0.05]

The size of the port audio recording buffer in seconds.

cWaveSink

Description: This component saves data to an uncompressed PCM WAVE file.

Class hierarchy: cSmileComponent → cDataSink → cWaveSink

Configuration options:

reader = <object of type 'cDataReader'> See the documentation of 'cDataReader' for more information (section 3.3.2). The configuration of the cDataReader sub-component, which handles the dataMemory interface for reading of input

blocksize = <numeric> [Default: 0]
The size of the data blocks to read at once, in frames (overwrites blocksize_sec, if set).

blocksizeR = <numeric> [Default: 0]
The size of the data blocks to read at once, in frames (this overwrites blocksize and blocksize_sec!) (this option is provided for compatibility only... it is exactly the same as 'blocksize').

blocksize_sec = <numeric> [Default: 0]
The size of the data blocks to read at once, in seconds.

blocksizeR_sec = <numeric> [Default: 0]
The size of the data blocks to read at once, in seconds (this overwrites blocksize_sec!) (this option is provided for compatibility only... it is exactly the same as 'blocksize').

filename = <string> [Default: 'output.wav']
The filename of the PCM wave file to write data to

sampleFormat = <string> [Default: '16bit']
openSMILE uses float for all data internally. Thus you must specify your desired sample format for the wave files here. Available formats:

- 8bit** 8-bit signed
- 16bit** 16-bit signed
- 24bit** 24-bit signed (packed in 4 bytes)
- 24bitp** 24-bit signed packed in 3 bytes
- 32bit** 32-bit signed integer
- float** 32-bit float (assumes sample value range -1...+1)

cWaveSinkCut

Description: This component writes data to uncompressed PCM WAVE files. Only chunks, based on timings received via smile messages are written to files. The files may have consecutive numbers appended to the file name.

Class hierarchy: cSmileComponent → cDataSink → cWaveSinkCut

Configuration options:

reader = <object of type 'cDataReader'> See the documentation of 'cDataReader' for more information (section 3.3.2). The configuration of the cDataReader sub-component, which handles the dataMemory interface for reading of input

blocksize = <numeric> [Default: 0]

The size of the data blocks to read at once, in frames (overwrites `blocksize_sec`, if set).

blocksizeR = <numeric> [Default: 0]

The size of the data blocks to read at once, in frames (this overwrites `blocksize` and `blocksize_sec`!) (this option is provided for compatibility only... it is exactly the same as 'blocksize').

blocksize_sec = <numeric> [Default: 0]

The size of the data blocks to read at once, in seconds.

blocksizeR_sec = <numeric> [Default: 0]

The size of the data blocks to read at once, in seconds (this overwrites `blocksize_sec`!) (this option is provided for compatibility only... it is exactly the same as 'blocksize').

fileBase = <string> [Default: 'output_segment_']

The base of the wave file name, if writing multiple output files (`multiOut=1`), or else the filename of the wave file to write to.

fileExtension = <string> [Default: '.wav']

The file extension to use when writing multiple output files (`multiOut=1`), else this option is ignored (the extension is set via 'fileBase' then which specifies the full file name).

fileNameFormatString = <string> [Default: '%s%04d%s']

Specifies how the filename will be formatted (*printf* compatible syntax, three parameters are available in the given order: `fileBase` (string), current index (integer), `fileExtension` (string)), the default should be reasonable, it generates filenames such as 'output_segment_XXXX.wav'.

startIndex = <numeric> [Default: 1]

The index of the first file for consecutive numbering of output files (if `multiOut=1`).

preSil = <numeric> [Default: 0.2]

Specifies the amount of silence at the turn beginning in seconds, i.e. the lag of the turn detector. This is the length of the data that will be added to the current segment prior to the turn start time received in the message from the turn detector component.

postSil = <numeric> [Default: 0.3]

Specifies the amount of silence at the turn end in seconds. This is the length of the data that will be added to the current segment after to the turn end time received in the message from the turn detector component.

multiOut = <numeric> [Default: 1]

1 = enable multiple file mode, i.e. multiple files segmented by `turnStart/turnEnd` messages ; 0 = write all frames (only between `turnStart/turnEnd` messages) concatenated to one file, i.e. effectively filtering out non-turn audio.

sampleFormat = <string> [Default: '16bit']

openSMILE uses float for all data internally. Thus you must specify your desired sample format for the wave files here. Available formats:

8bit 8-bit signed

16bit 16-bit signed
24bit 24-bit signed (packed in 4 bytes)
24bitp 24-bit signed packed in 3 bytes
32bit 32-bit signed integer
float 32-bit float (assumes sample value range -1...+1)

3.3.5 Live data sinks (classifiers)

This section contains the live data sink components (e.g. classifiers).

cLibsvmLiveSink

Description: This component classifies data frames ‘on-the-fly’ using the LibSVM library. Loading of ASCII and binary LibSVM models is supported (binary models are a LibSVM extension currently only included and distributed with openSMILE), as well as applying LibSVM scale files and openSMILE feature selection lists.

Class hierarchy: cSmileComponent → cDataSink → cLibsvmLiveSink

Configuration options:

reader = <object of type ‘cDataReader’> See the documentation of ‘cDataReader’ for more information (section 3.3.2). The configuration of the cDataReader sub-component, which handles the dataMemory interface for reading of input

blocksize = <numeric> [Default: 0]
 The size of the data blocks to read at once, in frames (overwrites blocksize_sec, if set).

blocksizeR = <numeric> [Default: 0]
 The size of the data blocks to read at once, in frames (this overwrites blocksize and blocksize_sec!) (this option is provided for compatibility only... it is exactly the same as ‘blocksize’).

blocksize_sec = <numeric> [Default: 0]
 The size of the data blocks to read at once, in seconds.

blocksizeR_sec = <numeric> [Default: 0]
 The size of the data blocks to read at once, in seconds (this overwrites blocksize_sec!) (this option is provided for compatibility only... it is exactly the same as ‘blocksize’).

model[] = <string> [Default: ‘svm.model’]
 LibSVM model file(s) to load. When loading multiple models, the currently active model, scaling, and feature selection, can be changed by other components via smile messages.

scale[] = <string> [Default: ‘svm.scale’]
 LibSVM scale file(s) to load. When loading multiple models, the currently active model, scaling, and feature selection, can be changed by other components via smile messages.

fselection[] = <string> [Default: ‘(null)’]
 Feature selection file(s) to apply (leave empty to use all features). The feature selection files must contain the exact names of the selected features, one feature per line. When loading multiple models, the currently active model, scaling, and feature selection, can be changed by other components via smile messages.

classes[] = <string> [Default: '(null)']

Class name lookup file(s), which map the libsvm class indices to actual class names (leave empty to display libsvm class numbers/indices) [note: currently only *one* class name lookup file is supported, i.e. only the first file in the array is loaded!].

predictProbability = <numeric> [Default: 0]

1 = predict class probabilities (confidences) (0 = no). The loaded model(s) must support probability inference.

printResult = <numeric> [Default: 0]

1 = print the classification/regression result to the console.

resultRecp = <string> [Default: '(null)']

A list of component(s) to send the 'classificationResult' smile messages to (use , to separate multiple recipients). Leave blank (NULL) to not send any messages.

resultMessageName = <string> [Default: 'svm_result']

Freely definable name that is sent with 'classificationResult' message. This name can be used to distinguish the messages when multiple classifiers for different tasks are present.

forceScale = <numeric> [Default: 1]

1 = for the input values, enforce the range specified in the scale file by clipping out-of-range values (after scaling).

lag = <numeric> [Default: 0]

If > 0, read data <lag> frames behind (should always remain 0 for this component, to classify the most recent frames).

3.3.6 Low-level features and signal processing

This section contains all low-level feature extractors and various signal processing components.

cAcf

Description: This component computes the autocorrelation function (ACF) by squaring a magnitude spectrum and applying an inverse Fast Fourier Transform. This component must read from a level containing *only* FFT magnitudes in a single field. Use the 'cTransformFFT' (section 3.3.6) and 'cFFTmagphase' (section 3.3.6) components to compute the magnitude spectrum from PCM frames. Computation of the Cepstrum is also supported (this applies a log() function to the magnitude spectra).

Class hierarchy: cSmileComponent → cDataProcessor → cVectorProcessor → cAcf

Configuration options:

reader = <object of type 'cDataReader'> The configuration of the cDataReader sub-component, which handles the dataMemory interface for data input. See the documentation of 'cDataReader' for more information (section 3.3.2).

writer = <object of type 'cDataWriter'> The configuration of the cDataWriter sub-component, which handles the dataMemory interface for data output. See the documentation of 'cDataWriter' for more information (section 3.3.2).

bufferSize = <numeric> [Default: 0]

The buffer size for the output level, in frames (default [0] = same as input level), this option overwrites 'bufferSize_sec'.

bufferSize_sec = <numeric> [Default: 0]

The buffer size for the output level, in seconds (default [0] = same as input level).

blockSize = <numeric> [Default: 0]

The size of data blocks to process, in frames (this sets both 'blockSizeR' and 'blockSizeW', and overwrites 'blockSize_sec').

blockSizeR = <numeric> [Default: 0]

The size of data blocks to read, in frames (overwrites 'blockSize').

blockSizeW = <numeric> [Default: 0]

The size of data blocks to write, in frames (overwrites 'blockSize').

blockSize_sec = <numeric> [Default: 0]

The size of data blocks to process, in seconds (this sets both 'blockSizeR_sec' and 'blockSizeW_sec').

blockSizeR_sec = <numeric> [Default: 0]

The size of data blocks to read, in seconds (overwrites 'blockSize_sec').

blockSizeW_sec = <numeric> [Default: 0]

The size of data blocks to write in seconds (overwrites 'blockSize_sec').

nameAppend = <string> [Default: 'acf']

A string suffix to append to the input field names (default: empty).

copyInputName = <numeric> [Default: 1]

1 = copy the input name (and optionally append a suffix, see the 'nameAppend' option)
0 = discard the input name and use only the 'nameAppend' string as new name.

processArrayFields = <numeric> [Default: 1]

1 = process each array field as one vector individually (and produce one output for each input array field). Only array fields (i.e. fields with more than one element) are processed if this is set.

0 = process complete input frame as one vector, ignoring the field/element structure.

usePower = <numeric> [Default: 1]

1 = square input values; you must leave this at default 1, when using FFT magnitude as input.

cepstrum = <numeric> [Default: 0]

1 = compute the Cepstrum instead of the ACF. This applies a log() to the magnitudes before transforming from the spectral domain back to the time domain. You might want to set 'nameAppend=cepstrum' when using this option.

cAmdf

Description: This component computes the Average Magnitude Difference Function (AMDF) for each input frame. Various methods for padding or warping at the border exist.

Class hierarchy: cSmileComponent → cDataProcessor → cVectorProcessor → cAmdf

Configuration options:

reader = <object of type ‘cDataReader’> The configuration of the cDataReader sub-component, which handles the dataMemory interface for data input. See the documentation of ‘cDataReader’ for more information (section 3.3.2).

writer = <object of type ‘cDataWriter’> The configuration of the cDataWriter sub-component, which handles the dataMemory interface for data output. See the documentation of ‘cDataWriter’ for more information (section 3.3.2).

bufferSize = <numeric> [Default: 0]
The buffer size for the output level, in frames (default [0] = same as input level), this option overwrites ‘bufferSize_sec’.

bufferSize_sec = <numeric> [Default: 0]
The buffer size for the output level, in seconds (default [0] = same as input level).

blockSize = <numeric> [Default: 0]
The size of data blocks to process, in frames (this sets both ‘blockSizeR’ and ‘blockSizeW’, and overwrites ‘blockSize_sec’).

blockSizeR = <numeric> [Default: 0]
The size of data blocks to read, in frames (overwrites ‘blockSize’).

blockSizeW = <numeric> [Default: 0]
The size of data blocks to write, in frames (overwrites ‘blockSize’).

blockSize_sec = <numeric> [Default: 0]
The size of data blocks to process, in seconds (this sets both ‘blockSizeR_sec’ and ‘blockSizeW_sec’).

blockSizeR_sec = <numeric> [Default: 0]
The size of data blocks to read, in seconds (overwrites ‘blockSize_sec’).

blockSizeW_sec = <numeric> [Default: 0]
The size of data blocks to write in seconds (overwrites ‘blockSize_sec’).

nameAppend = <string> [Default: ‘amdf’]
A string suffix to append to the input field names (default: empty).

copyInputName = <numeric> [Default: 1]
1 = copy the input name (and optionally append a suffix, see the ‘nameAppend’ option)
0 = discard the input name and use only the ‘nameAppend’ string as new name.

processArrayFields = <numeric> [Default: 1]
1 = process each array field as one vector individually (and produce one output for each input array field). Only array fields (i.e. fields with more than one element) are processed

if this is set.

0 = process complete input frame as one vector, ignoring the field/element structure.

nLag = <numeric> [Default: 0]

If $nLag > 0$, compute the AMDF up to maximum lag 'nLag' (or maximum possible lag limited by the frame size). If $nLag=0$, then always the maximum frame size will be used (this is the default). If $nLag < 0$, then $nLag=framesize/((-1)*nLag)$ will be used.

method = <string> [Default: 'limit']

The AMDF computation method (i.e. the handling of border conditions):

limit compute the AMDF only in regions where the shifted windows overlap

warp compute a cyclical AMDF by warping of the input

zeropad zero-pad the missing samples

invert = <numeric> [Default: 0]

1 = invert the AMDF output values (literally '1 minus AMDF'), i.e. so that the behaviour of the AMDF output corresponds more to that of an autocorrelation function.

cBuffer

Description: This resembles a simple data buffer. Its only function is to copy data from one (or more) level(s) to another level. When specifying more than one input level, this component can be used to concatenate frames.

Class hierarchy: cSmileComponent → cDataProcessor → cBuffer

Configuration options:

reader = <object of type 'cDataReader'> The configuration of the cDataReader sub-component, which handles the dataMemory interface for data input. See the documentation of 'cDataReader' for more information (section 3.3.2).

writer = <object of type 'cDataWriter'> The configuration of the cDataWriter sub-component, which handles the dataMemory interface for data output. See the documentation of 'cDataWriter' for more information (section 3.3.2).

bufferSize = <numeric> [Default: 0]

The buffer size for the output level, in frames (default [0] = same as input level), this option overwrites 'bufferSize_sec'.

bufferSize_sec = <numeric> [Default: 0]

The buffer size for the output level, in seconds (default [0] = same as input level).

blockSize = <numeric> [Default: 0]

The size of data blocks to process, in frames (this sets both 'blockSizeR' and 'blockSizeW', and overwrites 'blockSize_sec').

blockSizeR = <numeric> [Default: 0]

The size of data blocks to read, in frames (overwrites 'blockSize').

blockSizeW = <numeric> [Default: 0]

The size of data blocks to write, in frames (overwrites 'blockSize').

blocksize_sec = <numeric> [Default: 0]

The size of data blocks to process, in seconds (this sets both ‘blocksizeR_sec’ and ‘blocksizeW_sec’).

blocksizeR_sec = <numeric> [Default: 0]

The size of data blocks to read, in seconds (overwrites ‘blocksize_sec’).

blocksizeW_sec = <numeric> [Default: 0]

The size of data blocks to write in seconds (overwrites ‘blocksize_sec’).

nameAppend = <string> [Default: ‘(null)’]

A string suffix to append to the input field names (default: empty).

copyInputName = <numeric> [Default: 1]

1 = copy the input name (and optionally append a suffix, see the ‘nameAppend’ option)

0 = discard the input name and use only the ‘nameAppend’ string as new name.

vecCopy = <numeric> [Default: 1]

1 = copy vectors/frames one by one

0 = copy a window/matrix of size ‘copyBuf’

copyBuf = <numeric> [Default: 1024]

The size of the copy-at-once buffer in frames, if ‘vecCopy’=1

cCens

Description: This component computes CENS (energy normalised and smoothed discretised Chroma features) from raw Chroma features generated by the ‘cChroma’ (section 3.3.6) component.

Class hierarchy: cSmileComponent → cDataProcessor → cVectorProcessor → cCens

Configuration options:

reader = <object of type ‘cDataReader’> The configuration of the cDataReader sub-component, which handles the dataMemory interface for data input. See the documentation of ‘cDataReader’ for more information (section 3.3.2).

writer = <object of type ‘cDataWriter’> The configuration of the cDataWriter sub-component, which handles the dataMemory interface for data output. See the documentation of ‘cDataWriter’ for more information (section 3.3.2).

bufferize = <numeric> [Default: 0]

The buffer size for the output level, in frames (default [0] = same as input level), this option overwrites ‘bufferize_sec’.

bufferize_sec = <numeric> [Default: 0]

The buffer size for the output level, in seconds (default [0] = same as input level).

blocksize = <numeric> [Default: 0]

The size of data blocks to process, in frames (this sets both ‘blocksizeR’ and ‘blocksizeW’, and overwrites ‘blocksize_sec’).

blocksizeR = <numeric> [Default: 0]

The size of data blocks to read, in frames (overwrites ‘blocksize’).

blocksizeW = <numeric> [Default: 0]

The size of data blocks to write, in frames (overwrites ‘blocksize’).

blocksize_sec = <numeric> [Default: 0]

The size of data blocks to process, in seconds (this sets both ‘blocksizeR_sec’ and ‘blocksizeW_sec’).

blocksizeR_sec = <numeric> [Default: 0]

The size of data blocks to read, in seconds (overwrites ‘blocksize_sec’).

blocksizeW_sec = <numeric> [Default: 0]

The size of data blocks to write in seconds (overwrites ‘blocksize_sec’).

nameAppend = <string> [Default: ‘chroma’]

A string suffix to append to the input field names.

copyInputName = <numeric> [Default: 0]

1 = copy the input name (and optionally append a suffix, see the ‘nameAppend’ option)

0 = discard the input name and use only the ‘nameAppend’ string as new name.

processArrayFields = <numeric> [Default: 1]

1 = process each array field as one vector individually (and produce one output for each input array field). Only array fields (i.e. fields with more than one element) are processed if this is set.

0 = process complete input frame as one vector, ignoring the field/element structure.

window = <string> [Default: ‘han’]

The window function to use for temporal smoothing of discretised Chroma features; one of these: han (Hanning), ham (Hamming), bar (Bartlett)

downsampleRatio = <numeric> [Default: 10]

The integer ratio at which to downsample the resulting sequence of vectors. I.e. a value of 10 will average 10 frames and output 1 CENS frame. This value should be smaller than winlength (it need not, though).

winlength = <numeric> [Default: 41]

The length of the CENS smoothing window (see ‘window’ option), in frames.

winlength_sec = <numeric> [Default: 0.41]

The length of the CENS smoothing window, in seconds. This will be rounded upwards (ceil) to the closest length in frames. It overrides winlength, if set.

l2norm = <numeric> [Default: 1]

1/0 = enable/disable normalisation of CENS output vectors by their L2-norm.

cChroma

Description: This component computes CHROMA features from a semi-tone scaled spectrum generated by the ‘cTonespec’ (section 3.3.6) component.

Class hierarchy: cSmileComponent → cDataProcessor → cVectorProcessor → cChroma

Configuration options:

reader = <object of type ‘cDataReader’> The configuration of the cDataReader sub-component, which handles the dataMemory interface for data input. See the documentation of ‘cDataReader’ for more information (section 3.3.2).

writer = <object of type ‘cDataWriter’> The configuration of the cDataWriter sub-component, which handles the dataMemory interface for data output. See the documentation of ‘cDataWriter’ for more information (section 3.3.2).

bufferSize = <numeric> [Default: 0]
The buffer size for the output level, in frames (default [0] = same as input level), this option overwrites ‘bufferSize_sec’.

bufferSize_sec = <numeric> [Default: 0]
The buffer size for the output level, in seconds (default [0] = same as input level).

blockSize = <numeric> [Default: 0]
The size of data blocks to process, in frames (this sets both ‘blockSizeR’ and ‘blockSizeW’, and overwrites ‘blockSize_sec’).

blockSizeR = <numeric> [Default: 0]
The size of data blocks to read, in frames (overwrites ‘blockSize’).

blockSizeW = <numeric> [Default: 0]
The size of data blocks to write, in frames (overwrites ‘blockSize’).

blockSize_sec = <numeric> [Default: 0]
The size of data blocks to process, in seconds (this sets both ‘blockSizeR_sec’ and ‘blockSizeW_sec’).

blockSizeR_sec = <numeric> [Default: 0]
The size of data blocks to read, in seconds (overwrites ‘blockSize_sec’).

blockSizeW_sec = <numeric> [Default: 0]
The size of data blocks to write in seconds (overwrites ‘blockSize_sec’).

nameAppend = <string> [Default: ‘chroma’]
A string suffix to append to the input field names.

copyInputName = <numeric> [Default: 0]
1 = copy the input name (and optionally append a suffix, see the ‘nameAppend’ option)
0 = discard the input name and use only the ‘nameAppend’ string as new name.

processArrayFields = <numeric> [Default: 1]
1 = process each array field as one vector individually (and produce one output for each input array field). Only array fields (i.e. fields with more than one element) are processed if this is set.
0 = process complete input frame as one vector, ignoring the field/element structure.

octaveSize = <numeric> [Default: 12]
The size of an octave, i.e. the number of output bins, or the interval to which the input bins are mapped via warping.

cChordFeatures

Description: Chord features (computed from Chroma features (see section 3.3.6)). **FIXME:** write extended feature documentation.

Class hierarchy: cSmileComponent → cDataProcessor → cVectorProcessor → cChordFeatures

Configuration options:

reader = <object of type 'cDataReader'> The configuration of the cDataReader sub-component, which handles the dataMemory interface for data input. See the documentation of 'cDataReader' for more information (section 3.3.2).

writer = <object of type 'cDataWriter'> The configuration of the cDataWriter sub-component, which handles the dataMemory interface for data output. See the documentation of 'cDataWriter' for more information (section 3.3.2).

bufferSize = <numeric> [Default: 0]
The buffer size for the output level, in frames (default [0] = same as input level), this option overwrites 'bufferSize_sec'.

bufferSize_sec = <numeric> [Default: 0]
The buffer size for the output level, in seconds (default [0] = same as input level).

blockSize = <numeric> [Default: 0]
The size of data blocks to process, in frames (this sets both 'blockSizeR' and 'blockSizeW', and overwrites 'blockSize_sec').

blockSizeR = <numeric> [Default: 0]
The size of data blocks to read, in frames (overwrites 'blockSize').

blockSizeW = <numeric> [Default: 0]
The size of data blocks to write, in frames (overwrites 'blockSize').

blockSize_sec = <numeric> [Default: 0]
The size of data blocks to process, in seconds (this sets both 'blockSizeR_sec' and 'blockSizeW_sec').

blockSizeR_sec = <numeric> [Default: 0]
The size of data blocks to read, in seconds (overwrites 'blockSize_sec').

blockSizeW_sec = <numeric> [Default: 0]
The size of data blocks to write in seconds (overwrites 'blockSize_sec').

nameAppend = <string> [Default: 'chordbasedfeature']
A string suffix to append to the input field names.

copyInputName = <numeric> [Default: 1]
1 = copy the input name (and optionally append a suffix, see the 'nameAppend' option)
0 = discard the input name and use only the 'nameAppend' string as new name.

processArrayFields = <numeric> [Default: 1]
1 = process each array field as one vector individually (and produce one output for each input array field). Only array fields (i.e. fields with more than one element) are processed if this is set.
0 = process complete input frame as one vector, ignoring the field/element structure.

cChromaFeatures

Description: Chroma-based features (computed from Chroma features (see section 3.3.6)).
FIXME: write extended feature documentation. *Warning:* this component does not compute Chroma features (see the cChroma component for this)!

Class hierarchy: cSmileComponent → cDataProcessor → cVectorProcessor → cChromaFeatures

Configuration options:

reader = <object of type 'cDataReader'> The configuration of the cDataReader sub-component, which handles the dataMemory interface for data input. See the documentation of 'cDataReader' for more information (section 3.3.2).

writer = <object of type 'cDataWriter'> The configuration of the cDataWriter sub-component, which handles the dataMemory interface for data output. See the documentation of 'cDataWriter' for more information (section 3.3.2).

bufferSize = <numeric> [Default: 0]
 The buffer size for the output level, in frames (default [0] = same as input level), this option overwrites 'bufferSize_sec'.

bufferSize_sec = <numeric> [Default: 0]
 The buffer size for the output level, in seconds (default [0] = same as input level).

blockSize = <numeric> [Default: 0]
 The size of data blocks to process, in frames (this sets both 'blockSizeR' and 'blockSizeW', and overwrites 'blockSize_sec').

blockSizeR = <numeric> [Default: 0]
 The size of data blocks to read, in frames (overwrites 'blockSize').

blockSizeW = <numeric> [Default: 0]
 The size of data blocks to write, in frames (overwrites 'blockSize').

blockSize_sec = <numeric> [Default: 0]
 The size of data blocks to process, in seconds (this sets both 'blockSizeR_sec' and 'blockSizeW_sec').

blockSizeR_sec = <numeric> [Default: 0]
 The size of data blocks to read, in seconds (overwrites 'blockSize_sec').

blockSizeW_sec = <numeric> [Default: 0]
 The size of data blocks to write in seconds (overwrites 'blockSize_sec').

nameAppend = <string> [Default: 'chromabasedfeature']
 A string suffix to append to the input field names.

copyInputName = <numeric> [Default: 1]
 1 = copy the input name (and optionally append a suffix, see the 'nameAppend' option)
 0 = discard the input name and use only the 'nameAppend' string as new name.

processArrayFields = <numeric> [Default: 1]
 1 = process each array field as one vector individually (and produce one output for each

input array field). Only array fields (i.e. fields with more than one element) are processed if this is set.

0 = process complete input frame as one vector, ignoring the field/element structure.

cContourSmoother

Description: This component smooths data contours by applying a moving average filter of configurable length.

Class hierarchy: cSmileComponent → cDataProcessor → cWindowProcessor → cContourSmoother

Configuration options:

reader = <object of type 'cDataReader'> The configuration of the cDataReader sub-component, which handles the dataMemory interface for data input. See the documentation of 'cDataReader' for more information (section 3.3.2).

writer = <object of type 'cDataWriter'> The configuration of the cDataWriter sub-component, which handles the dataMemory interface for data output. See the documentation of 'cDataWriter' for more information (section 3.3.2).

bufferize = <numeric> [Default: 0]
The buffer size for the output level, in frames (default [0] = same as input level), this option overwrites 'bufferize_sec'.

bufferize_sec = <numeric> [Default: 0]
The buffer size for the output level, in seconds (default [0] = same as input level).

blocksize = <numeric> [Default: 0]
The size of data blocks to process, in frames (this sets both 'blocksizeR' and 'blocksizeW', and overwrites 'blocksize_sec').

blocksizeR = <numeric> [Default: 0]
The size of data blocks to read, in frames (overwrites 'blocksize').

blocksizeW = <numeric> [Default: 0]
The size of data blocks to write, in frames (overwrites 'blocksize').

blocksize_sec = <numeric> [Default: 0]
The size of data blocks to process, in seconds (this sets both 'blocksizeR_sec' and 'blocksizeW_sec').

blocksizeR_sec = <numeric> [Default: 0]
The size of data blocks to read, in seconds (overwrites 'blocksize_sec').

blocksizeW_sec = <numeric> [Default: 0]
The size of data blocks to write in seconds (overwrites 'blocksize_sec').

nameAppend = <string> [Default: 'sma']
A string suffix to append to the input field names.

copyInputName = <numeric> [Default: 1]
1 = copy the input name (and optionally append a suffix, see the 'nameAppend' option)
0 = discard the input name and use only the 'nameAppend' string as new name.

noPostEOIprocessing = <numeric> [Default: 0]

1 = do not process incomplete windows at the end of the input.

smaWin = <numeric> [Default: 3]

The size of the moving average window in frames. A larger window means more smoothing.

cDataSelector

Description: This component copies data from one level to another, thereby selecting frame fields and elements by their element/field name (use this for feature selection, if you have a list of selected features in a text file, where each line contains the name of exactly one feature).

Class hierarchy: cSmileComponent → cDataProcessor → cDataSelector

Configuration options:

reader = <object of type ‘cDataReader’> The configuration of the cDataReader sub-component, which handles the dataMemory interface for data input. See the documentation of ‘cDataReader’ for more information (section 3.3.2).

writer = <object of type ‘cDataWriter’> The configuration of the cDataWriter sub-component, which handles the dataMemory interface for data output. See the documentation of ‘cDataWriter’ for more information (section 3.3.2).

bufferSize = <numeric> [Default: 0]

The buffer size for the output level, in frames (default [0] = same as input level), this option overwrites ‘bufferSize_sec’.

bufferSize_sec = <numeric> [Default: 0]

The buffer size for the output level, in seconds (default [0] = same as input level).

blockSize = <numeric> [Default: 0]

The size of data blocks to process, in frames (this sets both ‘blockSizeR’ and ‘blockSizeW’, and overwrites ‘blockSize_sec’).

blockSizeR = <numeric> [Default: 0]

The size of data blocks to read, in frames (overwrites ‘blockSize’).

blockSizeW = <numeric> [Default: 0]

The size of data blocks to write, in frames (overwrites ‘blockSize’).

blockSize_sec = <numeric> [Default: 0]

The size of data blocks to process, in seconds (this sets both ‘blockSizeR_sec’ and ‘blockSizeW_sec’).

blockSizeR_sec = <numeric> [Default: 0]

The size of data blocks to read, in seconds (overwrites ‘blockSize_sec’).

blockSizeW_sec = <numeric> [Default: 0]

The size of data blocks to write in seconds (overwrites ‘blockSize_sec’).

nameAppend = <string> [Default: ‘(null)’]

A string suffix to append to the input field names (default: empty).

copyInputName = <numeric> [Default: 1]

1 = copy the input name (and optionally append a suffix, see the ‘nameAppend’ option)

0 = discard the input name and use only the ‘nameAppend’ string as new name.

selFile = <string> [Default: ‘(null)’]

The name of the data selection file to load. The file is a text file containing one element name per line of the elements which should be selected (case-sensitive!). (Note: the first two lines make up a header; the first line always contains ‘str’, the second line is number of features in list, next each line contains one feature name of the features to select)

selected[] = <string> [Default: ‘(null)’]

This is an alternative to loading ‘selFile’. An array of exact (case-sensitive) names of features / data elements to select.

newNames[] = <string> [Default: ‘(null)’]

An array of new names to assign to the selected features / data elements (optional). The order thereby corresponds to the order of data element names in the input.

elementMode = <numeric> [Default: 1]

1 = select elements exactly as given in the ‘selected’ array or in ‘selFile’ (in this case, only full element names are allowed (i.e. mfcc[1], mfcc[2] instead of mfcc, mfcc[], or mfcc[1-2]).

0 = automatically copy arrays or partial arrays, e.g. if field[1-4] or only ‘field’ is given as name in the selection array/file, then the partial (1-4) or complete field will be copied to the output.

cDbA

Description: This component performs dB(X) (dB(A),dB(B),dB(C),...) equal loudness weighting of FFT bin magnitudes. Currently only dB(A) weighting is implemented.

Class hierarchy: cSmileComponent → cDataProcessor → cVectorProcessor → cDbA

Configuration options:

reader = <object of type ‘cDataReader’> The configuration of the cDataReader sub-component, which handles the dataMemory interface for data input. See the documentation of ‘cDataReader’ for more information (section 3.3.2).

writer = <object of type ‘cDataWriter’> The configuration of the cDataWriter sub-component, which handles the dataMemory interface for data output. See the documentation of ‘cDataWriter’ for more information (section 3.3.2).

bufferSize = <numeric> [Default: 0]

The buffer size for the output level, in frames (default [0] = same as input level), this option overwrites ‘bufferSize_sec’.

bufferSize_sec = <numeric> [Default: 0]

The buffer size for the output level, in seconds (default [0] = same as input level).

blocksize = <numeric> [Default: 0]

The size of data blocks to process, in frames (this sets both ‘blocksizeR’ and ‘blocksizeW’, and overwrites ‘blocksize_sec’).

blocksizeR = <numeric> [Default: 0]

The size of data blocks to read, in frames (overwrites ‘blocksize’).

blocksizeW = <numeric> [Default: 0]

The size of data blocks to write, in frames (overwrites ‘blocksize’).

blocksize_sec = <numeric> [Default: 0]

The size of data blocks to process, in seconds (this sets both ‘blocksizeR_sec’ and ‘blocksizeW_sec’).

blocksizeR_sec = <numeric> [Default: 0]

The size of data blocks to read, in seconds (overwrites ‘blocksize_sec’).

blocksizeW_sec = <numeric> [Default: 0]

The size of data blocks to write in seconds (overwrites ‘blocksize_sec’).

nameAppend = <string> [Default: ‘(null)’]

A string suffix to append to the input field names (default: empty).

copyInputName = <numeric> [Default: 1]

1 = copy the input name (and optionally append a suffix, see the ‘nameAppend’ option)

0 = discard the input name and use only the ‘nameAppend’ string as new name.

processArrayFields = <numeric> [Default: 1]

1 = process each array field as one vector individually (and produce one output for each input array field). Only array fields (i.e. fields with more than one element) are processed if this is set.

0 = process complete input frame as one vector, ignoring the field/element structure.

curve = <char> [Default: ‘A’]

One character, which specifies the type of the curve to use (supported: A ; soon supported: B,C).

usePower = <numeric> [Default: 1]

1 = square the input magnitudes before multiplying with the db(X) weighting function (the output will then be a dB(X) weighted *power* spectrum).

cDeltaRegression

Description: This component computes delta regression coefficients from a data contour (x^t) using the regression equation 3.1, where W specifies half the size of the window to be used for computation of the regression coefficients. The default is $W = 2$. In order to provide HTK compatibility, equation 3.1 was implemented as described in [?].

$$d^t = \frac{\sum_{i=1}^W i \cdot (x^{t+i} - x^{t-i})}{2 \sum_{i=1}^W i^2} \quad (3.1)$$

In order to compute regression coefficients of an order higher than one, you must instantiate multiple cDeltaRegression components and connect them in a chain, where one reads from the output of the previous one.

Class hierarchy: cSmileComponent → cDataProcessor → cWindowProcessor → cDeltaRegression

Configuration options:

reader = <object of type ‘cDataReader’> The configuration of the cDataReader sub-component, which handles the dataMemory interface for data input. See the documentation of ‘cDataReader’ for more information (section 3.3.2).

writer = <object of type ‘cDataWriter’> The configuration of the cDataWriter sub-component, which handles the dataMemory interface for data output. See the documentation of ‘cDataWriter’ for more information (section 3.3.2).

bufferize = <numeric> [Default: 0]
The buffer size for the output level, in frames (default [0] = same as input level), this option overwrites ‘bufferize_sec’.

bufferize_sec = <numeric> [Default: 0]
The buffer size for the output level, in seconds (default [0] = same as input level).

blocksize = <numeric> [Default: 0]
The size of data blocks to process, in frames (this sets both ‘blocksizeR’ and ‘blocksizeW’, and overwrites ‘blocksize_sec’).

blocksizeR = <numeric> [Default: 0]
The size of data blocks to read, in frames (overwrites ‘blocksize’).

blocksizeW = <numeric> [Default: 0]
The size of data blocks to write, in frames (overwrites ‘blocksize’).

blocksize_sec = <numeric> [Default: 0]
The size of data blocks to process, in seconds (this sets both ‘blocksizeR_sec’ and ‘blocksizeW_sec’).

blocksizeR_sec = <numeric> [Default: 0]
The size of data blocks to read, in seconds (overwrites ‘blocksize_sec’).

blocksizeW_sec = <numeric> [Default: 0]
The size of data blocks to write in seconds (overwrites ‘blocksize_sec’).

nameAppend = <string> [Default: ‘de’]
A string suffix to append to the input field names.

copyInputName = <numeric> [Default: 1]
1 = copy the input name (and optionally append a suffix, see the ‘nameAppend’ option)
0 = discard the input name and use only the ‘nameAppend’ string as new name.

noPostEOIprocessing = <numeric> [Default: 0]
1 = do not process incomplete windows at the end of the input.

deltawin = <numeric> [Default: 2]
(= W): specifies the size of half of the delta regression window (If set to 0, a simple difference $d[n] = x[n] - x[n - 1]$ will be computed).

cEnergy

Description: This component computes logarithmic (log) and root-mean-square (rms) signal energy from PCM frames (size N , values x_n , $n = 1..N$).

Class hierarchy: cSmileComponent \rightarrow cDataProcessor \rightarrow cVectorProcessor \rightarrow cEnergy

Configuration options:

reader = <object of type 'cDataReader'> The configuration of the cDataReader sub-component, which handles the dataMemory interface for data input. See the documentation of 'cDataReader' for more information (section 3.3.2).

writer = <object of type 'cDataWriter'> The configuration of the cDataWriter sub-component, which handles the dataMemory interface for data output. See the documentation of 'cDataWriter' for more information (section 3.3.2).

bufferSize = <numeric> [Default: 0]
The buffer size for the output level, in frames (default [0] = same as input level), this option overwrites 'bufferSize_sec'.

bufferSize_sec = <numeric> [Default: 0]
The buffer size for the output level, in seconds (default [0] = same as input level).

blockSize = <numeric> [Default: 0]
The size of data blocks to process, in frames (this sets both 'blockSizeR' and 'blockSizeW', and overwrites 'blockSize_sec').

blockSizeR = <numeric> [Default: 0]
The size of data blocks to read, in frames (overwrites 'blockSize').

blockSizeW = <numeric> [Default: 0]
The size of data blocks to write, in frames (overwrites 'blockSize').

blockSize_sec = <numeric> [Default: 0]
The size of data blocks to process, in seconds (this sets both 'blockSizeR_sec' and 'blockSizeW_sec').

blockSizeR_sec = <numeric> [Default: 0]
The size of data blocks to read, in seconds (overwrites 'blockSize_sec').

blockSizeW_sec = <numeric> [Default: 0]
The size of data blocks to write in seconds (overwrites 'blockSize_sec').

nameAppend = <string> [Default: 'energy']
A string suffix to append to the input field names.

copyInputName = <numeric> [Default: 1]
1 = copy the input name (and optionally append a suffix, see the 'nameAppend' option)
0 = discard the input name and use only the 'nameAppend' string as new name.

processArrayFields = <numeric> [Default: 1]
1 = process each array field as one vector individually (and produce one output for each input array field). Only array fields (i.e. fields with more than one element) are processed

if this is set.

0 = process complete input frame as one vector, ignoring the field/element structure.

htkcompatible = <numeric> [Default: 0]

If set to 1, enable HTK compatible log-energy output (this will output log-energy ONLY! no rms energy, disregarding the ‘rms’ option).

rms = <numeric> [Default: 1]

1/0 = on/off: output of root-mean-square (RMS) energy E_r (frame size is N):

$$E_r = \sqrt{\left(\sum_{n=0}^N x_n^2\right) / N}.$$

log = <numeric> [Default: 1]

1/0 = on/off output logarithmic energy (log-energy), E_l . $E_l = \log \left[\left(\sum_{n=0}^N x_n^2 \right) / N \right]$. The argument of the log function is floored to 8.674676e-019. The output has the unit of ‘neper’ (which is similar to decibel, with the difference that the natural logarithm (base e) is used instead of the base-10 logarithm).

cFFTmagphase

Description: This component computes the magnitudes and the phases for each array in the input level (it thereby assumes that the arrays contain complex numbers with real and imaginary parts alternating, as computed by the cTransformFFT (section 3.3.6) component).

Class hierarchy: cSmileComponent → cDataProcessor → cVectorProcessor → cFFTmagphase

Configuration options:

reader = <object of type ‘cDataReader’> The configuration of the cDataReader sub-component, which handles the dataMemory interface for data input. See the documentation of ‘cDataReader’ for more information (section 3.3.2).

writer = <object of type ‘cDataWriter’> The configuration of the cDataWriter sub-component, which handles the dataMemory interface for data output. See the documentation of ‘cDataWriter’ for more information (section 3.3.2).

bufferize = <numeric> [Default: 0]

The buffer size for the output level, in frames (default [0] = same as input level), this option overwrites ‘bufferize_sec’.

bufferize_sec = <numeric> [Default: 0]

The buffer size for the output level, in seconds (default [0] = same as input level).

blocksize = <numeric> [Default: 0]

The size of data blocks to process, in frames (this sets both ‘blocksizeR’ and ‘blocksizeW’, and overwrites ‘blocksize_sec’).

blocksizeR = <numeric> [Default: 0]

The size of data blocks to read, in frames (overwrites ‘blocksize’).

blocksizeW = <numeric> [Default: 0]

The size of data blocks to write, in frames (overwrites ‘blocksize’).

blocksize_sec = <numeric> [Default: 0]

The size of data blocks to process, in seconds (this sets both ‘blocksizeR_sec’ and ‘blocksizeW_sec’).

blocksizeR_sec = <numeric> [Default: 0]

The size of data blocks to read, in seconds (overwrites ‘blocksize_sec’).

blocksizeW_sec = <numeric> [Default: 0]

The size of data blocks to write in seconds (overwrites ‘blocksize_sec’).

nameAppend = <string> [Default: ‘(null)’]

A string suffix to append to the input field names (default: empty).

copyInputName = <numeric> [Default: 1]

1 = copy the input name (and optionally append a suffix, see the ‘nameAppend’ option)

0 = discard the input name and use only the ‘nameAppend’ string as new name.

processArrayFields = <numeric> [Default: 1]

1 = process each array field as one vector individually (and produce one output for each input array field). Only array fields (i.e. fields with more than one element) are processed if this is set.

0 = process complete input frame as one vector, ignoring the field/element structure.

inverse = <numeric> [Default: 0]

If set to 1, converts magnitude and phase input data to complex frequency domain data.

magnitude = <numeric> [Default: 1]

1/0 = compute magnitude yes/no (or use magnitude as input to inverse transformation, must be enabled for inverse).

phase = <numeric> [Default: 0]

1/0 = compute phase yes/no (or use phase as input to inverse transformation, must be enabled for inverse).

cFormantLpc

Description: This component computes formant frequencies and bandwidths by solving for the roots of the LPC coefficient polynomial. The component requires lpc coefficients as input, which are generated by the cLpc component (section 3.3.6). The formant trajectories can be smoothed by the cFormantSmoother (section 3.3.6) component.

Class hierarchy: cSmileComponent → cDataProcessor → cVectorProcessor → cFormantLpc

Configuration options:

reader = <object of type ‘cDataReader’> The configuration of the cDataReader sub-component, which handles the dataMemory interface for data input. See the documentation of ‘cDataReader’ for more information (section 3.3.2).

writer = <object of type 'cDataWriter'> The configuration of the cDataWriter sub-component, which handles the dataMemory interface for data output. See the documentation of 'cDataWriter' for more information (section 3.3.2).

bufferize = <numeric> [Default: 0]
The buffer size for the output level, in frames (default [0] = same as input level), this option overwrites 'bufferize_sec'.

bufferize_sec = <numeric> [Default: 0]
The buffer size for the output level, in seconds (default [0] = same as input level).

blocksize = <numeric> [Default: 0]
The size of data blocks to process, in frames (this sets both 'blocksizeR' and 'blocksizeW', and overwrites 'blocksize_sec').

blocksizeR = <numeric> [Default: 0]
The size of data blocks to read, in frames (overwrites 'blocksize').

blocksizeW = <numeric> [Default: 0]
The size of data blocks to write, in frames (overwrites 'blocksize').

blocksize_sec = <numeric> [Default: 0]
The size of data blocks to process, in seconds (this sets both 'blocksizeR_sec' and 'blocksizeW_sec').

blocksizeR_sec = <numeric> [Default: 0]
The size of data blocks to read, in seconds (overwrites 'blocksize_sec').

blocksizeW_sec = <numeric> [Default: 0]
The size of data blocks to write in seconds (overwrites 'blocksize_sec').

nameAppend = <string> [Default: '(null)']
A string suffix to append to the input field names (default: empty).

copyInputName = <numeric> [Default: 1]
1 = copy the input name (and optionally append a suffix, see the 'nameAppend' option)
0 = discard the input name and use only the 'nameAppend' string as new name.

processArrayFields = <numeric> [Default: 0]
1 = process each array field as one vector individually (and produce one output for each input array field). Only array fields (i.e. fields with more than one element) are processed if this is set.
0 = process complete input frame as one vector, ignoring the field/element structure.

nFormants = <numeric> [Default: -1]
The maximum number of formants to detect (set to < 0 to automatically use the maximum number of possible formants (nLpcCoeff - 1)).

saveFormants = <numeric> [Default: 1]
If set to 1, output formant frequencies [field name: formantFreqLpc].

saveIntensity = <numeric> [Default: 0]
If set to 1, output formant frame intensity [field name: formantFrameIntensity].

saveNumberOfValidFormants = <numeric> [Default: 0]

If set to 1, output the number of valid formants [field name: nFormants].

saveBandwidths = <numeric> [Default: 0]

If set to 1, output formant bandwidths [field name: formantBandwidthLpc].

minF = <numeric> [Default: 50]

The minimum frequency of the formant frequency search range (in Hz).

maxF = <numeric> [Default: 5500]

The maximum detectable formant frequency (in Hz).

useLpSpec = <numeric> [Default: 0]

Experimental option: If set to 1, computes the formants from peaks found in the ‘lpSpectrum’ field instead of root solving the lpc coefficient polynomial.

medianFilter = <numeric> [Default: 0]

1 = enable formant post processing by a median filter of length ‘medianFilter’ (recommended: 5) (will be rounded up to the next odd number); 0 to disable median filter.

octaveCorrection = <numeric> [Default: 0]

Experimental option:

1 = prevent formant octave jumps (esp. when ‘medianFilter’ is enabled) by employing simple ‘octave’ correction.

0 = no correction.

cFormantSmoother

Description: This component performs temporal formant smoothing.

Input: candidates produced by a formantXXX component (e.g. formantLpc, section 3.3.6) and(!) – appended – an F0final or voicing field (i.e. a single element field, that has a value of 0 for unvoiced frames and a non-zero value for voiced frames).

Output: Smoothed formant frequency contours.

Class hierarchy: cSmileComponent → cDataProcessor → cVectorProcessor → cFormantSmoother

Configuration options:

reader = <object of type ‘cDataReader’> The configuration of the cDataReader sub-component, which handles the dataMemory interface for data input. See the documentation of ‘cDataReader’ for more information (section 3.3.2).

writer = <object of type ‘cDataWriter’> The configuration of the cDataWriter sub-component, which handles the dataMemory interface for data output. See the documentation of ‘cDataWriter’ for more information (section 3.3.2).

bufferSize = <numeric> [Default: 0]

The buffer size for the output level, in frames (default [0] = same as input level), this option overwrites ‘bufferSize_sec’.

bufferSize_sec = <numeric> [Default: 0]

The buffer size for the output level, in seconds (default [0] = same as input level).

- blocksize** = **<numeric>** [Default: 0]
 The size of data blocks to process, in frames (this sets both ‘blocksizeR’ and ‘blocksizeW’, and overwrites ‘blocksize_sec’).
- blocksizeR** = **<numeric>** [Default: 0]
 The size of data blocks to read, in frames (overwrites ‘blocksize’).
- blocksizeW** = **<numeric>** [Default: 0]
 The size of data blocks to write, in frames (overwrites ‘blocksize’).
- blocksize_sec** = **<numeric>** [Default: 0]
 The size of data blocks to process, in seconds (this sets both ‘blocksizeR_sec’ and ‘blocksizeW_sec’).
- blocksizeR_sec** = **<numeric>** [Default: 0]
 The size of data blocks to read, in seconds (overwrites ‘blocksize_sec’).
- blocksizeW_sec** = **<numeric>** [Default: 0]
 The size of data blocks to write in seconds (overwrites ‘blocksize_sec’).
- nameAppend** = **<string>** [Default: ‘(null)’]
 A string suffix to append to the input field names (default: empty).
- copyInputName** = **<numeric>** [Default: 1]
 1 = copy the input name (and optionally append a suffix, see the ‘nameAppend’ option)
 0 = discard the input name and use only the ‘nameAppend’ string as new name.
- processArrayFields** = **<numeric>** [Default: 0]
 1 = process each array field as one vector individually (and produce one output for each input array field). Only array fields (i.e. fields with more than one element) are processed if this is set.
 0 = process complete input frame as one vector, ignoring the field/element structure.
- medianFilter0** = **<numeric>** [Default: 0]
 If > 0, applies median filtering of candidates as the *first* processing step; the filter length is the value of ‘medianFilter0’ if > 0.
- postSmoothing** = **<numeric>** [Default: 0]
 If > 0, applies post processing (median and spike remover) over ‘postSmoothing’ frames (0 = no smoothing or use default determined by ‘postSmoothingMethod’).
- postSmoothingMethod** = **<string>** [Default: ‘simple’]
 The post processing method to use. One of the following:
- none** disable post smoothing
 - simple** simple post smoothing using only one frame delay (will smooth out one frame octave spikes)
 - median** apply a median filter to the output values (length = value of ‘postProcessing’ option)

F0field = <string> [Default: 'F0final']

The input field containing either **F0final** or **voicingFinalClipped** (i.e. a single element field, that has a value of 0 for unvoiced frames and a non-zero value for voiced frames). The name you give here is a partial name, i.e. the actual field names will be matched against *'F0field'*). *Note:* do not use the ***Env** (envelope) fields here, they are non-zero for *unvoiced* frames!

formantBandwidthField = <string> [Default: 'formantBand']

The input field containing formant bandwidths (the name you give here is a partial name, i.e. the actual field names will be matched against *'formantBandwidthField'*).

formantFreqField = <string> [Default: 'formantFreq']

The input field containing formant frequencies (the name you give here is a partial name, i.e. the actual field names will be matched against *'formantFreqField'*).

formantFrameIntensField = <string> [Default: 'formantFrameIntens']

The input field containing formant frame intensity (the name you give here is a partial name, i.e. the actual field names will be matched against *'formantFrameIntensField'*).

intensity = <numeric> [Default: 0]

If set to 1, output formant intensity.

nFormants = <numeric> [Default: 5]

This sets the maximum number of smoothed formants to output (set to 0 to disable the output of formants and bandwidths).

formants = <numeric> [Default: 1]

If set to 1, output formant frequencies (also see 'nFormants' option).

bandwidths = <numeric> [Default: 0]

If set to 1, output formant bandwidths (also see 'nFormants' option).

saveEnvs = <numeric> [Default: 0]

If set to 1, output formant frequency and bandwidth envelopes instead(!) of the actual data (i.e. the last value of a voiced frame is used for the following unvoiced frames).

no0f0 = <numeric> [Default: 0]

'no zero F_0 ': if set to 1, output data only when $F_0 > 0$, i.e. a voiced frame is detected. This may cause problem with some functionals and framer components, which don't support this variable length data yet.

cFramer

Description: This component creates frames from single dimensional input stream. It is possible to specify the frame step and frame size independently, thus allowing for overlapping frames or non continuous frames. Creating a single frame from the full-input (at the end of the input, when doing off-line processing), is also supported. Creating frames based on smile messages from other components is possible. Creating frames based on an off-line file list (e.g. a text file) is partially in the code but not yet finished.

Class hierarchy: cSmileComponent → cDataProcessor → cWinToVecProcessor → cFramer

Configuration options:

reader = <object of type ‘cDataReader’> The configuration of the cDataReader sub-component, which handles the dataMemory interface for data input. See the documentation of ‘cDataReader’ for more information (section 3.3.2).

writer = <object of type ‘cDataWriter’> The configuration of the cDataWriter sub-component, which handles the dataMemory interface for data output. See the documentation of ‘cDataWriter’ for more information (section 3.3.2).

bufferSize = <numeric> [Default: 0]
The buffer size for the output level, in frames (default [0] = same as input level), this option overwrites ‘bufferSize_sec’.

bufferSize_sec = <numeric> [Default: 0]
The buffer size for the output level, in seconds (default [0] = same as input level).

nameAppend = <string> [Default: ‘(null)’]
A string suffix to append to the input field names (default: empty)

copyInputName = <numeric> [Default: 1]
1 = copy the input name (and optionally append a suffix, see ‘nameAppend’ option)
0 = discard the input name and use only the ‘nameAppend’ string as new name.

frameMode = <string> [Default: ‘fixed’]
Specifies how to create frames:

fixed fixed frame size, given via the ‘frameSize’ option

full creates one frame at the end of the input only (off-line processing)

variable via smile message from another component (such as cTurnDetector (section 3.3.6)

list frame times list in config file (‘frameList’ option), or in external text file (‘frameList-File’ option). Currently: UNIMPLEMENTED.

frameListFile = <string> [Default: ‘(null)’]
Filename of a file with a list of frame intervals to load. This should be a text file with a comma separated list of intervals on a single line: 1-10,11-20, etc., if no interval is specified, i.e. no – is found then consecutive frames with the given number being the frame length are assumed; first index is 0; use the suffix “s” after the numbers to specify intervals in seconds (e.g. 0s-2.5s); use an ‘E’ instead of a number for ‘end of sequence’).

frameList = <string> [Default: ‘(null)’]
The list of frame intervals specified directly in the configuration file. This should be a string of comma separated list of intervals on a single line: 1-10,11-20, etc., if no interval is specified, i.e. no – is found then consecutive frames with the given number being the frame length are assumed; first index is 0; use the suffix “s” after the numbers to specify intervals in seconds (e.g. 0s-2.5s); use an ‘E’ instead of a number for ‘end of sequence’).

frameSize = <numeric> [Default: 0.025]
The frame size in seconds (0.0 = full input, same as ‘frameMode=full’).

frameStep = <numeric> [Default: 0]
The frame step (frame sampling period) in seconds (0 = set to the same value as ‘frame-Size’)

frameSizeFrames = <numeric> [Default: 0]

The frame size in input level frames (=samples for a pcm/wave input level) (overrides frameSize, if set and > 0).

frameStepFrames = <numeric> [Default: 0]

The frame step in input level frames (=samples for a pcm/wave input level) (overrides frameStep, if set and > 0).

frameCenter = <numeric> [Default: 0]

The frame center in seconds, i.e. where frames are sampled (0=left), see ‘frameCenterSpecial’ for examples on how the frame center options affect the sampling of frames.

frameCenterFrames = <numeric> [Default: 0]

The frame sampling center in input level frames (overrides ‘frameCenter’, if set), (0=left), see ‘frameCenterSpecial’ for examples on how the frame center options affect the sampling of frames.

frameCenterSpecial = <string> [Default: ‘left’]

The frame sampling center (overrides the other ‘frameCenter’ options, if set). The available special frame sampling points as strings are:

left = sample at the beginning of the frame (the first frame will be sampled from 0 to frameSize)

mid = sample in the middle of the frame middle (the first frame will be sampled from -frameSize/2 to frameSize/2; values at negative indices are padded with zeros)

right = sample at the end of the frame (the first frame will be sampled from -frameSize to 0; values at negative indices are padded with zeros, i.e. the first frame will be all 0s)

noPostEOIprocessing = <numeric> [Default: 1]

1 = do not process incomplete windows at the end of the input, i.e. all created frames have been sampled from segments that are exactly ‘frameSize’ in length. Excess data at the end of the input will be discarded. This is only relevant for off-line processing.

cFullinputMean

Description: This component performs mean normalising on a data series. A 2-pass analysis of the data is performed, which makes this component unusable for on-line analysis. In the first pass, no output is produced and the mean value (over time) is computed for each input element. In the second pass the mean vector is subtracted from all input frames, and the result is written to the output data memory level.

Attention: Due to the 2-pass processing the input level must be large enough to hold the whole data sequence.

Class hierarchy: cSmileComponent → cDataProcessor → cFullinputMean

Configuration options:

reader = <object of type ‘cDataReader’> The configuration of the cDataReader sub-component, which handles the dataMemory interface for data input. See the documentation of ‘cDataReader’ for more information (section 3.3.2).

writer = <object of type 'cDataWriter'> The configuration of the cDataWriter sub-component, which handles the dataMemory interface for data output. See the documentation of 'cDataWriter' for more information (section 3.3.2).

bufferize = <numeric> [Default: 0]
The buffer size for the output level, in frames (default [0] = same as input level), this option overwrites 'bufferize_sec'.

bufferize_sec = <numeric> [Default: 0]
The buffer size for the output level, in seconds (default [0] = same as input level).

blocksize = <numeric> [Default: 0]
The size of data blocks to process, in frames (this sets both 'blocksizeR' and 'blocksizeW', and overwrites 'blocksize_sec').

blocksizeR = <numeric> [Default: 0]
The size of data blocks to read, in frames (overwrites 'blocksize').

blocksizeW = <numeric> [Default: 0]
The size of data blocks to write, in frames (overwrites 'blocksize').

blocksize_sec = <numeric> [Default: 0]
The size of data blocks to process, in seconds (this sets both 'blocksizeR_sec' and 'blocksizeW_sec').

blocksizeR_sec = <numeric> [Default: 0]
The size of data blocks to read, in seconds (overwrites 'blocksize_sec').

blocksizeW_sec = <numeric> [Default: 0]
The size of data blocks to write in seconds (overwrites 'blocksize_sec').

nameAppend = <string> [Default: '(null)']
A string suffix to append to the input field names (default: empty).

copyInputName = <numeric> [Default: 1]
1 = copy the input name (and optionally append a suffix, see the 'nameAppend' option)
0 = discard the input name and use only the 'nameAppend' string as new name.

cIntensity

Description: This component computes simplified frame intensity (narrow band approximation). It expects *unwindowed* raw PCM frames as input. A Hamming window is internally applied and the resulting signal is squared before applying loudness compression, etc.

Class hierarchy: cSmileComponent → cDataProcessor → cVectorProcessor → cIntensity

Configuration options:

reader = <object of type 'cDataReader'> The configuration of the cDataReader sub-component, which handles the dataMemory interface for data input. See the documentation of 'cDataReader' for more information (section 3.3.2).

writer = <object of type 'cDataWriter'> The configuration of the cDataWriter sub-component, which handles the dataMemory interface for data output. See the documentation of 'cDataWriter' for more information (section 3.3.2).

bufferSize = <numeric> [Default: 0]

The buffer size for the output level, in frames (default [0] = same as input level), this option overwrites ‘bufferSize_sec’.

bufferSize_sec = <numeric> [Default: 0]

The buffer size for the output level, in seconds (default [0] = same as input level).

blockSize = <numeric> [Default: 0]

The size of data blocks to process, in frames (this sets both ‘blockSizeR’ and ‘blockSizeW’, and overwrites ‘blockSize_sec’).

blockSizeR = <numeric> [Default: 0]

The size of data blocks to read, in frames (overwrites ‘blockSize’).

blockSizeW = <numeric> [Default: 0]

The size of data blocks to write, in frames (overwrites ‘blockSize’).

blockSize_sec = <numeric> [Default: 0]

The size of data blocks to process, in seconds (this sets both ‘blockSizeR_sec’ and ‘blockSizeW_sec’).

blockSizeR_sec = <numeric> [Default: 0]

The size of data blocks to read, in seconds (overwrites ‘blockSize_sec’).

blockSizeW_sec = <numeric> [Default: 0]

The size of data blocks to write in seconds (overwrites ‘blockSize_sec’).

nameAppend = <string> [Default: ‘(null)’]

A string suffix to append to the input field names (default: empty).

copyInputName = <numeric> [Default: 1]

1 = copy the input name (and optionally append a suffix, see the ‘nameAppend’ option)
0 = discard the input name and use only the ‘nameAppend’ string as new name.

processArrayFields = <numeric> [Default: 1]

1 = process each array field as one vector individually (and produce one output for each input array field). Only array fields (i.e. fields with more than one element) are processed if this is set.

0 = process complete input frame as one vector, ignoring the field/element structure.

intensity = <numeric> [Default: 1]

1 = enable the output of intensity I (mean of squared input values multiplied by a Hamming window)

loudness = <numeric> [Default: 0]

1 = enable the output of loudness l : $l = (\frac{I}{I_0})^{0.3}$ where $I_0 = 0.000001$ (I_0 cannot be changed at the moment, it is hard-coded).

cLpc

Description: This component computes linear predictive coding (LPC) coefficients from PCM frames. Burg’s algorithm and the standard ACF/Durbin based method are implemented for lpc coefficient computation. LPC filter coefficients, reflection coefficients, residual signal, and LP

spectrum are supported.

Class hierarchy: cSmileComponent → cDataProcessor → cVectorProcessor → cLpc

Configuration options:

reader = <object of type ‘cDataReader’> The configuration of the cDataReader sub-component, which handles the dataMemory interface for data input. See the documentation of ‘cDataReader’ for more information (section 3.3.2).

writer = <object of type ‘cDataWriter’> The configuration of the cDataWriter sub-component, which handles the dataMemory interface for data output. See the documentation of ‘cDataWriter’ for more information (section 3.3.2).

bufferSize = <numeric> [Default: 0]
The buffer size for the output level, in frames (default [0] = same as input level), this option overwrites ‘bufferSize_sec’.

bufferSize_sec = <numeric> [Default: 0]
The buffer size for the output level, in seconds (default [0] = same as input level).

blockSize = <numeric> [Default: 0]
The size of data blocks to process, in frames (this sets both ‘blockSizeR’ and ‘blockSizeW’, and overwrites ‘blockSize_sec’).

blockSizeR = <numeric> [Default: 0]
The size of data blocks to read, in frames (overwrites ‘blockSize’).

blockSizeW = <numeric> [Default: 0]
The size of data blocks to write, in frames (overwrites ‘blockSize’).

blockSize_sec = <numeric> [Default: 0]
The size of data blocks to process, in seconds (this sets both ‘blockSizeR_sec’ and ‘blockSizeW_sec’).

blockSizeR_sec = <numeric> [Default: 0]
The size of data blocks to read, in seconds (overwrites ‘blockSize_sec’).

blockSizeW_sec = <numeric> [Default: 0]
The size of data blocks to write in seconds (overwrites ‘blockSize_sec’).

nameAppend = <string> [Default: ‘(null)’]
A string suffix to append to the input field names (default: empty).

copyInputName = <numeric> [Default: 1]
1 = copy the input name (and optionally append a suffix, see the ‘nameAppend’ option)
0 = discard the input name and use only the ‘nameAppend’ string as new name.

processArrayFields = <numeric> [Default: 1]
1 = process each array field as one vector individually (and produce one output for each input array field). Only array fields (i.e. fields with more than one element) are processed if this is set.
0 = process complete input frame as one vector, ignoring the field/element structure.

method = <string> [Default: 'acf']

This option sets the lpc method to use. Choose between: 'acf' acf (autocorrelation) method with Levinson-Durbin algorithm , 'burg' Burg method (N. Anderson (1978))

p = <numeric> [Default: 8]

Predictor order (= number of lpc coefficients).

saveLPCoeff = <numeric> [Default: 1]

1 = save LP coefficients to output.

lpGain = <numeric> [Default: 0]

1 = save lpc gain (error) in output vector.

saveRefCoeff = <numeric> [Default: 0]

1 = save reflection coefficients to output.

residual = <numeric> [Default: 0]

1 = compute lpc residual signal and store in output frame.

forwardFilter = <numeric> [Default: 0]

1 = apply forward instead of inverse filter when computing residual.

lpSpectrum = <numeric> [Default: 0]

1 = compute lp spectrum using 'lpSpecDeltaF' as frequency resolution or 'lpSpecBins' bins.

lpSpecDeltaF = <numeric> [Default: 10]

Frequency resolution of lp spectrum (only applicable if 'lpSpectrum=1').

lpSpecBins = <numeric> [Default: 100]

Number of bins to compute lp spectrum for (overrides 'lpSpecDeltaF', only applicable if 'lpSpectrum=1').

cLsp

Description: This component computes LSP (line spectral pair frequencies, also known as LSF) from LPC coefficients (cLpc, section 3.3.6) by partial factorisation of the LPC polynomial.

Class hierarchy: cSmileComponent → cDataProcessor → cVectorProcessor → cLsp

Configuration options:

reader = <object of type 'cDataReader'> The configuration of the cDataReader sub-component, which handles the dataMemory interface for data input. See the documentation of 'cDataReader' for more information (section 3.3.2).

writer = <object of type 'cDataWriter'> The configuration of the cDataWriter sub-component, which handles the dataMemory interface for data output. See the documentation of 'cDataWriter' for more information (section 3.3.2).

bufferSize = <numeric> [Default: 0]

The buffer size for the output level, in frames (default [0] = same as input level), this option overwrites 'bufferSize_sec'.

buffer_size_sec = <numeric> [Default: 0]

The buffer size for the output level, in seconds (default [0] = same as input level).

blocksize = <numeric> [Default: 0]

The size of data blocks to process, in frames (this sets both ‘blocksizeR’ and ‘blocksizeW’, and overwrites ‘blocksize_sec’).

blocksizeR = <numeric> [Default: 0]

The size of data blocks to read, in frames (overwrites ‘blocksize’).

blocksizeW = <numeric> [Default: 0]

The size of data blocks to write, in frames (overwrites ‘blocksize’).

blocksize_sec = <numeric> [Default: 0]

The size of data blocks to process, in seconds (this sets both ‘blocksizeR_sec’ and ‘blocksizeW_sec’).

blocksizeR_sec = <numeric> [Default: 0]

The size of data blocks to read, in seconds (overwrites ‘blocksize_sec’).

blocksizeW_sec = <numeric> [Default: 0]

The size of data blocks to write in seconds (overwrites ‘blocksize_sec’).

nameAppend = <string> [Default: ‘(null)’]

A string suffix to append to the input field names (default: empty).

copyInputName = <numeric> [Default: 1]

1 = copy the input name (and optionally append a suffix, see the ‘nameAppend’ option)

0 = discard the input name and use only the ‘nameAppend’ string as new name.

processArrayFields = <numeric> [Default: 0]

1 = process each array field as one vector individually (and produce one output for each input array field). Only array fields (i.e. fields with more than one element) are processed if this is set.

0 = process complete input frame as one vector, ignoring the field/element structure.

cMelspec

Description: This component computes an N-band Mel/Bark/Semitone-frequency spectrum (critical band spectrum) by applying overlapping triangular filters equidistant on the Mel/Bark/Semitone-frequency scale to an FFT magnitude spectrum (generated by the cFFTmagphase component, section 3.3.6). You can use this critical band spectrum to generate MFCC with the cMfcc component (section 3.3.6), or PLP and auditory spectra with the cPlp component (section 3.3.6).

Class hierarchy: cSmileComponent → cDataProcessor → cVectorProcessor → cMelspec

Configuration options:

reader = <object of type ‘cDataReader’> The configuration of the cDataReader sub-component, which handles the dataMemory interface for data input. See the documentation of ‘cDataReader’ for more information (section 3.3.2).

writer = <object of type ‘cDataWriter’> The configuration of the cDataWriter sub-component, which handles the dataMemory interface for data output. See the documentation of ‘cDataWriter’ for more information (section 3.3.2).

bufferSize = <numeric> [Default: 0]

The buffer size for the output level, in frames (default [0] = same as input level), this option overwrites ‘bufferSize_sec’.

bufferSize_sec = <numeric> [Default: 0]

The buffer size for the output level, in seconds (default [0] = same as input level).

blockSize = <numeric> [Default: 0]

The size of data blocks to process, in frames (this sets both ‘blockSizeR’ and ‘blockSizeW’, and overwrites ‘blockSize_sec’).

blockSizeR = <numeric> [Default: 0]

The size of data blocks to read, in frames (overwrites ‘blockSize’).

blockSizeW = <numeric> [Default: 0]

The size of data blocks to write, in frames (overwrites ‘blockSize’).

blockSize_sec = <numeric> [Default: 0]

The size of data blocks to process, in seconds (this sets both ‘blockSizeR_sec’ and ‘blockSizeW_sec’).

blockSizeR_sec = <numeric> [Default: 0]

The size of data blocks to read, in seconds (overwrites ‘blockSize_sec’).

blockSizeW_sec = <numeric> [Default: 0]

The size of data blocks to write in seconds (overwrites ‘blockSize_sec’).

nameAppend = <string> [Default: ‘(null)’]

A string suffix to append to the input field names (default: empty).

copyInputName = <numeric> [Default: 1]

1 = copy the input name (and optionally append a suffix, see the ‘nameAppend’ option)
0 = discard the input name and use only the ‘nameAppend’ string as new name.

processArrayFields = <numeric> [Default: 1]

1 = process each array field as one vector individually (and produce one output for each input array field). Only array fields (i.e. fields with more than one element) are processed if this is set.

0 = process complete input frame as one vector, ignoring the field/element structure.

nBands = <numeric> [Default: 26]

The number of Mel/Bark/Semitone band filters the filterbank from ‘lofreq’ – ‘hifreq’ contains.

lofreq = <numeric> [Default: 20]

The lower cut-off frequency of the filterbank (Hz).

hifreq = <numeric> [Default: 8000]

The upper cut-off frequency of the filterbank (Hz).

usePower = <numeric> [Default: 0]

Set this to 1, to use the power spectrum instead of magnitude spectrum, i.e. if this option is set to 1, the input values are squared by the cMelspec component.

showFbank = <numeric> [Default: 0]

If this is set to 1, the bandwidths and centre frequencies of the filters in the filterbank are printed to openSMILE log output (console and/or file).

htkcompatible = <numeric> [Default: 1]

1 = enable htk compatible output (audio sample scaling -32767..+32767 instead of openSMILE's -1.0..1.0).

inverse = <numeric> [Default: 0]

1 = compute fft magnitude spectrum from mel spectrum; Note that if this option is set, 'nBands' specifies the number of fft bands to create! [NOT YET FULLY TESTED]

specScale = <string> [Default: 'mel']

The frequency scale to design the critical band filterbank in:

mel Mel-frequency scale ($m = 1127 \ln(1 + f/700)$)

bark Bark scale approximation (Critical band rate z): $z = [26.81 / (1.0 + 1960/f)] - 0.53$

bark_schroed = Bark scale approximation according to Schroeder (1977):

$$6 \log \left(\frac{f_{Hz}}{600} + \left[\left(\frac{f_{Hz}}{600} \right)^2 + 1 \right]^{.5} \right)$$

bark_speex = Bark scale approximation as used in the Speex codec package

semi = semi-tone scale with first note (0) = 'firstNote' (default 27.5 Hz)

(semitone = $12 * \log(f_{Hz}/\text{firstNote}) / \log(2)$) [experimental]

log = logarithmic scale with base 'logScaleBase' (default = 2, octave scale).

logScaleBase = <numeric> [Default: 2]

The base for log scales (a log base of 2.0 - the default - corresponds to an octave target scale).

firstNote = <numeric> [Default: 27.5]

The first note (in Hz) for a semi-tone scale.

cMfcc

Description: This component computes Mel-frequency cepstral coefficients (MFCC) from a critical band spectrum (see 'cMelspec', section 3.3.6). An i-DCT of type-II is used from transformation from the spectral to the cepstral domain. Liftering of cepstral coefficients is supported. HTK compatible values can be computed, if the 'htkcompatible' option is set to 1 in this component and in the cMelspec component.

Class hierarchy: cSmileComponent → cDataProcessor → cVectorProcessor → cMfcc

Configuration options:

reader = <object of type 'cDataReader'> The configuration of the cDataReader sub-component, which handles the dataMemory interface for data input. See the documentation of 'cDataReader' for more information (section 3.3.2).

writer = <object of type 'cDataWriter'> The configuration of the cDataWriter sub-component, which handles the dataMemory interface for data output. See the documentation of 'cDataWriter' for more information (section 3.3.2).

bufferSize = <numeric> [Default: 0]

The buffer size for the output level, in frames (default [0] = same as input level), this option overwrites 'bufferSize_sec'.

bufferSize_sec = <numeric> [Default: 0]

The buffer size for the output level, in seconds (default [0] = same as input level).

blockSize = <numeric> [Default: 0]

The size of data blocks to process, in frames (this sets both 'blockSizeR' and 'blockSizeW', and overwrites 'blockSize_sec').

blockSizeR = <numeric> [Default: 0]

The size of data blocks to read, in frames (overwrites 'blockSize').

blockSizeW = <numeric> [Default: 0]

The size of data blocks to write, in frames (overwrites 'blockSize').

blockSize_sec = <numeric> [Default: 0]

The size of data blocks to process, in seconds (this sets both 'blockSizeR_sec' and 'blockSizeW_sec').

blockSizeR_sec = <numeric> [Default: 0]

The size of data blocks to read, in seconds (overwrites 'blockSize_sec').

blockSizeW_sec = <numeric> [Default: 0]

The size of data blocks to write in seconds (overwrites 'blockSize_sec').

nameAppend = <string> [Default: '(null)']

A string suffix to append to the input field names (default: empty).

copyInputName = <numeric> [Default: 1]

1 = copy the input name (and optionally append a suffix, see the 'nameAppend' option)
0 = discard the input name and use only the 'nameAppend' string as new name.

processArrayFields = <numeric> [Default: 1]

1 = process each array field as one vector individually (and produce one output for each input array field). Only array fields (i.e. fields with more than one element) are processed if this is set.

0 = process complete input frame as one vector, ignoring the field/element structure.

firstMfcc = <numeric> [Default: 1]

The first MFCC to compute.

lastMfcc = <numeric> [Default: 12]

The last MFCC to compute.

nMfcc = <numeric> [Default: 12]

Use this option to specify the number of MFCC, instead of specifying 'lastMfcc'.

melfloor = <numeric> [Default: 1e-08]

The minimum value allowed for Mel-spectra when taking the log spectrum (this parameter will be forced to 1.0 when htkcompatible=1)

cepLifter = <numeric> [Default: 22]

Parameter for cepstral ‘liftering’, set this to 0.0 to disable cepstral liftering.

htkcompatible = <numeric> [Default: 1]

1 = append the 0-th coefficient at the end instead of placing it as the first element of the output vector. This produces HTK compatible Mfcc output.

cMZcr

Description: This component computes these time signal properties: zero-crossing rate, mean-crossing rate, dc offset, max/min value, and absolute maximum value of a PCM frame.

Class hierarchy: cSmileComponent → cDataProcessor → cVectorProcessor → cMZcr

Configuration options:

reader = <object of type ‘cDataReader’> The configuration of the cDataReader sub-component, which handles the dataMemory interface for data input. See the documentation of ‘cDataReader’ for more information (section 3.3.2).

writer = <object of type ‘cDataWriter’> The configuration of the cDataWriter sub-component, which handles the dataMemory interface for data output. See the documentation of ‘cDataWriter’ for more information (section 3.3.2).

bufferize = <numeric> [Default: 0]

The buffer size for the output level, in frames (default [0] = same as input level), this option overwrites ‘bufferize_sec’.

bufferize_sec = <numeric> [Default: 0]

The buffer size for the output level, in seconds (default [0] = same as input level).

blocksize = <numeric> [Default: 0]

The size of data blocks to process, in frames (this sets both ‘blocksizeR’ and ‘blocksizeW’, and overwrites ‘blocksize_sec’).

blocksizeR = <numeric> [Default: 0]

The size of data blocks to read, in frames (overwrites ‘blocksize’).

blocksizeW = <numeric> [Default: 0]

The size of data blocks to write, in frames (overwrites ‘blocksize’).

blocksize_sec = <numeric> [Default: 0]

The size of data blocks to process, in seconds (this sets both ‘blocksizeR_sec’ and ‘blocksizeW_sec’).

blocksizeR_sec = <numeric> [Default: 0]

The size of data blocks to read, in seconds (overwrites ‘blocksize_sec’).

blocksizeW_sec = <numeric> [Default: 0]

The size of data blocks to write in seconds (overwrites ‘blocksize_sec’).

nameAppend = <string> [Default: ‘(null)’]

A string suffix to append to the input field names (default: empty).

copyInputName = <numeric> [Default: 1]

1 = copy the input name (and optionally append a suffix, see the ‘nameAppend’ option)

0 = discard the input name and use only the ‘nameAppend’ string as new name.

processArrayFields = <numeric> [Default: 1]

1 = process each array field as one vector individually (and produce one output for each input array field). Only array fields (i.e. fields with more than one element) are processed if this is set.

0 = process complete input frame as one vector, ignoring the field/element structure.

zcr = <numeric> [Default: 1]

(1/0=yes/no) compute zero-crossing rate (zcr).

mcr = <numeric> [Default: 1]

(1/0=yes/no) compute mean-crossing rate (mcr) (i.e. the rate at which a signal crosses its mean, for signals with mean 0 this is identical to the zero-crossing rate).

amax = <numeric> [Default: 1]

(1/0=yes/no) enable output of the maximum *absolute* sample value.

maxmin = <numeric> [Default: 1]

(1/0=yes/no) enable output of the maximum and minimum sample values.

dc = <numeric> [Default: 0]

(1/0=yes/no) compute the DC-offset (= the arithmetic mean of input values).

cPitchACF

Description: This component computes the fundamental frequency and the probability of voicing via an ACF/Cepstrum based method. The input must be an ACF field and a Cepstrum field, concatenated exactly in this order. Both fields can be generated by a cAcf component (section 3.3.6), each. *Note:* this component does not conform to the new pitch component standard, as defined by the cPitchBase class.

Class hierarchy: cSmileComponent → cDataProcessor → cVectorProcessor → cPitchACF

Configuration options:

reader = <object of type ‘cDataReader’> The configuration of the cDataReader sub-component, which handles the dataMemory interface for data input. See the documentation of ‘cDataReader’ for more information (section 3.3.2).

writer = <object of type ‘cDataWriter’> The configuration of the cDataWriter sub-component, which handles the dataMemory interface for data output. See the documentation of ‘cDataWriter’ for more information (section 3.3.2).

bufferSize = <numeric> [Default: 0]

The buffer size for the output level, in frames (default [0] = same as input level), this option overwrites ‘bufferSize_sec’.

bufferSize_sec = <numeric> [Default: 0]

The buffer size for the output level, in seconds (default [0] = same as input level).

blocksize = <numeric> [Default: 0]

The size of data blocks to process, in frames (this sets both ‘blocksizeR’ and ‘blocksizeW’, and overwrites ‘blocksize_sec’).

blocksizeR = <numeric> [Default: 0]

The size of data blocks to read, in frames (overwrites ‘blocksize’).

blocksizeW = <numeric> [Default: 0]

The size of data blocks to write, in frames (overwrites ‘blocksize’).

blocksize_sec = <numeric> [Default: 0]

The size of data blocks to process, in seconds (this sets both ‘blocksizeR_sec’ and ‘blocksizeW_sec’).

blocksizeR_sec = <numeric> [Default: 0]

The size of data blocks to read, in seconds (overwrites ‘blocksize_sec’).

blocksizeW_sec = <numeric> [Default: 0]

The size of data blocks to write in seconds (overwrites ‘blocksize_sec’).

nameAppend = <string> [Default: ‘(null)’]

A string suffix to append to the input field names (default: empty).

copyInputName = <numeric> [Default: 1]

1 = copy the input name (and optionally append a suffix, see the ‘nameAppend’ option)

0 = discard the input name and use only the ‘nameAppend’ string as new name.

processArrayFields = <numeric> [Default: 0]

1 = process each array field as one vector individually (and produce one output for each input array field). Only array fields (i.e. fields with more than one element) are processed if this is set.

0 = process complete input frame as one vector, ignoring the field/element structure.

maxPitch = <numeric> [Default: 500]

Maximum detectable fundamental frequency in Hz.

voiceProb = <numeric> [Default: 1]

1/0 = on/off: output of voicing probability.

voiceQual = <numeric> [Default: 0]

1/0 = on/off: output of fundamental frequency ‘quality’ (= ZCR of ACF).

HNR = <numeric> [Default: 1]

1/0 = on/off; output of log harmonics-to-noise ratio (HNR) computed from the ACF:

$$HNR = 10 \cdot \log \frac{ACF(T_0)}{ACF(0) - ACF(T_0)}$$

F0 = <numeric> [Default: 1]

1/0 = on/off: output of F_0 (fundamental frequency, pitch) (*Note: the F_0 output is 0 in unvoiced segments*)

F0raw = <numeric> [Default: 0]

1/0 = on/off: output of raw F_0 candidate without thresholding (i.e. forcing to 0) in unvoiced segments.

F0env = <numeric> [Default: 0]

1/0 = on/off: output of F_0 envelope (exponential decay smoothing) (*Note*: this differs from the envelope computed by the cPitchBase descendant components, such as cPitchShs!)

voicingCutoff = <numeric> [Default: 0.55]

This sets the voicing probability threshold for pitch detection [0.0 - 1.0]. Frames with voicing probability values above this threshold will be considered as voiced.

cPitchDirection

Description: This component reads pitch data, detects pseudo syllables, and computes pitch direction estimates per (pseudo-)syllable. Thereby the classes *falling*, *flat*, and *rising* are distinguished. Required input fields: F0, F0env, and ‘loudness’ or ‘RMSenergy’. Thus, you must concatenate data from a cPitchXXX component (cPitchSHS, cPitchACF, etc.) and a cEnergy or cIntensity component.

Class hierarchy: cSmileComponent → cDataProcessor → cPitchDirection

Configuration options:

reader = <object of type ‘cDataReader’> The configuration of the cDataReader sub-component, which handles the dataMemory interface for data input. See the documentation of ‘cDataReader’ for more information (section 3.3.2).

writer = <object of type ‘cDataWriter’> The configuration of the cDataWriter sub-component, which handles the dataMemory interface for data output. See the documentation of ‘cDataWriter’ for more information (section 3.3.2).

bufferize = <numeric> [Default: 0]

The buffer size for the output level, in frames (default [0] = same as input level), this option overwrites ‘bufferize_sec’.

bufferize_sec = <numeric> [Default: 0]

The buffer size for the output level, in seconds (default [0] = same as input level).

blocksize = <numeric> [Default: 0]

The size of data blocks to process, in frames (this sets both ‘blocksizeR’ and ‘blocksizeW’, and overwrites ‘blocksize_sec’).

blocksizeR = <numeric> [Default: 0]

The size of data blocks to read, in frames (overwrites ‘blocksize’).

blocksizeW = <numeric> [Default: 0]

The size of data blocks to write, in frames (overwrites ‘blocksize’).

blocksize_sec = <numeric> [Default: 0]

The size of data blocks to process, in seconds (this sets both ‘blocksizeR_sec’ and ‘blocksizeW_sec’).

blocksizeR_sec = <numeric> [Default: 0]

The size of data blocks to read, in seconds (overwrites ‘blocksize_sec’).

blocksizeW_sec = <numeric> [Default: 0]

The size of data blocks to write in seconds (overwrites ‘blocksize_sec’).

nameAppend = <string> [Default: '(null)']

A string suffix to append to the input field names (default: empty).

copyInputName = <numeric> [Default: 1]

1 = copy the input name (and optionally append a suffix, see the 'nameAppend' option)

0 = discard the input name and use only the 'nameAppend' string as new name.

ltbs = <numeric> [Default: 0.2]

The size of the long-term average buffer in seconds.

stbs = <numeric> [Default: 0.05]

The size of the short-term average buffer in seconds.

directionMsgRecp = <string> [Default: '(null)']

Recipient component(s) for per syllable event-based pitch direction message. (rise/fall/rise-fall/fall-rise message are sent only if and as often as a such event occurs on a syllable)

speakingRateBsize = <numeric> [Default: 20]

The buffer size for computation of speaking rate.

F0direction = <numeric> [Default: 1]

1 = enable output of F0 direction as numeric value (fall: -1.0 / flat: 0.0 / rise: 1.0).

directionScore = <numeric> [Default: 1]

1 = enable output of F0 direction score (short term mean - long term mean).

speakingRate = <numeric> [Default: 0]

1 = enable output of current speaking rate in Hz (is is output for every frame, thus, a lot of redundancy here).

F0avg = <numeric> [Default: 0]

1 = enable output of long term average F0.

F0smooth = <numeric> [Default: 0]

1 = enable output of exponentially smoothed F0.

onlyTurn = <numeric> [Default: 0]

1 = send pitch direction messages (directionMsgRecp) only during speech turns (voice activity) (according to turnStart/turnEnd messages received from cTurnDetector).

turnStartMessage = <string> [Default: 'turnStart']

Use this option to define a custom message name for turn start messages, i.e. if you want to use voice activity start/end messages instead.

turnEndMessage = <string> [Default: 'turnEnd']

Use this option to define a custom message name for turn end messages, i.e. if you want to use voice activity start/end messages instead.

cPitchJitter

Description: This component computes Voice Quality parameters Jitter (pitch period deviations) and Shimmer (pitch period amplitude deviations). It requires the raw PCM frames (generated by a cFramer component) and the corresponding fundamental frequency (F_0) (cPitchACF

or `cPitchSHS`) as inputs.

Class hierarchy: `cSmileComponent` \rightarrow `cDataProcessor` \rightarrow `cPitchJitter`

Configuration options:

reader = **<object of type ‘cDataReader’>** The configuration of the `cDataReader` sub-component, which handles the `dataMemory` interface for data input. See the documentation of ‘`cDataReader`’ for more information (section 3.3.2).

writer = **<object of type ‘cDataWriter’>** The configuration of the `cDataWriter` sub-component, which handles the `dataMemory` interface for data output. See the documentation of ‘`cDataWriter`’ for more information (section 3.3.2).

bufferSize = **<numeric>** [Default: 0]
The buffer size for the output level, in frames (default [0] = same as input level), this option overwrites ‘`bufferSize_sec`’.

bufferSize_sec = **<numeric>** [Default: 0]
The buffer size for the output level, in seconds (default [0] = same as input level).

blockSize = **<numeric>** [Default: 0]
The size of data blocks to process, in frames (this sets both ‘`blockSizeR`’ and ‘`blockSizeW`’, and overwrites ‘`blockSize_sec`’).

blockSizeR = **<numeric>** [Default: 0]
The size of data blocks to read, in frames (overwrites ‘`blockSize`’).

blockSizeW = **<numeric>** [Default: 0]
The size of data blocks to write, in frames (overwrites ‘`blockSize`’).

blockSize_sec = **<numeric>** [Default: 0]
The size of data blocks to process, in seconds (this sets both ‘`blockSizeR_sec`’ and ‘`blockSizeW_sec`’).

blockSizeR_sec = **<numeric>** [Default: 0]
The size of data blocks to read, in seconds (overwrites ‘`blockSize_sec`’).

blockSizeW_sec = **<numeric>** [Default: 0]
The size of data blocks to write in seconds (overwrites ‘`blockSize_sec`’).

nameAppend = **<string>** [Default: ‘(null)’]
A string suffix to append to the input field names (default: empty).

copyInputName = **<numeric>** [Default: 1]
1 = copy the input name (and optionally append a suffix, see the ‘`nameAppend`’ option)
0 = discard the input name and use only the ‘`nameAppend`’ string as new name.

F0reader = **<object of type ‘cDataReader’>** Configuration of the `dataMemory` reader sub-component which is used to read the F_0 estimate from a pitch component output (e.g. `cPitchShs`). See the documentation of ‘`cDataReader`’ for more information (section 3.3.2).

F0field = **<string>** [Default: ‘F0final’]
The name of the field in ‘`F0reader.dmLevel`’ containing the F_0 estimate (in Hz) (usually `F0final` or `F0raw`).

searchRangeRel = <numeric> [Default: 0.25]

The relative search range for period deviations (Jitter): $\max T_0, \min T_0 = (1.0 \pm \text{searchRangeRel}) * T_0$

jitterLocal = <numeric> [Default: 0]

1 = enable computation of F0 jitter (period length variations). jitterLocal = the average absolute difference between consecutive periods, divided by the average period length of all periods in the frame.

jitterDDP = <numeric> [Default: 0]

1 = enable computation of F0 jitter (period length variations). jitterDDP = the average absolute difference between consecutive differences between consecutive periods, divided by the average period length of all periods in the frame.

jitterLocalEnv = <numeric> [Default: 0]

1 = compute envelope of jitterLocal (i.e. fill jitter values in unvoiced frames with value of last voiced segment). Use this in conjunction with statistical functionals such as means.

jitterDDPEnv = <numeric> [Default: 0]

1 = compute envelope of jitterDDP (i.e. fill jitter values in unvoiced frames with value of last voiced segment). Use this in conjunction with statistical functionals such as means.

shimmerLocal = <numeric> [Default: 0]

1 = enable computation of F0 shimmer (amplitude variations). shimmerLocal = the average absolute difference between the interpolated peak amplitudes of consecutive periods, divided by the average peak amplitude of all periods in the frame.

shimmerLocalEnv = <numeric> [Default: 0]

1 = compute envelope of shimmerLocal (i.e. fill shimmer values in unvoiced frames with value of last voiced segment). Use this in conjunction with statistical functionals such as means.

onlyVoiced = <numeric> [Default: 0]

1 = produce output only for voiced frames. I.e. do not output 0 jitter/shimmer values for unvoiced frames. WARNING: this option is not fully supported by the functionals component, yet.

periodLengths = <numeric> [Default: 0]

1 = enable output of individual period lengths.

periodStarts = <numeric> [Default: 0]

1 = enable output of individual period start times.

cPitchShs

Description: This component computes the fundamental frequency via the Sub-Harmonic-Summation (SHS) [?] method (this is related to the Harmonic Product Spectrum method). This method is also used in Praat. The method allows to identify the fundamental frequency from the harmonic structure, i. e. the fundamental frequency can be identified even if the actual fundamental is missing due to a bandpass channel (such as in telephone speech). The cPitchSmootherViterbi component implements pitch smoothing with a modification of the Viterbi algorithm described in [?].

This component expects an octave scale (log-domain) magnitude spectrum as input (this can be generated by the ‘cSpecScale’ (section 3.3.6) component). The main steps of the SHS algorithm implemented in this component are:

- Subharmonic summation: The input spectrum is shifted left by an octave, multiplied by the compression factor, and added to the original, unshifted version of itself. This is repeated ‘nHarmonics’ times (shifting by 2, 3, etc. octaves respectively and multiplying with powers of the compression factor). The output is the subharmonic summation spectrum.
- Peak picking: The ‘nCandidates’ highest (amplitude) peaks in the subharmonic summation spectrum are found via standard peak picking methods. The actual peak amplitudes and locations are interpolated by quadratic curve fitting through the peak point and its left and right neighbour. The amplitudes (after interpolation) of the peaks are saved as the pitch candidate scores and the positions of the peaks on the frequency axis are saved as F_0 values of the pitch candidates.
- Voicing probability estimation: The arithmetic mean (μ_s) of the bins in the subharmonic summation spectrum is computed. For each pitch candidate i with a pitch candidate score s_{ci} (= peak amplitude) greater than μ_s the voicing probability p_{vi} is computed as $p_{vi} = 1.0 - \frac{\mu_s}{s_{ci}}$. Otherwise ($s_{ci} \leq \mu_s$), $p_{vi} = 0$.

In order to pick the most prominent pitch candidate and perform some short term smoothing and octave error correction you should use the ‘cPitchSmoother’ component (section 3.3.6).

Class hierarchy: cSmileComponent \rightarrow cDataProcessor \rightarrow cVectorProcessor \rightarrow cPitchBase \rightarrow cPitchSHS

Configuration options:

reader = <object of type ‘cDataReader’> The configuration of the cDataReader sub-component, which handles the dataMemory interface for data input. See the documentation of ‘cDataReader’ for more information (section 3.3.2).

writer = <object of type ‘cDataWriter’> The configuration of the cDataWriter sub-component, which handles the dataMemory interface for data output. See the documentation of ‘cDataWriter’ for more information (section 3.3.2).

bufferSize = <numeric> [Default: 0]
The buffer size for the output level, in frames (default [0] = same as input level), this option overwrites ‘bufferSize_sec’.

bufferSize_sec = <numeric> [Default: 0]
The buffer size for the output level, in seconds (default [0] = same as input level).

blockSize = <numeric> [Default: 0]
The size of data blocks to process, in frames (this sets both ‘blockSizeR’ and ‘blockSizeW’, and overwrites ‘blockSize_sec’).

blockSizeR = <numeric> [Default: 0]
The size of data blocks to read, in frames (overwrites ‘blockSize’).

blockSizeW = <numeric> [Default: 0]
The size of data blocks to write, in frames (overwrites ‘blockSize’).

- blocksize_sec** = <numeric> [Default: 0]
 The size of data blocks to process, in seconds (this sets both ‘blocksizeR_sec’ and ‘blocksizeW_sec’).
- blocksizeR_sec** = <numeric> [Default: 0]
 The size of data blocks to read, in seconds (overwrites ‘blocksize_sec’).
- blocksizeW_sec** = <numeric> [Default: 0]
 The size of data blocks to write in seconds (overwrites ‘blocksize_sec’).
- nameAppend** = <string> [Default: ‘(null)’]
 A string suffix to append to the input field names (default: empty).
- copyInputName** = <numeric> [Default: 1]
 1 = copy the input name (and optionally append a suffix, see the ‘nameAppend’ option)
 0 = discard the input name and use only the ‘nameAppend’ string as new name.
- processArrayFields** = <numeric> [Default: 0]
 1 = process each array field as one vector individually (and produce one output for each input array field). Only array fields (i.e. fields with more than one element) are processed if this is set.
 0 = process complete input frame as one vector, ignoring the field/element structure.
- maxPitch** = <numeric> [Default: 620]
 Maximum detectable pitch in Hz.
- minPitch** = <numeric> [Default: 52]
 Minimum detectable pitch in Hz.
- nCandidates** = <numeric> [Default: 3]
 The number of F_0 candidates to output [1-20] (0 disables output of candidates AND their voicing probabilities)
- scores** = <numeric> [Default: 1]
 1/0 = on/off: output of F0 candidates scores, if available.
- voicing** = <numeric> [Default: 1]
 1/0 = on/off: output of voicing probability for F0 candidates.
- F0C1** = <numeric> [Default: 0]
 1/0 = on/off: output of raw best F0 candidate without thresholding (forcing to 0) in unvoiced segments.
- voicingC1** = <numeric> [Default: 0]
 1/0 = on/off: output of output voicing (pseudo) probability for best candidate.
- F0raw** = <numeric> [Default: 0]
 1/0 = on/off: output of raw F0 (best candidate), > 0 only for voiced segments (using voicingCutoff threshold).
- voicingClip** = <numeric> [Default: 0]
 1/0 = on/off: output of voicing of raw F0 (best candidate), > 0 only for voiced segments (using voicingCutoff threshold).

voicingCutoff = <numeric> [Default: 0.75]

This sets the voicing (pseudo) probability threshold for pitch detection. Frames with voicing probability values above this threshold will be considered as voiced.

inputFieldSearch = <string> [Default: 'Mag_logScale']

A part of the name to find the pitch detectors input field by ('Mag' searches e.g. for *Mag*, and will match fftMag fields)

octaveCorrection = <numeric> [Default: 0]

1 = enable low-level octave correction tuned for the SHS algorithm (will affect F0C1, voicingC1 and F0raw output fields) [EXPERIMENTAL! MAY BREAK CORRECT PITCH DETECTION!]

nHarmonics = <numeric> [Default: 15]

Number of harmonics to consider for subharmonic sampling (feasible values: 5-15).

compressionFactor = <numeric> [Default: 0.85]

The factor for successive compression of sub-harmonics.

cPitchSmoother

Description: This component performs temporal pitch smoothing and pitch octave correction. Input: candidates produced by a pitchBase descendant (e.g. cPitchShs).

Class hierarchy: cSmileComponent → cDataProcessor → cVectorProcessor → cPitchSmoother

Configuration options:

reader = <object of type 'cDataReader'> The configuration of the cDataReader sub-component, which handles the dataMemory interface for data input. See the documentation of 'cDataReader' for more information (section 3.3.2).

writer = <object of type 'cDataWriter'> The configuration of the cDataWriter sub-component, which handles the dataMemory interface for data output. See the documentation of 'cDataWriter' for more information (section 3.3.2).

bufferSize = <numeric> [Default: 0]

The buffer size for the output level, in frames (default [0] = same as input level), this option overwrites 'bufferSize_sec'.

bufferSize_sec = <numeric> [Default: 0]

The buffer size for the output level, in seconds (default [0] = same as input level).

blockSize = <numeric> [Default: 0]

The size of data blocks to process, in frames (this sets both 'blockSizeR' and 'blockSizeW', and overwrites 'blockSize_sec').

blockSizeR = <numeric> [Default: 0]

The size of data blocks to read, in frames (overwrites 'blockSize').

blockSizeW = <numeric> [Default: 0]

The size of data blocks to write, in frames (overwrites 'blockSize').

- blocksize_sec** = <numeric> [Default: 0]
 The size of data blocks to process, in seconds (this sets both ‘blocksizeR_sec’ and ‘blocksizeW_sec’).
- blocksizeR_sec** = <numeric> [Default: 0]
 The size of data blocks to read, in seconds (overwrites ‘blocksize_sec’).
- blocksizeW_sec** = <numeric> [Default: 0]
 The size of data blocks to write in seconds (overwrites ‘blocksize_sec’).
- nameAppend** = <string> [Default: ‘(null)’]
 A string suffix to append to the input field names (default: empty).
- copyInputName** = <numeric> [Default: 1]
 1 = copy the input name (and optionally append a suffix, see the ‘nameAppend’ option)
 0 = discard the input name and use only the ‘nameAppend’ string as new name.
- processArrayFields** = <numeric> [Default: 0]
 1 = process each array field as one vector individually (and produce one output for each input array field). Only array fields (i.e. fields with more than one element) are processed if this is set.
 0 = process complete input frame as one vector, ignoring the field/element structure.
- medianFilter0** = <numeric> [Default: 0]
 Apply median filtering of candidates as the *first* processing step; the filter length is ‘medianFilter0’ if > 0.
- postSmoothing** = <numeric> [Default: 0]
 Apply post processing (median and spike remover) over ‘postSmoothing’ frames (0 = no smoothing or use default set by ‘postSmoothingMethod’ option).
- postSmoothingMethod** = <string> [Default: ‘simple’]
 Post processing method to use. One of the following:
none disable post smoothing
simple simple post smoothing using only 1 frame delay (will smooth out 1 frame octave spikes)
median will apply a median filter to the output values (length = value of ‘postProcessing’ option)
- octaveCorrection** = <numeric> [Default: 1]
 Enable intelligent cross candidate fundamental frequency octave correction.
- F0final** = <numeric> [Default: 1]
 1 = Enable output of final (corrected and smoothed) F_0 .
- F0finalEnv** = <numeric> [Default: 0]
 1 = Enable output of envelope of final smoothed F_0 (i.e. there will be no 0 values (except for end and beginning)).
- no0f0** = <numeric> [Default: 0]
 1 = enable ‘no zero F_0 ’, output data only when $F_0 > 0$, i.e. a voiced frame is detected. This may cause problem with some functionals and framer components, which don’t support this variable length data yet.

voicingFinalClipped = <numeric> [Default: 0]

1 = Enable output of final smoothed and clipped voicing (pseudo) probability. ‘Clipped’ means that the voicing probability is set to 0 for unvoiced regions, i.e. where the probability lies below the voicing threshold.

voicingFinalUnclipped = <numeric> [Default: 0]

1 = Enable output of final smoothed, raw voicing (pseudo) probability (*unclipped*: not set to 0 during unvoiced regions).

F0raw = <numeric> [Default: 0]

1 = Enable output of ‘F0raw’ copied from input.

voicingC1 = <numeric> [Default: 0]

1 = Enable output of ‘voicingC1’ copied from input.

voicingClip = <numeric> [Default: 0]

1 = Enable output of ‘voicingClip’ copied from input.

cPlp

Description: This component computes perceptual linear predictive cepstral coefficients (PLP) and RASTA-PLP from a critical band spectrum (generated by the ‘cMelspec’ component (section 3.3.6, for example)). The component is capable of performing the following processing steps:

1. Take the natural logarithm of the critical band powers
2. RASTA filtering
3. Computation of auditory spectrum (equal loudness curve and loudness compression applied to critical band spectrum)
4. Inverse of the natural logarithm
5. Inverse DFT to obtain autocorrelation coefficients
6. Linear prediction analysis (Durbin recursion) on autocorrelation coefficients
7. Computation of cepstral coefficients from lp coefficients
8. Cepstral ‘liftering’

Each one of these steps can be enabled and disabled individually.

Class hierarchy: cSmileComponent → cDataProcessor → cVectorProcessor → cPlp

Configuration options:

reader = <object of type ‘cDataReader’> The configuration of the cDataReader sub-component, which handles the dataMemory interface for data input. See the documentation of ‘cDataReader’ for more information (section 3.3.2).

writer = <object of type ‘cDataWriter’> The configuration of the cDataWriter sub-component, which handles the dataMemory interface for data output. See the documentation of ‘cDataWriter’ for more information (section 3.3.2).

- bufferSize** = <numeric> [Default: 0]
The buffer size for the output level, in frames (default [0] = same as input level), this option overwrites 'bufferSize_sec'.
- bufferSize_sec** = <numeric> [Default: 0]
The buffer size for the output level, in seconds (default [0] = same as input level).
- blockSize** = <numeric> [Default: 0]
The size of data blocks to process, in frames (this sets both 'blockSizeR' and 'blockSizeW', and overwrites 'blockSize_sec').
- blockSizeR** = <numeric> [Default: 0]
The size of data blocks to read, in frames (overwrites 'blockSize').
- blockSizeW** = <numeric> [Default: 0]
The size of data blocks to write, in frames (overwrites 'blockSize').
- blockSize_sec** = <numeric> [Default: 0]
The size of data blocks to process, in seconds (this sets both 'blockSizeR_sec' and 'blockSizeW_sec').
- blockSizeR_sec** = <numeric> [Default: 0]
The size of data blocks to read, in seconds (overwrites 'blockSize_sec').
- blockSizeW_sec** = <numeric> [Default: 0]
The size of data blocks to write in seconds (overwrites 'blockSize_sec').
- nameAppend** = <string> [Default: '(null)']
A string suffix to append to the input field names (default: empty).
- copyInputName** = <numeric> [Default: 1]
1 = copy the input name (and optionally append a suffix, see the 'nameAppend' option)
0 = discard the input name and use only the 'nameAppend' string as new name.
- processArrayFields** = <numeric> [Default: 1]
1 = process each array field as one vector individually (and produce one output for each input array field). Only array fields (i.e. fields with more than one element) are processed if this is set.
0 = process complete input frame as one vector, ignoring the field/element structure.
- lpOrder** = <numeric> [Default: 5]
The order of the linear predictor (5th order is optimal according to Hermansky 1990, JASA).
- nCeps** = <numeric> [Default: -1]
The number of cepstral coefficients (must be <= 'lpOrder'; set to -1 for nCeps = lpOrder)
- firstCC** = <numeric> [Default: 1]
The first cepstral coefficient to compute (set to 0 to include the 0-th coefficient, which is defined as $-\log(1/\text{lpGain})$).
- lastCC** = <numeric> [Default: -1]
The last cepstral coefficient to compute (set to -1 to use nCeps, else lastCC will override nCeps!).

doLog = <numeric> [Default: 1]
 Take the log of input bands (1=yes / 0=no).

doAud = <numeric> [Default: 1]
 Do auditory processing (equal loudness curve and loudness compression) (1=yes / 0=no).

RASTA = <numeric> [Default: 0]
 Perform RASTA (temporal) filtering (1=yes / 0=no).

rastraUpperCutoff = <numeric> [Default: 29.0]
 Upper cut-off frequency of RASTA bandpass filter in Hz.

rastraLowerCutoff = <numeric> [Default: 1.0]
 Lower cut-off frequency of RASTA bandpass filter in Hz.

doInvLog = <numeric> [Default: 1]
 Apply inverse logarithm after power compression (1=yes / 0=no).

doIDFT = <numeric> [Default: 1]
 Apply I(nverse)DFT after power compression and inverse log (1=yes / 0=no).

doLP = <numeric> [Default: 1]
 Do lp analysis on autocorrelation function (1=yes / 0=no).

doLpToCeps = <numeric> [Default: 1]
 Convert lp coefficients to cepstral coefficients (1=yes / 0=no).

cepLifter = <numeric> [Default: 0]
 Parameter for cepstral ‘liftering’, set to 0.0 to disable cepstral liftering.

compression = <numeric> [Default: 0.33]
 Compression factor for applying the ‘power law of hearing’.

melfloor = <numeric> [Default: 9.3e-10]
 Minimum value of Mel-spectra when computing cepstral coefficients (will be forced to 1.0 when htkcompatible=1).

htkcompatible = <numeric> [Default: 1]
 Set correct Mel-floor (1.0) and force HTK compatible PLP output (1/0 = yes/no).
 htkcompatible = 1, forces the following settings:

- **melfloor** = 1.0 (signal scaling 0..32767*32767)
- append 0-th coeff instead of having it as first value
- **doAud** = 1 , **doLog**=0 , **doInvLog**=0 (**doIDFT**, **doLP**, and **doLpToCeps** are not forced to 1, this enables generation of HTK compatible auditory spectra, etc. (these, of course, are not compatible, i.e. are not the same as HTK’s PLP-CC))
- the 0-th audspec component is used as DC component in the i-DFT (else the DC component is zero).

cPreemphasis

Description: This component performs Pre- and De-emphasis of speech signals using a 1-st order difference equation: $y[n] = x[n] - k \cdot x[n - 1]$.

Class hierarchy: cSmileComponent \rightarrow cDataProcessor \rightarrow cWindowProcessor \rightarrow cPreemphasis

Configuration options:

reader = <object of type 'cDataReader'> The configuration of the cDataReader sub-component, which handles the dataMemory interface for data input. See the documentation of 'cDataReader' for more information (section 3.3.2).

writer = <object of type 'cDataWriter'> The configuration of the cDataWriter sub-component, which handles the dataMemory interface for data output. See the documentation of 'cDataWriter' for more information (section 3.3.2).

bufferize = <numeric> [Default: 0]
The buffer size for the output level, in frames (default [0] = same as input level), this option overwrites 'bufferize_sec'.

bufferize_sec = <numeric> [Default: 0]
The buffer size for the output level, in seconds (default [0] = same as input level).

blocksize = <numeric> [Default: 0]
The size of data blocks to process, in frames (this sets both 'blocksizeR' and 'blocksizeW', and overwrites 'blocksize_sec').

blocksizeR = <numeric> [Default: 0]
The size of data blocks to read, in frames (overwrites 'blocksize').

blocksizeW = <numeric> [Default: 0]
The size of data blocks to write, in frames (overwrites 'blocksize').

blocksize_sec = <numeric> [Default: 0]
The size of data blocks to process, in seconds (this sets both 'blocksizeR_sec' and 'blocksizeW_sec').

blocksizeR_sec = <numeric> [Default: 0]
The size of data blocks to read, in seconds (overwrites 'blocksize_sec').

blocksizeW_sec = <numeric> [Default: 0]
The size of data blocks to write in seconds (overwrites 'blocksize_sec').

nameAppend = <string> [Default: '(null)']
A string suffix to append to the input field names (default: empty).

copyInputName = <numeric> [Default: 1]
1 = copy the input name (and optionally append a suffix, see the 'nameAppend' option)
0 = discard the input name and use only the 'nameAppend' string as new name.

noPostEOIprocessing = <numeric> [Default: 0]
1 = do not process incomplete windows at the end of the input.

k = <numeric> [Default: 0.97]

The pre-emphasis coefficient k in $y[n] = x[n] - k \cdot x[n - 1]$.

f = <numeric> [Default: 0]

The pre-emphasis frequency f in Hz, which can be used to compute the filter coefficient k : $k = \exp\left(-2\pi \cdot \frac{f}{\text{samplingFreq.}}\right)$ (if set, f will override k !)

de = <numeric> [Default: 0]

1 = perform de-emphasis instead of pre-emphasis (i.e. $y[n] = x[n] + k \cdot x[n - 1]$)

cSemaineSpeakerID1

Description: This component implements voice activity detection based on an on-line self-adapting, euclidean distance based, nearest neighbour classifier. It supports a user voice model, a background noise/silence model, and multiple agent voice models for discriminating the agent voice from the user's voice. *Required input:* A field with various features such as MFCC, PLP, Spectra, etc., followed by a 'vadBin' field, created by a rule-based voice activity component.

Class hierarchy: cSmileComponent \rightarrow cDataProcessor \rightarrow cSemaineSpeakerID1

Configuration options:

reader = <object of type 'cDataReader'> The configuration of the cDataReader sub-component, which handles the dataMemory interface for data input. See the documentation of 'cDataReader' for more information (section 3.3.2).

writer = <object of type 'cDataWriter'> The configuration of the cDataWriter sub-component, which handles the dataMemory interface for data output. See the documentation of 'cDataWriter' for more information (section 3.3.2).

bufferSize = <numeric> [Default: 0]

The buffer size for the output level, in frames (default [0] = same as input level), this option overwrites 'bufferSize_sec'.

bufferSize_sec = <numeric> [Default: 0]

The buffer size for the output level, in seconds (default [0] = same as input level).

blockSize = <numeric> [Default: 0]

The size of data blocks to process, in frames (this sets both 'blockSizeR' and 'blockSizeW', and overwrites 'blockSize_sec').

blockSizeR = <numeric> [Default: 0]

The size of data blocks to read, in frames (overwrites 'blockSize').

blockSizeW = <numeric> [Default: 0]

The size of data blocks to write, in frames (overwrites 'blockSize').

blockSize_sec = <numeric> [Default: 0]

The size of data blocks to process, in seconds (this sets both 'blockSizeR_sec' and 'blockSizeW_sec').

blockSizeR_sec = <numeric> [Default: 0]

The size of data blocks to read, in seconds (overwrites 'blockSize_sec').

blocksizeW_sec = <numeric> [Default: 0]

The size of data blocks to write in seconds (overwrites 'blocksize_sec').

nameAppend = <string> [Default: '(null)']

A string suffix to append to the input field names (default: empty).

copyInputName = <numeric> [Default: 1]

1 = copy the input name (and optionally append a suffix, see the 'nameAppend' option)

0 = discard the input name and use only the 'nameAppend' string as new name.

trainDur = <numeric> [Default: 500]

The duration of the initial training phase, in frames. The training phase will end when at least 'trainDur' frames for each model (voice and noise) have been collected. Thus it might happen that more frames have been collected for the noise model than for the speech model or vice versa before the training phase ends. During the training phase the model is not used for auto-updating of itself, the rule based voice activity input is used during this stage.

agentBlockTime = <numeric> [Default: 1000]

The initial user speech time (in frames) during which to block detection of voice activity while the agent speaks (this information is received via smile messages). This must be high enough in order to have robust enough models and to avoid adapting the user speech model to the agent's voice.

maxTrainTime = <numeric> [Default: 10000]

The maximum time to re-train the user model for (only user speech time, measured from the beginning of the turn). Use this to prevent over adaptation, since practical observations revealed that the models become unstable after a longer period of time. Set this to -1 for infinite re-training.

debug = <numeric> [Default: 0]

Show detailed classification result as debug output. Use only for debugging!

a0 = <numeric> [Default: 0.05]

The decay factor of fuzzy vad smoothing.

a1 = <numeric> [Default: 0.2]

The attack factor of fuzzy vad (i.e. a0) smoothing.

initSpeechModel = <string> [Default: '(null)']

The filename of the file from which to load initial user speech model coefficients from.

initNoiseModel = <string> [Default: '(null)']

The filename of the file from which to load initial noise model coefficients from.

initAgentModel = <string> [Default: 'model.agent']

The filename of the file from which to load initial learn agent voice model coefficients from.

agentModelWeight = <numeric> [Default: -1]

The weight of the initial agent model(s). This represents the number of frames the model represents; set to this to -1 to disable on-line model updates, set to -2 to read the weight from the model file, set to 0 to disable the initial model.

noiseModelWeight = <numeric> [Default: 2000]

The weight of the initial noise model. This represents the number of frames the model represents; set to -1 to disable on-line model updates, set to -2 to read the weight from the model file, set to 0 to disable the initial model.

speechModelWeight = <numeric> [Default: 1000]

The weight of the initial user speech model. This represents the number of frames the model represents; set to -1 to disable on-line model updates, set to -2 to read the weight from the model file, set to 0 to disable the initial model.

saveSpeechModel = <string> [Default: '(null)']

The file to save the current user speech model coefficients to, when openSMILE reaches the end of the input or is terminated gracefully from an external source (e.g. via Ctrl+C).

saveNoiseModel = <string> [Default: '(null)']

The file to save the current noise model coefficients to, when openSMILE reaches the end of the input or is terminated gracefully from an external source (e.g. via Ctrl+C).

saveAgentModel = <string> [Default: '(null)']

The file to save the current agent speech model coefficients to, when openSMILE reaches the end of the input or is terminated gracefully from an external source (e.g. via Ctrl+C).

numAgents = <numeric> [Default: 4]

The number of agent voices (i.e. number of models in agent model file).

alwaysRejectAgent = <numeric> [Default: 0]

1 = never detect user speech activity, while the agent is speaking (currently agent speech is only determined on the basis of messages received from other components).

ruleVadOnly = <numeric> [Default: 0]

1 = do not use speaker adaptive voice activity detection, use only rule based vad with dynamic thresholds (i.e. this component is bypassed and the input is forwarded to the output). (This option also implies 'alwaysRejectAgent=1')

cSmileResample

Description: This component implements a spectral domain resampling algorithm. Input frames are transferred to the spectral domain using an FFT, and a modified DFT is performed to synthesise samples at the new rate.

Class hierarchy: cSmileComponent → cDataProcessor → cSmileResample

Configuration options:

reader = <object of type 'cDataReader'> The configuration of the cDataReader sub-component, which handles the dataMemory interface for data input. See the documentation of 'cDataReader' for more information (section 3.3.2).

writer = <object of type 'cDataWriter'> The configuration of the cDataWriter sub-component, which handles the dataMemory interface for data output. See the documentation of 'cDataWriter' for more information (section 3.3.2).

bufferSize = <numeric> [Default: 0]

The buffer size for the output level, in frames (default [0] = same as input level), this option overwrites 'bufferSize_sec'.

bufferSize_sec = <numeric> [Default: 0]

The buffer size for the output level, in seconds (default [0] = same as input level).

blocksize = <numeric> [Default: 0]

The size of data blocks to process, in frames (this sets both 'blocksizeR' and 'blocksizeW', and overwrites 'blocksize_sec').

blocksizeR = <numeric> [Default: 0]

The size of data blocks to read, in frames (overwrites 'blocksize').

blocksizeW = <numeric> [Default: 0]

The size of data blocks to write, in frames (overwrites 'blocksize').

blocksize_sec = <numeric> [Default: 0]

The size of data blocks to process, in seconds (this sets both 'blocksizeR_sec' and 'blocksizeW_sec').

blocksizeR_sec = <numeric> [Default: 0]

The size of data blocks to read, in seconds (overwrites 'blocksize_sec').

blocksizeW_sec = <numeric> [Default: 0]

The size of data blocks to write in seconds (overwrites 'blocksize_sec').

nameAppend = <string> [Default: '(null)']

A string suffix to append to the input field names (default: empty).

copyInputName = <numeric> [Default: 1]

1 = copy the input name (and optionally append a suffix, see the 'nameAppend' option)
0 = discard the input name and use only the 'nameAppend' string as new name.

targetRate = <numeric> [Default: 16000]

The target sampling frequency in Hz.

resampleRatio = <numeric> [Default: 1]

A fixed resample ratio $a = f_{sNew}/f_{sCurrent}$. If set, this overrides the 'targetRate' option.

pitchRatio = <numeric> [Default: 1]

Low-quality pitch scaling factor, if $\neq 1.0$.

winSize = <numeric> [Default: 0.03]

Internal window size in seconds (will be rounded to nearest power of 2 frame size internally). This affects the quality of the re-sampling and the accuracy of the target sampling rate. Larger window sizes allow for a more accurate target sampling frequency, i.e. less pitch distortion.

cSpecResample

Description: This component implements a spectral domain resampling component. Input frames are complex valued spectral domain data, which will be shifted and scaled by this component, and a modified DFT is performed to synthesise samples at the new rate.

Class hierarchy: cSmileComponent \rightarrow cDataProcessor \rightarrow cVectorProcessor \rightarrow cSpecResample

Configuration options:

reader = <object of type ‘cDataReader’> The configuration of the cDataReader sub-component, which handles the dataMemory interface for data input. See the documentation of ‘cDataReader’ for more information (section 3.3.2).

writer = <object of type ‘cDataWriter’> The configuration of the cDataWriter sub-component, which handles the dataMemory interface for data output. See the documentation of ‘cDataWriter’ for more information (section 3.3.2).

bufferSize = <numeric> [Default: 0]
The buffer size for the output level, in frames (default [0] = same as input level), this option overwrites ‘bufferSize_sec’.

bufferSize_sec = <numeric> [Default: 0]
The buffer size for the output level, in seconds (default [0] = same as input level).

blockSize = <numeric> [Default: 0]
The size of data blocks to process, in frames (this sets both ‘blockSizeR’ and ‘blockSizeW’, and overwrites ‘blockSize_sec’).

blockSizeR = <numeric> [Default: 0]
The size of data blocks to read, in frames (overwrites ‘blockSize’).

blockSizeW = <numeric> [Default: 0]
The size of data blocks to write, in frames (overwrites ‘blockSize’).

blockSize_sec = <numeric> [Default: 0]
The size of data blocks to process, in seconds (this sets both ‘blockSizeR_sec’ and ‘blockSizeW_sec’).

blockSizeR_sec = <numeric> [Default: 0]
The size of data blocks to read, in seconds (overwrites ‘blockSize_sec’).

blockSizeW_sec = <numeric> [Default: 0]
The size of data blocks to write in seconds (overwrites ‘blockSize_sec’).

nameAppend = <string> [Default: ‘(null)’]
A string suffix to append to the input field names (default: empty).

copyInputName = <numeric> [Default: 1]
1 = copy the input name (and optionally append a suffix, see the ‘nameAppend’ option)
0 = discard the input name and use only the ‘nameAppend’ string as new name.

processArrayFields = <numeric> [Default: 0]
1 = process each array field as one vector individually (and produce one output for each input array field). Only array fields (i.e. fields with more than one element) are processed if this is set.
0 = process complete input frame as one vector, ignoring field/element structure.

targetFs = <numeric> [Default: 16000]
The target sampling frequency in Hz.

resampleRatio = <numeric> [Default: 1]
Specifies a fixed resample ratio a ($a = fs_{\text{New}} / fs_{\text{Current}}$). If set, this overrides ‘targetFs’.

inputFieldPartial = <string> [Default: '(null)']

The name of the input field to search for. (NULL (default): use full input vector)

cSpecScale

Description: This component performs linear/non-linear axis scaling of FFT magnitude spectra with spline interpolation.

Class hierarchy: cSmileComponent → cDataProcessor → cVectorProcessor → cSpecScale

Configuration options:

reader = <object of type 'cDataReader'> The configuration of the cDataReader sub-component, which handles the dataMemory interface for data input. See the documentation of 'cDataReader' for more information (section 3.3.2).

writer = <object of type 'cDataWriter'> The configuration of the cDataWriter sub-component, which handles the dataMemory interface for data output. See the documentation of 'cDataWriter' for more information (section 3.3.2).

bufferSize = <numeric> [Default: 0]

The buffer size for the output level, in frames (default [0] = same as input level), this option overwrites 'bufferSize_sec'.

bufferSize_sec = <numeric> [Default: 0]

The buffer size for the output level, in seconds (default [0] = same as input level).

blockSize = <numeric> [Default: 0]

The size of data blocks to process, in frames (this sets both 'blockSizeR' and 'blockSizeW', and overwrites 'blockSize_sec').

blockSizeR = <numeric> [Default: 0]

The size of data blocks to read, in frames (overwrites 'blockSize').

blockSizeW = <numeric> [Default: 0]

The size of data blocks to write, in frames (overwrites 'blockSize').

blockSize_sec = <numeric> [Default: 0]

The size of data blocks to process, in seconds (this sets both 'blockSizeR_sec' and 'blockSizeW_sec').

blockSizeR_sec = <numeric> [Default: 0]

The size of data blocks to read, in seconds (overwrites 'blockSize_sec').

blockSizeW_sec = <numeric> [Default: 0]

The size of data blocks to write in seconds (overwrites 'blockSize_sec').

nameAppend = <string> [Default: '(null)']

A string suffix to append to the input field names (default: empty).

copyInputName = <numeric> [Default: 1]

1 = copy the input name (and optionally append a suffix, see the 'nameAppend' option)

0 = discard the input name and use only the 'nameAppend' string as new name.

- processArrayFields** = **<numeric>** [Default: 0]
 1 = process each array field as one vector individually (and produce one output for each input array field). Only array fields (i.e. fields with more than one element) are processed if this is set.
 0 = process complete input frame as one vector, ignoring field/element structure.
- scale** = **<string>** [Default: 'log']
 The target scale. One of the following:
log(arithmetic) logarithmic scale, see 'logScaleBase'
oct(ave) octave scale = logarithmic scale with base 2
sem(itone) musical semi-tone scale (see 'firstNote' option)
lin(ear) linear scale
- sourceScale** = **<string>** [Default: 'lin']
 The source scale (currently only 'lin(ear)' is supported).
- logScaleBase** = **<numeric>** [Default: 2]
 The base for log scales (a log base of 2.0 – the default – corresponds to an octave target scale).
- logSourceScaleBase** = **<numeric>** [Default: 2]
 The base for log source scales (a log base of 2.0 - the default - corresponds to an octave source scale)
- firstNote** = **<numeric>** [Default: 55]
 The first note (in Hz) for a semi-tone scale.
- interpMethod** = **<string>** [Default: 'spline']
 The interpolation method for rescaled spectra, choose between: 'none', 'spline'
- minF** = **<numeric>** [Default: 25]
 The minimum frequency of the target scale.
- maxF** = **<numeric>** [Default: -1]
 The maximum frequency of the target scale (-1.0 : set to maximum frequency of the source spectrum)
- nPointsTarget** = **<numeric>** [Default: 0]
 The number of frequency points in the target spectrum (j= 0 : same as input spectrum)
- specSmooth** = **<numeric>** [Default: 0]
 1 = perform spectral smoothing before applying the scale transformation.
- specEnhance** = **<numeric>** [Default: 0]
 1 = do spectral peak enhancement before applying smoothing (if enabled) and the scale transformation.
- auditoryWeighting** = **<numeric>** [Default: 0]
 1 = enable auditory weighting after scale transformation (this is currently only supported for octave (log2) scales).

cSpectral

Description: This component computes spectral features such as the spectral flux, roll-off points, spectral centroid, position of spectral maximum and minimum, and user defined band energies (rectangular summation of FFT magnitudes).

Class hierarchy: cSmileComponent → cDataProcessor → cVectorProcessor → cSpectral

Configuration options:

reader = <object of type 'cDataReader'> The configuration of the cDataReader sub-component, which handles the dataMemory interface for data input. See the documentation of 'cDataReader' for more information (section 3.3.2).

writer = <object of type 'cDataWriter'> The configuration of the cDataWriter sub-component, which handles the dataMemory interface for data output. See the documentation of 'cDataWriter' for more information (section 3.3.2).

bufferize = <numeric> [Default: 0]
The buffer size for the output level, in frames (default [0] = same as input level), this option overwrites 'bufferize_sec'.

bufferize_sec = <numeric> [Default: 0]
The buffer size for the output level, in seconds (default [0] = same as input level).

blocksize = <numeric> [Default: 0]
The size of data blocks to process, in frames (this sets both 'blocksizeR' and 'blocksizeW', and overwrites 'blocksize_sec').

blocksizeR = <numeric> [Default: 0]
The size of data blocks to read, in frames (overwrites 'blocksize').

blocksizeW = <numeric> [Default: 0]
The size of data blocks to write, in frames (overwrites 'blocksize').

blocksize_sec = <numeric> [Default: 0]
The size of data blocks to process, in seconds (this sets both 'blocksizeR_sec' and 'blocksizeW_sec').

blocksizeR_sec = <numeric> [Default: 0]
The size of data blocks to read, in seconds (overwrites 'blocksize_sec').

blocksizeW_sec = <numeric> [Default: 0]
The size of data blocks to write in seconds (overwrites 'blocksize_sec').

nameAppend = <string> [Default: '(null)']
A string suffix to append to the input field names (default: empty).

copyInputName = <numeric> [Default: 1]
1 = copy the input name (and optionally append a suffix, see the 'nameAppend' option)
0 = discard the input name and use only the 'nameAppend' string as new name.

processArrayFields = <numeric> [Default: 0]
1 = process each array field as one vector individually (and produce one output for each input array field). Only array fields (i.e. fields with more than one element) are processed

if this is set.

0 = process complete input frame as one vector, ignoring field/element structure.

squareInput = <numeric> [Default: 1]

1/0 = square input values (e.g. if input is magnitude and not power spectrum)

bands[= <string>] [Default: '250-650']

bands[n] = LoFrq(Hz)-HiFrq(Hz) (e.g. 0-250), compute energy in the given spectral band by summation of FFT bins in this band.

rollOff[= <numeric>] [Default: 0.9]

rollOff[n] = X (X in the range 0..1), compute X*100 percent spectral roll-off point. The X * 100 percent spectral roll-off point is determined as the frequency below which X * 100 percent of the total signal energy fall. Sometimes we only find the term 'spectral roll-off' in the literature (especially Music Information Retrieval), it then refers to the 95% roll-off point.

flux = <numeric> [Default: 1]

(1/0=yes/no) enable computation of spectral flux. Spectral Flux F_S^t for N FFT bins is computed via equation 3.2, whereby E^t is the energy of the frame at time t .

$$F_S^t = \sqrt{\frac{1}{N} \sum_{f=1}^N \left(\frac{X^t(f)}{E^t} - \frac{X^{t-1}(f)}{E^{t-1}} \right)^2} \quad (3.2)$$

centroid = <numeric> [Default: 1]

(1/0=yes/no) enable computation of spectral centroid. Spectral centroid (C_S^t) at time t is computed via equation 3.3. $X^t(f)$ is the spectral magnitude at time t in bin f .

$$C_S^t = \frac{\sum_{\forall f} f \cdot X^t(f)}{\sum_{\forall f} X^t(f)} \quad (3.3)$$

maxPos = <numeric> [Default: 1]

(1/0=yes/no) enable computation of the position of the maximum magnitude spectral bin (in Hz).

minPos = <numeric> [Default: 1]

(1/0=yes/no) enable computation of position of the minimum magnitude spectral bin (in Hz).

entropy = <numeric> [Default: 0]

(1/0=yes/no) enable computation of the spectral entropy. The spectral entropy is computed from the normalised magnitude spectrum (or power spectrum, if 'squareInput=1'). $X^t(f)$ is the spectral magnitude at time t in bin f . The normalised spectrum $X_n^t(f)$ is obtained as:

$$X_n^t(f) = \frac{X^t(f)}{\sum_{\forall f} X^t(f)} \quad (3.4)$$

The entropy H is then computed as:

$$H = - \sum_{\forall f} X_n^t(f) \cdot \log_2(X_n^t(f)) \quad (3.5)$$

variance = <numeric> [Default: 0]

(1/0=yes/no) enable computation of spectral variance (mpeg7: spectral spread). The spectral variance is computed from the magnitude spectrum (or power spectrum, if 'squareInput=1'). $X^t(f)$ is the spectral magnitude at time t in bin f . The variance σ^2 is then computed as:

$$\sigma^2 = \frac{1}{\sum_{\forall f} X^t(f)} \sum_{\forall f} (f - C_S^t)^2 \cdot X^t(f) \quad (3.6)$$

skewness = <numeric> [Default: 0]

(1/0=yes/no) enable computation of spectral skewness. This is a measure of the asymmetry of the spectral distribution around its centroid C_S^t . The spectral skewness is computed from the magnitude spectrum (or power spectrum, if 'squareInput=1'). $X^t(f)$ is the spectral magnitude at time t in bin f . The skewness γ_1 is then computed as:

$$\gamma_1 = \frac{1}{\sigma^3 \sum_{\forall f} X^t(f)} \sum_{\forall f} (f - C_S^t)^3 \cdot X^t(f) \quad (3.7)$$

kurtosis = <numeric> [Default: 0]

(1/0=yes/no) enable computation of spectral kurtosis. The kurtosis is an indicator for the peakedness of the spectrum, i.e. if it is rather flat (< 3), follows a normal distribution around the centroid ($= 3$), or has a pronounced peak at the centroid (> 3). The spectral kurtosis is computed from the magnitude spectrum (or power spectrum, if 'squareInput=1'). $X^t(f)$ is the spectral magnitude at time t in bin f . The skewness γ_1 is then computed as:

$$\gamma_1 = \frac{1}{\sigma^4 \sum_{\forall f} X^t(f)} \sum_{\forall f} (f - C_S^t)^4 \cdot X^t(f) \quad (3.8)$$

slope = <numeric> [Default: 0]

(1/0=yes/no) enable computation of the spectral slope m_S , using the following linear regression equation:

$$m_S = \frac{N \left(\sum_{\forall f} f \cdot X^t(f) \right) - \left(\sum_{\forall f} f \right) \cdot \left(\sum_{\forall f} X^t(f) \right)}{N \left(\sum_{\forall f} f^2 \right) - \left(\sum_{\forall f} f \right)^2} \quad (3.9)$$

cTonefilt

Description: This component implements an on-line, sample by sample semi-tone filter bank which can be used as first step for the computation of CHROMA features as a replacement of cTonespec (section 3.3.6). In contrast to cTonespec, this component can compute semi-tone spectra at any rate and resolution, because it uses a continuous time-domain filter-bank instead of windowing and short-term FFT. It expects un-windowed PCM data as input (generated by the cWaveSource (section 3.3.3) component, for example).

Class hierarchy: cSmileComponent \rightarrow cDataProcessor \rightarrow cTonefilt

Configuration options:

reader = <object of type ‘cDataReader’> The configuration of the cDataReader sub-component, which handles the dataMemory interface for data input. See the documentation of ‘cDataReader’ for more information (section 3.3.2).

writer = <object of type ‘cDataWriter’> The configuration of the cDataWriter sub-component, which handles the dataMemory interface for data output. See the documentation of ‘cDataWriter’ for more information (section 3.3.2).

bufferSize = <numeric> [Default: 0]
The buffer size for the output level, in frames (default [0] = same as input level), this option overwrites ‘bufferSize_sec’.

bufferSize_sec = <numeric> [Default: 0]
The buffer size for the output level, in seconds (default [0] = same as input level).

blockSize = <numeric> [Default: 0]
The size of data blocks to process, in frames (this sets both ‘blockSizeR’ and ‘blockSizeW’, and overwrites ‘blockSize_sec’).

blockSizeR = <numeric> [Default: 0]
The size of data blocks to read, in frames (overwrites ‘blockSize’).

blockSizeW = <numeric> [Default: 0]
The size of data blocks to write, in frames (overwrites ‘blockSize’).

blockSize_sec = <numeric> [Default: 0]
The size of data blocks to process, in seconds (this sets both ‘blockSizeR_sec’ and ‘blockSizeW_sec’).

blockSizeR_sec = <numeric> [Default: 0]
The size of data blocks to read, in seconds (overwrites ‘blockSize_sec’).

blockSizeW_sec = <numeric> [Default: 0]
The size of data blocks to write in seconds (overwrites ‘blockSize_sec’).

nameAppend = <string> [Default: ‘(null)’]
A string suffix to append to the input field names (default: empty).

copyInputName = <numeric> [Default: 1]
1 = copy the input name (and optionally append a suffix, see the ‘nameAppend’ option)
0 = discard the input name and use only the ‘nameAppend’ string as new name.

nNotes = <numeric> [Default: 48]
The number of semi-tone filters in the filter bank.

firstNote = <numeric> [Default: 55]
The frequency of the first note in Hz.

decayF0 = <numeric> [Default: 0.9995]
The gliding average decay coefficient for the first note (lowest frequency).

decayFN = <numeric> [Default: 0.998]
The gliding average decay coefficient for the last note (highest frequency) (must be < ‘decayF0’!); decay coefficients for intermediate frequencies will be interpolated linearly from the start and end coefficients.

outputPeriod = <numeric> [Default: 0.1]
 Specifies the period at which to produce output frames, in seconds.

cTonespec

Description: This component computes a semi-tone spectrum from an FFT magnitude spectrum (produced by cFFTMagphase, section 3.3.6).

Class hierarchy: cSmileComponent → cDataProcessor → cVectorProcessor → cTonespec

Configuration options:

reader = <object of type ‘cDataReader’> The configuration of the cDataReader sub-component, which handles the dataMemory interface for data input. See the documentation of ‘cDataReader’ for more information (section 3.3.2).

writer = <object of type ‘cDataWriter’> The configuration of the cDataWriter sub-component, which handles the dataMemory interface for data output. See the documentation of ‘cDataWriter’ for more information (section 3.3.2).

bufferSize = <numeric> [Default: 0]
 The buffer size for the output level, in frames (default [0] = same as input level), this option overwrites ‘bufferSize_sec’.

bufferSize_sec = <numeric> [Default: 0]
 The buffer size for the output level, in seconds (default [0] = same as input level).

blockSize = <numeric> [Default: 0]
 The size of data blocks to process, in frames (this sets both ‘blockSizeR’ and ‘blockSizeW’, and overwrites ‘blockSize_sec’).

blockSizeR = <numeric> [Default: 0]
 The size of data blocks to read, in frames (overwrites ‘blockSize’).

blockSizeW = <numeric> [Default: 0]
 The size of data blocks to write, in frames (overwrites ‘blockSize’).

blockSize_sec = <numeric> [Default: 0]
 The size of data blocks to process, in seconds (this sets both ‘blockSizeR_sec’ and ‘blockSizeW_sec’).

blockSizeR_sec = <numeric> [Default: 0]
 The size of data blocks to read, in seconds (overwrites ‘blockSize_sec’).

blockSizeW_sec = <numeric> [Default: 0]
 The size of data blocks to write in seconds (overwrites ‘blockSize_sec’).

nameAppend = <string> [Default: ‘(null)’]
 A string suffix to append to the input field names (default: empty).

copyInputName = <numeric> [Default: 1]
 1 = copy the input name (and optionally append a suffix, see the ‘nameAppend’ option)
 0 = discard the input name and use only the ‘nameAppend’ string as new name.

processArrayFields = **<numeric>** [Default: 1]
 1 = process each array field as one vector individually (and produce one output for each input array field). Only array fields (i.e. fields with more than one element) are processed if this is set.
 0 = process complete input frame as one vector, ignoring field/element structure.

nOctaves = **<numeric>** [Default: 6]
 The number of octaves (12 semitones) the spectrum should span.

firstNote = **<numeric>** [Default: 55]
 The frequency of the first note (in Hz).

filterType = **<string>** [Default: 'gau']
 The shape of the semitone filter:

tri Triangular

trp Triangular-powered (squared)

gau Gaussian

usePower = **<numeric>** [Default: 0]
 1 = compute the semi-tone spectrum from the power spectrum instead of the magnitudes (= square input values).

dbA = **<numeric>** [Default: 1]
 1 = apply a built-in dB(A) weighting to the linear scale (magnitude or power) spectrum (1/0 = yes/no).

cTransformFFT

Description: This component performs an FFT on a sequence of real values (one frame), the output is the complex domain result of the transform. Use the **cFFTmagphase** component to compute magnitudes and phases from this complex output.

Class hierarchy: **cSmileComponent** → **cDataProcessor** → **cVectorProcessor** → **cTransformFFT**

Configuration options:

reader = **<object of type 'cDataReader'>** The configuration of the **cDataReader** sub-component, which handles the **dataMemory** interface for data input. See the documentation of 'cDataReader' for more information (section 3.3.2).

writer = **<object of type 'cDataWriter'>** The configuration of the **cDataWriter** sub-component, which handles the **dataMemory** interface for data output. See the documentation of 'cDataWriter' for more information (section 3.3.2).

bufferSize = **<numeric>** [Default: 0]
 The buffer size for the output level, in frames (default [0] = same as input level), this option overwrites 'bufferSize_sec'.

bufferSize_sec = **<numeric>** [Default: 0]
 The buffer size for the output level, in seconds (default [0] = same as input level).

blocksize = <numeric> [Default: 0]

The size of data blocks to process, in frames (this sets both ‘blocksizeR’ and ‘blocksizeW’, and overwrites ‘blocksize_sec’).

blocksizeR = <numeric> [Default: 0]

The size of data blocks to read, in frames (overwrites ‘blocksize’).

blocksizeW = <numeric> [Default: 0]

The size of data blocks to write, in frames (overwrites ‘blocksize’).

blocksize_sec = <numeric> [Default: 0]

The size of data blocks to process, in seconds (this sets both ‘blocksizeR_sec’ and ‘blocksizeW_sec’).

blocksizeR_sec = <numeric> [Default: 0]

The size of data blocks to read, in seconds (overwrites ‘blocksize_sec’).

blocksizeW_sec = <numeric> [Default: 0]

The size of data blocks to write in seconds (overwrites ‘blocksize_sec’).

nameAppend = <string> [Default: ‘(null)’]

A string suffix to append to the input field names (default: empty).

copyInputName = <numeric> [Default: 1]

1 = copy the input name (and optionally append a suffix, see the ‘nameAppend’ option)
0 = discard the input name and use only the ‘nameAppend’ string as new name.

processArrayFields = <numeric> [Default: 1]

1 = process each array field as one vector individually (and produce one output for each input array field). Only array fields (i.e. fields with more than one element) are processed if this is set.

0 = process complete input frame as one vector, ignoring field/element structure.

inverse = <numeric> [Default: 0]

1 = perform inverse FFT.

cTurnDetector

Description: Speaker turn detector using data from a cEnergy (section 3.3.6), cVadV1 (section 3.3.6), or cSemaineSpeakerID1 (adaptive VAD, section 3.3.6) to determine speaker turns and identify continuous segments of voice activity. It produces an output field ‘isTurn’ and is capable of sending smile messages indicating the turn start and end, as well as turn status messages.

Class hierarchy: cSmileComponent → cDataProcessor → cTurnDetector

Configuration options:

reader = <object of type ‘cDataReader’> The configuration of the cDataReader sub-component, which handles the dataMemory interface for data input. See the documentation of ‘cDataReader’ for more information (section 3.3.2).

writer = <object of type 'cDataWriter'> The configuration of the cDataWriter sub-component, which handles the dataMemory interface for data output. See the documentation of 'cDataWriter' for more information (section 3.3.2).

bufferSize = <numeric> [Default: 0]
The buffer size for the output level, in frames (default [0] = same as input level), this option overwrites 'bufferSize_sec'.

bufferSize_sec = <numeric> [Default: 0]
The buffer size for the output level, in seconds (default [0] = same as input level).

blockSize = <numeric> [Default: 0]
The size of data blocks to process, in frames (this sets both 'blockSizeR' and 'blockSizeW', and overwrites 'blockSize_sec').

blockSizeR = <numeric> [Default: 0]
The size of data blocks to read, in frames (overwrites 'blockSize').

blockSizeW = <numeric> [Default: 0]
The size of data blocks to write, in frames (overwrites 'blockSize').

blockSize_sec = <numeric> [Default: 0]
The size of data blocks to process, in seconds (this sets both 'blockSizeR_sec' and 'blockSizeW_sec').

blockSizeR_sec = <numeric> [Default: 0]
The size of data blocks to read, in seconds (overwrites 'blockSize_sec').

blockSizeW_sec = <numeric> [Default: 0]
The size of data blocks to write in seconds (overwrites 'blockSize_sec').

nameAppend = <string> [Default: '(null)']
A string suffix to append to the input field names (default: empty).

copyInputName = <numeric> [Default: 1]
1 = copy the input name (and optionally append a suffix, see the 'nameAppend' option)
0 = discard the input name and use only the 'nameAppend' string as new name.

threshold = <numeric> [Default: 0.001]
The silence threshold to use (the default value is for RMS energy, change it to -13.0 for log energy, or 0.55 for VAD input (binary and fuzzy)).

autoThreshold = <numeric> [Default: 0]
1 = automatically adjust threshold (EXPERIMENTAL).

minmaxDecay = <numeric> [Default: 0.9995]
The decay constant used for min/max values in auto-thresholder (a larger value means a slower recovery from loud sounds).

nPre = <numeric> [Default: 10]
The number of frames which must be greater than the threshold in series, until a turn start is detected.

nPost = <numeric> [Default: 20]

The number of frames which must be smaller than the threshold in series until a turn end is detected.

useRMS = <numeric> [Default: 1]

1 = the provided energy field in the input is RMS energy instead of log energy.

readVad = <numeric> [Default: 0]

1 = use the result ('vadBin') from the cVadV1 or cSemaineSpeakerID component instead of reading frame rms/log energy (the threshold is set to 0.55 if this option is enabled).

idx = <numeric> [Default: -1]

The index of the RMS or LOG energy field to use (-1 to automatically find the field).

messageRecp = <string> [Default: '(null)']

The (cWinToVecProcessor type) component(s) to send 'frameTime' messages to (use , to separate multiple recipients), leave blank (NULL) to not send any messages. The messages will be sent at the turn end and (optionally) during the turn at fixed intervals configured by the 'msgInterval' parameter (if it is not 0).

msgInterval = <numeric> [Default: 0]

Interval at which to send 'frameTime' messages during an ongoing turn. Set to 0 to disable sending of intra turn messages.

eventRecp = <string> [Default: '(null)']

The component(s) to send 'turnStart/turnEnd' messages to (use , to separate multiple recipients), leave blank (NULL) to not send any messages

statusRecp = <string> [Default: '(null)']

The component(s) to send 'turnSpeakingStatus' messages to (use , to separate multiple recipients), leave blank (NULL) to not send any messages

maxTurnLength = <numeric> [Default: 0]

The maximum turn length in seconds (≤ 0 = infinite). A turn end will be favoured by temporarily reducing the 'nPost' setting to 1 after 'maxTurnLength'.

maxTurnLengthGrace = <numeric> [Default: 1]

The grace period to grant, after 'maxTurnLength' is reached (in seconds). After a turn length of maxTurnLength + maxTurnLengthGrace an immediate turn end will be forced.

debug = <numeric> [Default: 4]

The log-level to show some turn detector specific debug messages on.

cVadV1

Description: A voice activity detector based on Line-Spectral-Frequencies (cLsp, section 3.3.6), Mel spectra ((cMelspec, section 3.3.6), and energy. Fuzzy scores related to the deviation from the observed long-term mean values are computed from these input fields. This component requires input of the following type in the following order: **MelSpec**; **lsf**; **energy**. An example excerpt from an example configuration file is shown here:

```
***** example config for cVadV1 *****
```

```
[enV:cEnergy]
```

```

reader.dmLevel=frame
writer.dmLevel=energy
nameAppend=energy
rms=1
log=1

[lpc:cLpc]
reader.dmLevel=frames
saveRefCoeff=0
writer.dmLevel=lpc
p=10

[mspecV:cMelspec]
reader.dmLevel=fftmagnitude
writer.dmLevel=mspec
htkcompatible = 0
usePower = 0
nBands = 14
lofreq = 50
hifreq = 4000

[lsp:cLsp]
reader.dmLevel=lpc
writer.dmLevel=lsp

[vad:cVadV1]
reader.dmLevel=mspec;lsp;energy
writer.dmLevel=vad11
writer.levelconf.noHang=1
debug=\cm[vaddebug{0}:1=debug vad]
threshold=\cm[threshold{-13}:VAD threshold]
disableDynamicVAD=\cm[disableDynamicVAD{0}:disable dynamic threshold
    vad, instead use energy based vad only, the energy threshold can be
    set via the 'threshold' option]
;threshold=\cm[threshold{-13.0}:VAD energy threshold, minimum energy
    for dynamic vad, can be very small, it is used only as a backup;
    real threshold if disableDynamicVAD is set, in that case you should
    set the threshold to approx.]

```

Note: this is only an example configuration, you must adjust this to your setup and add the components the global componentInstance section.

Class hierarchy: cSmileComponent → cDataProcessor → cVadV1

Configuration options:

reader = <object of type 'cDataReader'> The configuration of the cDataReader sub-component, which handles the dataMemory interface for data input. See the documentation of 'cDataReader' for more information (section 3.3.2).

writer = <object of type 'cDataWriter'> The configuration of the cDataWriter sub-component, which handles the dataMemory interface for data output. See the documentation of 'cDataWriter' for more information (section 3.3.2).

bufferSize = <numeric> [Default: 0]

The buffer size for the output level, in frames (default [0] = same as input level), this

option overwrites ‘buffer_size_sec’.

buffer_size_sec = <numeric> [Default: 0]

The buffer size for the output level, in seconds (default [0] = same as input level).

blocksize = <numeric> [Default: 0]

The size of data blocks to process, in frames (this sets both ‘blocksizeR’ and ‘blocksizeW’, and overwrites ‘blocksize_sec’).

blocksizeR = <numeric> [Default: 0]

The size of data blocks to read, in frames (overwrites ‘blocksize’).

blocksizeW = <numeric> [Default: 0]

The size of data blocks to write, in frames (overwrites ‘blocksize’).

blocksize_sec = <numeric> [Default: 0]

The size of data blocks to process, in seconds (this sets both ‘blocksizeR_sec’ and ‘blocksizeW_sec’).

blocksizeR_sec = <numeric> [Default: 0]

The size of data blocks to read, in seconds (overwrites ‘blocksize_sec’).

blocksizeW_sec = <numeric> [Default: 0]

The size of data blocks to write in seconds (overwrites ‘blocksize_sec’).

nameAppend = <string> [Default: ‘(null)’]

A string suffix to append to the input field names (default: empty).

copyInputName = <numeric> [Default: 1]

1 = copy the input name (and optionally append a suffix, see the ‘nameAppend’ option)

0 = discard the input name and use only the ‘nameAppend’ string as new name.

threshold = <numeric> [Default: -13]

The minimum rms/log energy threshold to use (or the actual rms energy threshold, if disableDynamicVAD==1)

disableDynamicVAD = <numeric> [Default: 0]

1/0 = yes/no, whether dynamic VAD is disabled (default is enabled).

debug = <numeric> [Default: 0]

1/0 enable/disable VAD debug output. Use for testing and debugging only!

cValbasedSelector

Description: This component copies only those frames from the input to the output that match a certain threshold criterion, i.e. where a specified value N exceeds a certain threshold.

Class hierarchy: cSmileComponent → cDataProcessor → cValbasedSelector

Configuration options:

reader = <object of type ‘cDataReader’> The configuration of the cDataReader sub-component, which handles the dataMemory interface for data input. See the documentation of ‘cDataReader’ for more information (section 3.3.2).

writer = <object of type ‘cDataWriter’> The configuration of the cDataWriter sub-component, which handles the dataMemory interface for data output. See the documentation of ‘cDataWriter’ for more information (section 3.3.2).

bufferSize = <numeric> [Default: 0]

The buffer size for the output level, in frames (default [0] = same as input level), this option overwrites ‘bufferSize_sec’.

bufferSize_sec = <numeric> [Default: 0]

The buffer size for the output level, in seconds (default [0] = same as input level).

blockSize = <numeric> [Default: 0]

The size of data blocks to process, in frames (this sets both ‘blockSizeR’ and ‘blockSizeW’, and overwrites ‘blockSize_sec’).

blockSizeR = <numeric> [Default: 0]

The size of data blocks to read, in frames (overwrites ‘blockSize’).

blockSizeW = <numeric> [Default: 0]

The size of data blocks to write, in frames (overwrites ‘blockSize’).

blockSize_sec = <numeric> [Default: 0]

The size of data blocks to process, in seconds (this sets both ‘blockSizeR_sec’ and ‘blockSizeW_sec’).

blockSizeR_sec = <numeric> [Default: 0]

The size of data blocks to read, in seconds (overwrites ‘blockSize_sec’).

blockSizeW_sec = <numeric> [Default: 0]

The size of data blocks to write in seconds (overwrites ‘blockSize_sec’).

nameAppend = <string> [Default: ‘(null)’]

A string suffix to append to the input field names (default: empty).

copyInputName = <numeric> [Default: 1]

1 = copy the input name (and optionally append a suffix, see the ‘nameAppend’ option)
0 = discard the input name and use only the ‘nameAppend’ string as new name.

threshold = <numeric> [Default: 1]

Threshold for selection (see also the ‘invert’ option).

idx = <numeric> [Default: 0]

The index of the element to base the selection decision on. Currently only *one* element is supported, *no* vector based thresholds etc. are possible.

invert = <numeric> [Default: 0]

1 = output the frame when element[idx] < threshold
0 = output the frame if element[idx] => ‘threshold’.

allowEqual = <numeric> [Default: 0]

If this option is set to 1, also output the frame when element[idx] = ‘threshold’.

removeIdx = <numeric> [Default: 0]

1 = remove field element[idx] in the output vector
0 = keep field element[idx] in the output vector

cVecGlMean

Description: This is the old component for cepstral mean subtraction. It has been replaced by the cVectorMVN (section 3.3.6) component, which is more powerful. It computes gliding mean of input vectors and subtracts it from the vectors (use this component for Cepstral Mean Subtraction).

Class hierarchy: cSmileComponent → cDataProcessor → cVectorProcessor → cVecGlMean

Configuration options:

reader = <object of type ‘cDataReader’> The configuration of the cDataReader sub-component, which handles the dataMemory interface for data input. See the documentation of ‘cDataReader’ for more information (section 3.3.2).

writer = <object of type ‘cDataWriter’> The configuration of the cDataWriter sub-component, which handles the dataMemory interface for data output. See the documentation of ‘cDataWriter’ for more information (section 3.3.2).

bufferize = <numeric> [Default: 0]
The buffer size for the output level, in frames (default [0] = same as input level), this option overwrites ‘bufferize_sec’.

bufferize_sec = <numeric> [Default: 0]
The buffer size for the output level, in seconds (default [0] = same as input level).

blocksize = <numeric> [Default: 0]
The size of data blocks to process, in frames (this sets both ‘blocksizeR’ and ‘blocksizeW’, and overwrites ‘blocksize_sec’).

blocksizeR = <numeric> [Default: 0]
The size of data blocks to read, in frames (overwrites ‘blocksize’).

blocksizeW = <numeric> [Default: 0]
The size of data blocks to write, in frames (overwrites ‘blocksize’).

blocksize_sec = <numeric> [Default: 0]
The size of data blocks to process, in seconds (this sets both ‘blocksizeR_sec’ and ‘blocksizeW_sec’).

blocksizeR_sec = <numeric> [Default: 0]
The size of data blocks to read, in seconds (overwrites ‘blocksize_sec’).

blocksizeW_sec = <numeric> [Default: 0]
The size of data blocks to write in seconds (overwrites ‘blocksize_sec’).

nameAppend = <string> [Default: ‘zeromean’]
A string suffix to append to the input field names (default: empty).

copyInputName = <numeric> [Default: 1]
1 = copy the input name (and optionally append a suffix, see the ‘nameAppend’ option)
0 = discard the input name and use only the ‘nameAppend’ string as new name.

processArrayFields = <numeric> [Default: 1]
1 = process each array field as one vector individually (and produce one output for each

input array field). Only array fields (i.e. fields with more than one element) are processed if this is set.

0 = process complete input frame as one vector, ignoring field/element structure.

initFile = <string> [Default: '(null)']

Filename of the file from which to load the initial mean values.

updateMethod = <string> [Default: 'iavg']

Specifies how to incrementally update the mean values. One of the following:

null do not perform any updates, use only initial values.

exp exponential: $\mu_1 = \alpha \cdot \mu_0 + (1 - \alpha) \cdot x$

fix compute the mean over a buffer of fixed length (see 'fixedBuffer' option)

avg moving average method with weighted fixed initial values

iavg moving average method with weighted variable (updated per turn) initial values

Note: if 'resetOnTurn' = 0 then 'avg' and 'iavg' methods are identical.

alpha = <numeric> [Default: 0.995]

The weighting factor α in gliding average computation.

fixedBuffer = <numeric> [Default: 5]

The size of the fixed length buffer (in seconds).

turnOnlyUpdate = <numeric> [Default: 1]

1 = perform mean update only during turns (works for all methods).

invertTurn = <numeric> [Default: 0]

1 = invert turn state (i.e. this changes a 'turnOnly' option into "not turn" option).

resetOnTurn = <numeric> [Default: 0]

1 = reset mean values at the beginning of each new turn.

turnOnlyNormalise = <numeric> [Default: 0]

1 = normalise only during turns (in between, data will pass through unmodified).

htkcompatible = <numeric> [Default: 0]

A flag that indicates (if set to 1) whether the last coefficient in 'initFile' is loaded into means[0]. Use this when reading htk-compatible cmn init files, and *not* using htk-compatible MFCCs or PLPs.

turnStartMessage = <string> [Default: 'turnStart']

You can use this option to define a custom message name for turn start messages this component listens for, e.g. if you want to use voice activity start/end messages instead.

turnEndMessage = <string> [Default: 'turnEnd']

You can use this option to define a custom message name for turn end messages this component listens for, e.g. if you want to use voice activity start/end messages instead

cVectorConcat

Description: This component is similar to cBuffer (section 3.3.6), however it is optimised for concatenating frames from multiple levels, thereby reading frame by frame. Reading multiple frames at once (as is supported by cBuffer) is not supported.

Class hierarchy: cSmileComponent → cDataProcessor → cVectorProcessor → cVecGlMean

Configuration options:

reader = <object of type 'cDataReader'> The configuration of the cDataReader sub-component, which handles the dataMemory interface for data input. See the documentation of 'cDataReader' for more information (section 3.3.2).

writer = <object of type 'cDataWriter'> The configuration of the cDataWriter sub-component, which handles the dataMemory interface for data output. See the documentation of 'cDataWriter' for more information (section 3.3.2).

bufferSize = <numeric> [Default: 0]
The buffer size for the output level, in frames (default [0] = same as input level), this option overwrites 'bufferSize_sec'.

bufferSize_sec = <numeric> [Default: 0]
The buffer size for the output level, in seconds (default [0] = same as input level).

blockSize = <numeric> [Default: 0]
The size of data blocks to process, in frames (this sets both 'blockSizeR' and 'blockSizeW', and overwrites 'blockSize_sec').

blockSizeR = <numeric> [Default: 0]
The size of data blocks to read, in frames (overwrites 'blockSize').

blockSizeW = <numeric> [Default: 0]
The size of data blocks to write, in frames (overwrites 'blockSize').

blockSize_sec = <numeric> [Default: 0]
The size of data blocks to process, in seconds (this sets both 'blockSizeR_sec' and 'blockSizeW_sec').

blockSizeR_sec = <numeric> [Default: 0]
The size of data blocks to read, in seconds (overwrites 'blockSize_sec').

blockSizeW_sec = <numeric> [Default: 0]
The size of data blocks to write in seconds (overwrites 'blockSize_sec').

nameAppend = <string> [Default: '(null)']
A string suffix to append to the input field names (default: empty).

copyInputName = <numeric> [Default: 1]
1 = copy the input name (and optionally append a suffix, see the 'nameAppend' option)
0 = discard the input name and use only the 'nameAppend' string as new name.

processArrayFields = <numeric> [Default: 1]
1 = process each array field as one vector individually (and produce one output for each input array field). Only array fields (i.e. fields with more than one element) are processed if this is set.
0 = process complete input frame as one vector, ignoring field/element structure.

cVectorHEQ

Description: This component extends the base classes `cVectorTransform` ((section 3.3.6)) and `cVectorMVN` (section 3.3.6), and implements histogram equalisation. This technique is used for noise robust speech recognition.

Class hierarchy: `cSmileComponent` \rightarrow `cDataProcessor` \rightarrow `cVectorProcessor` \rightarrow `cVectorTransform` \rightarrow `cVectorMVN` \rightarrow `cVectorHEQ`

Configuration options:

reader = **<object of type ‘cDataReader’>** The configuration of the `cDataReader` sub-component, which handles the `dataMemory` interface for data input. See the documentation of ‘`cDataReader`’ for more information (section 3.3.2).

writer = **<object of type ‘cDataWriter’>** The configuration of the `cDataWriter` sub-component, which handles the `dataMemory` interface for data output. See the documentation of ‘`cDataWriter`’ for more information (section 3.3.2).

bufferSize = **<numeric>** [Default: 0]
The buffer size for the output level, in frames (default [0] = same as input level), this option overwrites ‘`bufferSize_sec`’.

bufferSize_sec = **<numeric>** [Default: 0]
The buffer size for the output level, in seconds (default [0] = same as input level).

blockSize = **<numeric>** [Default: 0]
The size of data blocks to process, in frames (this sets both ‘`blockSizeR`’ and ‘`blockSizeW`’, and overwrites ‘`blockSize_sec`’).

blockSizeR = **<numeric>** [Default: 0]
The size of data blocks to read, in frames (overwrites ‘`blockSize`’).

blockSizeW = **<numeric>** [Default: 0]
The size of data blocks to write, in frames (overwrites ‘`blockSize`’).

blockSize_sec = **<numeric>** [Default: 0]
The size of data blocks to process, in seconds (this sets both ‘`blockSizeR_sec`’ and ‘`blockSizeW_sec`’).

blockSizeR_sec = **<numeric>** [Default: 0]
The size of data blocks to read, in seconds (overwrites ‘`blockSize_sec`’).

blockSizeW_sec = **<numeric>** [Default: 0]
The size of data blocks to write in seconds (overwrites ‘`blockSize_sec`’).

nameAppend = **<string>** [Default: ‘null’]
A string suffix to append to the input field names (default: empty).

copyInputName = **<numeric>** [Default: 1]
1 = copy the input name (and optionally append a suffix, see the ‘`nameAppend`’ option)
0 = discard the input name and use only the ‘`nameAppend`’ string as new name.

processArrayFields = **<numeric>** [Default: 1]
1 = process each array field as one vector individually (and produce one output for each

input array field). Only array fields (i.e. fields with more than one element) are processed if this is set.

0 = process complete input frame as one vector, ignoring field/element structure.

mode = <string> [Default: 'analysis']

This sets the mode of operation:

an(alysis) analyse all incoming data and compute transform for later use. No transformation (no modification) of data is performed and no data is output (the tick() function of the cVectorTransform component always returns 0 when in analysis mode!).

tr(ansformation) apply a pre-computed transform loaded from the file 'initFile'. The transform is applied to the input data, however no on-line adaptation of the transform is performed.

in(cremental) use the transform loaded from the file 'initFile' as initial transform (if 'initFile' is not empty). The transform is incrementally update from new input data using the update method specified via the 'updateMethod' option.

initFile = <string> [Default: '(null)']

The file to load the (initial) transformation data from (see vectorTransform.cpp for documentation of the file format)

saveFile = <string> [Default: '(null)']

The file to save transformation data to. The file is always saved in the new smile binary transformation data format (see the documentation of the cVectorTransform base class in section 3.3.6) for documentation of the file format).

updateMethod = <string> [Default: 'buf']

Specifies how to incrementally update the transform. One of the following strings:

fix/buf the transform is computed over a history buffer of fixed length

avg cumulative average method (sum of all input values so far, normalised by the count of all input values so far) with weighted fixed initial values.

alpha = <numeric> [Default: 0.995]

The weighting factor α for exponential transform update.

weight = <numeric> [Default: 100]

The weighting factor for the 'avg' mean update, i.e. the factor the initial transform parameters are weighted by when building the cumulative average.

fixedBuffer = <numeric> [Default: 5]

The size of the fixed length buffer (in seconds) for the fixed buffer update method 'fix' or 'buf'.

turnOnlyUpdate = <numeric> [Default: 0]

1 = perform transform update only during turns (between turnStart and turnEnd messages) (works for all methods).

invertTurn = <numeric> [Default: 0]

1 = invert the turn state (i.e. this changes a 'turnOnly' option into 'not turn' option).

- resetOnTurn** = **<numeric>** [Default: 0]
 1 = reset transform values at the beginning of each new turn (only in mode ‘analysis’ and ‘incremental’)
- turnOnlyNormalise** = **<numeric>** [Default: 0]
 1 = apply the transform only to turns, in between data will pass through unmodified. ‘invertTurn’ will also invert this option.
- turnOnlyOutput** = **<numeric>** [Default: 0]
 1 = output data to write level only during a turn (this will implicitly set turnOnlyNormalise = 1). ‘invertTurn’ will also invert this option.
- htkcompatible** = **<numeric>** [Default: 0]
 A flag that indicates (if set to 1) whether the last coefficient in ‘initFile’ is loaded into means[0] (use this only when reading htk-compatible CMN init files, and *not* using htk-compatible MFCCs or PLPs)
- turnStartMessage** = **<string>** [Default: ‘turnStart’]
 You can use this option to define a custom message name for the turn start message, i.e. if you want to use voice activity start/end messages instead.
- turnEndMessage** = **<string>** [Default: ‘turnEnd’]
 You can use this option to define a custom message name for the turn end message, i.e. if you want to use voice activity start/end messages instead.
- meanEnable** = **<numeric>** [Default: 1]
 1 = enable normalisation to 0 mean.
- stdEnable** = **<numeric>** [Default: 1]
 1 = enable standardisation to standard deviation 1.
- normEnable** = **<numeric>** [Default: 0]
 1 = enable normalisation (scaling) of values to the range -1 to +1 (this can *not* be used in conjunction with ‘stdEnable=1’).
- numIntervals** = **<numeric>** [Default: 1000]
 The number of discrete bins to use during histogram computation (a higher number leads to a more fine grained histogram).
- histRange** = **<numeric>** [Default: 4]
 The range of the histogram in terms of a factor applied to the standard deviation (range [mean-histRange*stddev; mean+histRange*stddev]). Please note, that this can lead to a changing range, if the standard deviation changes. This is not yet well supported since it requires histogram rescaling. We recommend to initialise with an MVN file, and disable on-line updates of the standard deviation, or use the option ‘fixedRange=1’.
- meanTolerance** = **<numeric>** [Default: 0.01]
 The percentage of allowed mean drift before the histogram range is adjusted and the histogram is rescaled.

rangeTolerance = <numeric> [Default: 0.01]

The percentage of allowed range drift before the histogram range is adjusted and the histogram is rescaled.

fixedRange = <numeric> [Default: 0]

1 = use a fixed range from the MVN initialisation file and do *not* update the range when the standard deviation changes.

noHeqInit = <numeric> [Default: 0]

1 = do not use HEQ data from the init file (this is to be used in conjunction with ‘fixedRange=1’)

stdRange = <numeric> [Default: 0]

1 = use a fixed absolute histogram range. This implies ‘fixedRange=1’, but does not use the MVN data from the init file. Instead, the init data is generated as mean 0 and standard deviation 1, the histogram range is then given by the absolute value of the ‘histRange’ parameter as: -histRange to +histRange.

cVectorMVN

Description: This component extends the base class cVectorTransform (section 3.3.6) and implements mean/variance normalisation. You can use this component to perform on-line cepstral mean normalisation. See cFullinputMean (section 3.3.6) for off-line cepstral mean normalisation.

Class hierarchy: cSmileComponent → cDataProcessor → cVectorProcessor → cVectorTransform → cVectorMVN

Configuration options:

reader = <object of type ‘cDataReader’> The configuration of the cDataReader sub-component, which handles the dataMemory interface for data input. See the documentation of ‘cDataReader’ for more information (section 3.3.2).

writer = <object of type ‘cDataWriter’> The configuration of the cDataWriter sub-component, which handles the dataMemory interface for data output. See the documentation of ‘cDataWriter’ for more information (section 3.3.2).

bufferSize = <numeric> [Default: 0]

The buffer size for the output level, in frames (default [0] = same as input level), this option overwrites ‘bufferSize_sec’.

bufferSize_sec = <numeric> [Default: 0]

The buffer size for the output level, in seconds (default [0] = same as input level).

blockSize = <numeric> [Default: 0]

The size of data blocks to process, in frames (this sets both ‘blockSizeR’ and ‘blockSizeW’, and overwrites ‘blockSize_sec’).

blockSizeR = <numeric> [Default: 0]

The size of data blocks to read, in frames (overwrites ‘blockSize’).

blockSizeW = <numeric> [Default: 0]

The size of data blocks to write, in frames (overwrites ‘blockSize’).

blocksize_sec = <numeric> [Default: 0]

The size of data blocks to process, in seconds (this sets both ‘blocksizeR_sec’ and ‘blocksizeW_sec’).

blocksizeR_sec = <numeric> [Default: 0]

The size of data blocks to read, in seconds (overwrites ‘blocksize_sec’).

blocksizeW_sec = <numeric> [Default: 0]

The size of data blocks to write in seconds (overwrites ‘blocksize_sec’).

nameAppend = <string> [Default: ‘null’]

A string suffix to append to the input field names (default: empty).

copyInputName = <numeric> [Default: 1]

1 = copy the input name (and optionally append a suffix, see the ‘nameAppend’ option)
0 = discard the input name and use only the ‘nameAppend’ string as new name.

processArrayFields = <numeric> [Default: 1]

1 = process each array field as one vector individually (and produce one output for each input array field). Only array fields (i.e. fields with more than one element) are processed if this is set.

0 = process complete input frame as one vector, ignoring field/element structure.

mode = <string> [Default: ‘analysis’]

This sets the mode of operation:

an(alysis) analyse all incoming data and compute transform for later use. No transformation (no modification) of data is performed and no data is output (the tick() function of the cVectorTransform component always returns 0 when in analysis mode!).

tr(ansformation) apply a pre-computed transform loaded from the file ‘initFile’. The transform is applied to the input data, however no on-line adaptation of the transform is performed.

in(cremental) use the transform loaded from the file ‘initFile’ as initial transform (if ‘initFile’ is not empty). The transform is incrementally update from new input data using the update method specified via the ‘updateMethod’ option.

initFile = <string> [Default: ‘(null)’]

The file to load the (initial) transformation data from (see vectorTransform.cpp for documentation of the file format)

saveFile = <string> [Default: ‘(null)’]

The file to save transformation data to. The file is always saved in the new smile binary transformation data format (see the documentation of the cVectorTransform base class in section 3.3.6) for documentation of the file format).

updateMethod = <string> [Default: ‘buf’]

Specifies how to incrementally update the transform. One of the following strings:

fix/buf the transform is computed over a history buffer of fixed length

avg cumulative average method (sum of all input values so far, normalised by the count of all input values so far) with weighted fixed initial values.

alpha = <numeric> [Default: 0.995]

The weighting factor α for exponential transform update.

weight = <numeric> [Default: 100]

The weighting factor for the ‘avg’ mean update, i.e. the factor the initial transform parameters are weighted by when building the cumulative average.

fixedBuffer = <numeric> [Default: 5]

The size of the fixed length buffer (in seconds) for the fixed buffer update method ‘fix’ or ‘buf’.

turnOnlyUpdate = <numeric> [Default: 0]

1 = perform transform update only during turns (between turnStart and turnEnd messages) (works for all methods).

invertTurn = <numeric> [Default: 0]

1 = invert the turn state (i.e. this changes a ‘turnOnly’ option into ‘not turn’ option).

resetOnTurn = <numeric> [Default: 0]

1 = reset transform values at the beginning of each new turn (only in mode ‘analysis’ and ‘incremental’)

turnOnlyNormalise = <numeric> [Default: 0]

1 = apply the transform only to turns, in between data will pass through unmodified. ‘invertTurn’ will also invert this option.

turnOnlyOutput = <numeric> [Default: 0]

1 = output data to write level only during a turn (this will implicitly set turnOnlyNormalise = 1). ‘invertTurn’ will also invert this option.

htkcompatible = <numeric> [Default: 0]

A flag that indicates (if set to 1) whether the last coefficient in ‘initFile’ is loaded into means[0] (use this only when reading htk-compatible CMN init files, and *not* using htk-compatible MFCCs or PLPs)

turnStartMessage = <string> [Default: ‘turnStart’]

You can use this option to define a custom message name for the turn start message, i.e. if you want to use voice activity start/end messages instead.

turnEndMessage = <string> [Default: ‘turnEnd’]

You can use this option to define a custom message name for the turn end message, i.e. if you want to use voice activity start/end messages instead.

meanEnable = <numeric> [Default: 1]

1 = enable normalisation to 0 mean.

stdEnable = <numeric> [Default: 1]

1 = enable standardisation to standard deviation 1.

normEnable = <numeric> [Default: 0]

1 = enable normalisation (scaling) of values to the range -1 to +1 (this can *not* be used in conjunction with ‘stdEnable=1’).

cVectorTransform

This section describes the functionality implemented in the cVectorTransform class. The class itself cannot be instantiated as a component, thus, this section is not a component reference, as the other sections. However, important concepts, such as transform file formats are described here, which are common to components that base on cVectorTransform, such as cVectorMVN (section 3.3.6) and cVectorHEQ (section 3.3.6).

This base class provides support for

- saving computed data at the end of processing (level or custom)
- saving computed data continuously (level or custom)
- applying pre-computed transformation data loaded from file, w/o adaptation
- applying pre-computed transformation data loaded from file, with on-line adaptation
- applying purely on-line adaptive transformation

The class can split the input into segments, which are dynamically controlled by ‘turnStart’ and ‘turnEnd’ smile messages received from other components. Thereby, different modes of operation are distinguished:

- pure analysis: the transform data is computed, no output is generated (except for the transformation data (coefficients, etc.) itself, if it is output to a level). Computation of the transform data is possible for the full input or on a per segment base.
- pure transform: pre-computed transform data is applied, no transform data update is performed
- transform with online adaptation: combines a) and b)
in this mode the output of the transform coefficients to a level is *not* possible (this is only possible in analysis mode).

Transform data can be saved to files and restored from files. Thereby three simple file formats are supported: HTK’s CMN text-based formats, a simple smile binary file format (deprecated), and a new smile binary transform data format. Transform data will always be saved in the new smile binary transform data format. When loading a transform, the file format will be auto-determined according to the following rules. The first 4 bytes are checked for the following tokens (in the given order):

1. magic ID: 0xEE 0x11 0x11 0x00 → new smile binary format
2. HTK Header: < M E A → HTK text-based format
3. everything else → deprecated smile binary format

The three file formats are described in the following:

- **HTK CMN Format** (text-based)

```
<MEAN> N
Val1 Val2 Val3 ...
```

Thereby, N is the number of values in the vector. The values are listed in the second line, separated by a single spaces. Please make sure the <MEAN> is not preceded by whitespaces.

- **deprecated, simple binary 2xN matrix data format:** Dimensions ($2*N \times \text{double}$):
 $N \times \text{double}$ = means,
 $N \times \text{double}$ = stddevs
The vector size N is computed as half of the file size and is thus not contained in the file.
- **advanced binary smile transformation format:**
File header:
Magic (int32) (content: 0xEE 0x11 0x11 0x00)
No. of vectors (int32)
No. of groups (int32)
No. of timeunits (int32)
Vector size (int32)
No. of userData fields (int32)
TypeID (int32)
Reserved (16 byte)
User data: 'No. of userData' \times (double, 8 byte)
Matrix data: 'No. of vectors' \times 'Vector size' \times (double, 8 byte)

cVectorPreemphasis

Description: This component performs per frame pre-emphasis without an inter-frame state memory. This is the way HTK does pre-emphasis. See the cPreemphasis (section 3.3.6) component for implementation details and continuous pre-emphasis filtering.

Class hierarchy: cSmileComponent \rightarrow cDataProcessor \rightarrow cVectorProcessor \rightarrow cVectorPreemphasis

Configuration options:

reader = <object of type 'cDataReader'> The configuration of the cDataReader sub-component, which handles the dataMemory interface for data input. See the documentation of 'cDataReader' for more information (section 3.3.2).

writer = <object of type 'cDataWriter'> The configuration of the cDataWriter sub-component, which handles the dataMemory interface for data output. See the documentation of 'cDataWriter' for more information (section 3.3.2).

bufferSize = <numeric> [Default: 0]
The buffer size for the output level, in frames (default [0] = same as input level), this option overwrites 'bufferSize_sec'.

bufferSize_sec = <numeric> [Default: 0]
The buffer size for the output level, in seconds (default [0] = same as input level).

blockSize = <numeric> [Default: 0]
The size of data blocks to process, in frames (this sets both 'blockSizeR' and 'blockSizeW', and overwrites 'blockSize_sec').

blockSizeR = <numeric> [Default: 0]
The size of data blocks to read, in frames (overwrites 'blockSize').

blocksizeW = <numeric> [Default: 0]

The size of data blocks to write, in frames (overwrites ‘blocksize’).

blocksize_sec = <numeric> [Default: 0]

The size of data blocks to process, in seconds (this sets both ‘blocksizeR_sec’ and ‘blocksizeW_sec’).

blocksizeR_sec = <numeric> [Default: 0]

The size of data blocks to read, in seconds (overwrites ‘blocksize_sec’).

blocksizeW_sec = <numeric> [Default: 0]

The size of data blocks to write in seconds (overwrites ‘blocksize_sec’).

nameAppend = <string> [Default: ‘null’]

A string suffix to append to the input field names (default: empty).

copyInputName = <numeric> [Default: 1]

1 = copy the input name (and optionally append a suffix, see the ‘nameAppend’ option)

0 = discard the input name and use only the ‘nameAppend’ string as new name.

processArrayFields = <numeric> [Default: 1]

1 = process each array field as one vector individually (and produce one output for each input array field). Only array fields (i.e. fields with more than one element) are processed if this is set.

0 = process complete input frame as one vector, ignoring field/element structure.

k = <numeric> [Default: 0.97]

The pre-emphasis coefficient k in $y[n] = x[n] - k \cdot x[n - 1]$.

f = <numeric> [Default: 0]

The pre-emphasis frequency f in Hz, which can be used to compute the filter coefficient

k : $k = \exp\left(-2\pi \cdot \frac{f}{\text{samplingFreq.}}\right)$ (if set, f will override k !)

de = <numeric> [Default: 0]

1 = perform de-emphasis instead of pre-emphasis (i.e. $y[n] = x[n] + k \cdot x[n - 1]$)

cVectorOperation

Description: This component performs elementary operations on vectors (i.e. basically everything that does not require history or context, everything that can be performed on single vectors w/o external data (except for constant parameters, etc.)). See the ‘operation’ option below for currently supported operations.

Class hierarchy: cSmileComponent → cDataProcessor → cVectorProcessor → cVectorOperation

Configuration options:

reader = <object of type ‘cDataReader’> The configuration of the cDataReader sub-component, which handles the dataMemory interface for data input. See the documentation of ‘cDataReader’ for more information (section 3.3.2).

writer = <object of type 'cDataWriter'> The configuration of the cDataWriter sub-component, which handles the dataMemory interface for data output. See the documentation of 'cDataWriter' for more information (section 3.3.2).

bufferize = <numeric> [Default: 0]

The buffer size for the output level, in frames (default [0] = same as input level), this option overwrites 'bufferize_sec'.

bufferize_sec = <numeric> [Default: 0]

The buffer size for the output level, in seconds (default [0] = same as input level).

blocksize = <numeric> [Default: 0]

The size of data blocks to process, in frames (this sets both 'blocksizeR' and 'blocksizeW', and overwrites 'blocksize_sec').

blocksizeR = <numeric> [Default: 0]

The size of data blocks to read, in frames (overwrites 'blocksize').

blocksizeW = <numeric> [Default: 0]

The size of data blocks to write, in frames (overwrites 'blocksize').

blocksize_sec = <numeric> [Default: 0]

The size of data blocks to process, in seconds (this sets both 'blocksizeR_sec' and 'blocksizeW_sec').

blocksizeR_sec = <numeric> [Default: 0]

The size of data blocks to read, in seconds (overwrites 'blocksize_sec').

blocksizeW_sec = <numeric> [Default: 0]

The size of data blocks to write in seconds (overwrites 'blocksize_sec').

nameAppend = <string> [Default: 'null']

A string suffix to append to the input field names (default: empty).

copyInputName = <numeric> [Default: 1]

1 = copy the input name (and optionally append a suffix, see the 'nameAppend' option)
0 = discard the input name and use only the 'nameAppend' string as new name.

processArrayFields = <numeric> [Default: 1]

1 = process each array field as one vector individually (and produce one output for each input array field). Only array fields (i.e. fields with more than one element) are processed if this is set.

0 = process complete input frame as one vector, ignoring field/element structure.

operation = <string> [Default: 'norm']

A string which specifies the type of operation to perform. The following tokens are valid and indicate the currently supported operations:

norm normalise vector length (euclidean norm, L2) to 1

mul multiply the elements in the input vector by 'param1'

add add 'param1' to each element of the input vector

log compute the natural logarithm of each element

lgA compute the base-‘param1’ logarithm of the elements
nl1 normalise the vector sum (L1 norm) to 1
sqr compute the square root of the elements in the vector
pow take the elements to the power of ‘param1’
exp raise ‘param1’ to the power of the vector elements
ee raise the base e to the power of the vector elements
abs take the absolute value of each element

To combine multiple operations in series, you must use multiple `cVectorOperation` components.

param1 = <numeric> [Default: 1]
 parameter 1, see ‘operation’ option for details.

param2 = <numeric> [Default: 1]
 parameter 2, see ‘operation’ option for details. (currently unused)

logfloor = <numeric> [Default: 1e-07]
 Floor value for `log()` function arguments.

powOnlyPos = <numeric> [Default: 0]
 If ‘operation’ = ‘pow’, do not take negative values to the power of ‘param1’, instead, output 0. This is necessary to avoid ‘Not-a-number’ values if the exponent is rational and the base is negative.

cWeightedDiff

Description: This component computes a weighted and smoothed differential by considering the change of the current value wrt. a short term average window (see [?] for a description).

Class hierarchy: `cSmileComponent` → `cDataProcessor` → `cWindowProcessor` → `cWeightedDiff`

Configuration options:

reader = <object of type ‘cDataReader’> The configuration of the `cDataReader` sub-component, which handles the `dataMemory` interface for data input. See the documentation of ‘cDataReader’ for more information (section 3.3.2).

writer = <object of type ‘cDataWriter’> The configuration of the `cDataWriter` sub-component, which handles the `dataMemory` interface for data output. See the documentation of ‘cDataWriter’ for more information (section 3.3.2).

bufferSize = <numeric> [Default: 0]
 The buffer size for the output level, in frames (default [0] = same as input level), this option overwrites ‘bufferSize_sec’.

bufferSize_sec = <numeric> [Default: 0]
 The buffer size for the output level, in seconds (default [0] = same as input level).

blocksize = <numeric> [Default: 0]

The size of data blocks to process, in frames (this sets both ‘blocksizeR’ and ‘blocksizeW’, and overwrites ‘blocksize_sec’).

blocksizeR = <numeric> [Default: 0]

The size of data blocks to read, in frames (overwrites ‘blocksize’).

blocksizeW = <numeric> [Default: 0]

The size of data blocks to write, in frames (overwrites ‘blocksize’).

blocksize_sec = <numeric> [Default: 0]

The size of data blocks to process, in seconds (this sets both ‘blocksizeR_sec’ and ‘blocksizeW_sec’).

blocksizeR_sec = <numeric> [Default: 0]

The size of data blocks to read, in seconds (overwrites ‘blocksize_sec’).

blocksizeW_sec = <numeric> [Default: 0]

The size of data blocks to write in seconds (overwrites ‘blocksize_sec’).

nameAppend = <string> [Default: ‘null’]

A string suffix to append to the input field names (default: empty).

copyInputName = <numeric> [Default: 1]

1 = copy the input name (and optionally append a suffix, see the ‘nameAppend’ option)

0 = discard the input name and use only the ‘nameAppend’ string as new name.

noPostEOIprocessing = <numeric> [Default: 0]

1 = do not process incomplete windows at the end of the input.

leftwin = <numeric> [Default: 10]

The left (past) context window for smoothing, in frames (this overwrites ‘leftwin_sec’, if set).

leftwin_sec = <numeric> [Default: 0.1]

The left (past) context window for smoothing, in seconds (this will be rounded to the nearest number of frames).

rightwin = <numeric> [Default: 20]

The right (future) context window for smoothing or weighting (see the ‘doRightWeight’ option), in frames (this overwrites rightwin_sec, if set).

rightwin_sec = <numeric> [Default: 0.2]

The right (future) context window for smoothing or weighting (see ‘doRightWeight’ option), in seconds.

doRightWeight = <numeric> [Default: 1]

1 = use right mean for weighting

-1 = use left mean for weighting

0 = use right mean for differential only if ‘rightwin > 0’.

cWindower

Description: This component applies a window function (i.e. multiplies values of a frame with the values of the window function) to data frames.

Class hierarchy: cSmileComponent → cDataProcessor → cVectorProcessor → cWindower

Configuration options:

reader = <object of type 'cDataReader'> The configuration of the cDataReader sub-component, which handles the dataMemory interface for data input. See the documentation of 'cDataReader' for more information (section 3.3.2).

writer = <object of type 'cDataWriter'> The configuration of the cDataWriter sub-component, which handles the dataMemory interface for data output. See the documentation of 'cDataWriter' for more information (section 3.3.2).

bufferSize = <numeric> [Default: 0]
The buffer size for the output level, in frames (default [0] = same as input level), this option overwrites 'bufferSize_sec'.

bufferSize_sec = <numeric> [Default: 0]
The buffer size for the output level, in seconds (default [0] = same as input level).

blockSize = <numeric> [Default: 0]
The size of data blocks to process, in frames (this sets both 'blockSizeR' and 'blockSizeW', and overwrites 'blockSize_sec').

blockSizeR = <numeric> [Default: 0]
The size of data blocks to read, in frames (overwrites 'blockSize').

blockSizeW = <numeric> [Default: 0]
The size of data blocks to write, in frames (overwrites 'blockSize').

blockSize_sec = <numeric> [Default: 0]
The size of data blocks to process, in seconds (this sets both 'blockSizeR_sec' and 'blockSizeW_sec').

blockSizeR_sec = <numeric> [Default: 0]
The size of data blocks to read, in seconds (overwrites 'blockSize_sec').

blockSizeW_sec = <numeric> [Default: 0]
The size of data blocks to write in seconds (overwrites 'blockSize_sec').

nameAppend = <string> [Default: 'null']
A string suffix to append to the input field names (default: empty).

copyInputName = <numeric> [Default: 1]
1 = copy the input name (and optionally append a suffix, see the 'nameAppend' option)
0 = discard the input name and use only the 'nameAppend' string as new name.

processArrayFields = <numeric> [Default: 1]
1 = process each array field as one vector individually (and produce one output for each input array field). Only array fields (i.e. fields with more than one element) are processed

if this is set.

0 = process complete input frame as one vector, ignoring field/element structure.

gain = <numeric> [Default: 1]

This option allows you to specify a scaling factor by which window function (which is by default normalised to max. 1) should be multiplied by.

offset = <numeric> [Default: 0]

This specifies an offset which will be added to the samples after multiplying with the window function.

winFunc = <string> [Default: 'Han']

This option selects the window function to use. In the following equations n specifies a sample index in the range from $0..N-1$, and N specifies the frame size. You can choose from the following list of window functions:

Han Hann window (= raised cosine window; use this, if you want to re-synthesis from the spectral domain, also use 50% overlap in the framer!).

$$w_{Han}[n] = 0.5 \left(1 - \cos \left(\frac{2\pi n}{N-1} \right) \right) \quad (3.10)$$

Ham Hamming window

$$w_{Ham}[n] = 0.54 + 0.46 \cos \left(\frac{2\pi n}{N-1} \right) \quad (3.11)$$

Rec Rectangular window (i.e. constant 1, no windowing)

$$w_{Rec}[n] = 1 \quad (3.12)$$

Gau Gaussian window (see the 'sigma' option)

$$w_{Gau}[n] = e^{-\frac{1}{2} \left(\frac{n-(N-1)/2}{\sigma(N-1.0)/2} \right)^2} \quad (3.13)$$

Sin Sine/cosine window

$$w_{Sin}[n] = \sin \left(\frac{\pi n}{N-1.0} \right) \quad (3.14)$$

Tri Triangular window (non zero-valued end points)

$$w_{Tri}[n] = \begin{cases} \frac{2(n+1)}{N} & \text{if } n < \frac{N}{2} \\ \frac{2(N-n)}{N} & \text{if } n \geq \frac{N}{2} \end{cases}$$

Bar Bartlett window (= Triangular window with zero-valued end points)

$$w_{Bar}[n] = \begin{cases} \frac{2(n)}{N-1} & \text{if } n < \frac{N}{2} \\ \frac{2(N-n-1)}{N-1} & \text{if } n \geq \frac{N}{2} \end{cases}$$

BaH Bartlett-Hann window (see the 'alpha0-2' options)

$$w_{BaH}[n] = \alpha_0 - \alpha_1 \left| \frac{n}{N-1} - \frac{1}{2} \right| - \alpha_2 \cos \left(\frac{2\pi n}{N-1} \right) \quad (3.15)$$

The defaults for the α parameters are: $\alpha_0 = 0.62$, $\alpha_1 = 0.48$, $\alpha_2 = 0.38$

Bla Blackmann window (see the ‘alpha’ option)

$$w_{Bla}[n] = \frac{1 - \alpha}{2} - \frac{1}{2} \cos\left(\frac{2\pi n}{N-1}\right) + \frac{\alpha}{2} \cos\left(\frac{4\pi n}{N-1}\right) \quad (3.16)$$

The default for the α parameter is 0.16.

BlH Blackmann-Harris window (see the ‘alpha0-3’ options)

$$w_{BlH}[n] = \alpha_0 - \alpha_1 \cos\left(\frac{2\pi n}{N-1}\right) + \alpha_2 \cos\left(\frac{4\pi n}{N-1}\right) - \alpha_3 \cos\left(\frac{6\pi n}{N-1}\right) \quad (3.17)$$

The defaults for the α parameters are: $\alpha_0 = 0.35875$, $\alpha_1 = 0.48829$, $\alpha_2 = 0.14128$, $\alpha_3 = 0.01168$

Lac Lanczos window (also known as ‘sinc’ window)

$$w_{Lac}[n] = \frac{\sin\left(\pi \frac{2n}{N-1} - 1\right)}{\frac{2n}{N-1} - 1} \quad (3.18)$$

sigma = <numeric> [Default: 0.4]

Standard deviation σ for the Gaussian window. Recommended: $\sigma < 0.5$

alpha0 = <numeric> [Default: 0]

Parameter α_0 for Blackmann(-Harris) / Bartlett-Hann windows (optional!)

alpha1 = <numeric> [Default: 0]

Parameter α_1 for Blackmann(-Harris) / Bartlett-Hann windows (optional!)

alpha2 = <numeric> [Default: 0]

Parameter α_2 for Blackmann(-Harris) / Bartlett-Hann windows (optional!)

alpha3 = <numeric> [Default: 0]

Parameter α_3 for Blackmann-Harris window (optional!)

alpha = <numeric> [Default: 0.16]

Parameter α for the Blackmann window

3.3.7 Functionals

Functionals are computed by the cFunctionals component documented in the next section. This component contains various sub-components which implement the individual functionals groups.

cFunctionals

Description: This component applies functionals to a series of input frames and outputs one a single static ‘summary’ vector. The length of this vector is independent of the length of the input sequence. This component uses various cFunctional* sub-components, which implement the actual functionality. The configuration options of these sub-components are documented within this section.

Class hierarchy: cSmileComponent \rightarrow cDataProcessor \rightarrow cWinToVecProcessor \rightarrow cFunctionals

Configuration options:

reader = <object of type ‘cDataReader’> The configuration of the cDataReader sub-component, which handles the dataMemory interface for data input. See the documentation of ‘cDataReader’ for more information (section 3.3.2).

writer = <object of type ‘cDataWriter’> The configuration of the cDataWriter sub-component, which handles the dataMemory interface for data output. See the documentation of ‘cDataWriter’ for more information (section 3.3.2).

bufferSize = <numeric> [Default: 0]
The buffer size for the output level, in frames (default [0] = same as input level), this option overwrites ‘bufferSize_sec’.

bufferSize_sec = <numeric> [Default: 0]
The buffer size for the output level, in seconds (default [0] = same as input level).

nameAppend = <string> [Default: ‘(null)’]
A string suffix to append to the input field names (default: empty)

copyInputName = <numeric> [Default: 1]
1 = copy the input name (and optionally append a suffix, see ‘nameAppend’ option)
0 = discard the input name and use only the ‘nameAppend’ string as new name.

frameMode = <string> [Default: ‘fixed’]
Specifies how to create frames:

fixed fixed frame size, given via the ‘frameSize’ option

full creates one frame at the end of the input only (off-line processing)

variable via smile message from another component (such as cTurnDetector (section 3.3.6)

list frame times list in config file (‘frameList’ option), or in external text file (‘frameList-File’ option). Currently: UNIMPLEMENTED.

frameListFile = <string> [Default: ‘(null)’]
Filename of a file with a list of frame intervals to load. This should be a text file with a comma separated list of intervals on a single line: 1-10,11-20, etc., if no interval is specified, i.e. no – is found then consecutive frames with the given number being the frame length are assumed; first index is 0; use the suffix “s” after the numbers to specify intervals in seconds (e.g. 0s-2.5s); use an ‘E’ instead of a number for ‘end of sequence’).

frameList = <string> [Default: ‘(null)’]
The list of frame intervals specified directly in the configuration file. This should be a string of comma separated list of intervals on a single line: 1-10,11-20, etc., if no interval is specified, i.e. no – is found then consecutive frames with the given number being the frame length are assumed; first index is 0; use the suffix “s” after the numbers to specify intervals in seconds (e.g. 0s-2.5s); use an ‘E’ instead of a number for ‘end of sequence’).

frameSize = <numeric> [Default: 0.025]
The frame size in seconds (0.0 = full input, same as ‘frameMode=full’).

frameStep = <numeric> [Default: 0]
The frame step (frame sampling period) in seconds (0 = set to the same value as ‘frame-Size’)

frameSizeFrames = <numeric> [Default: 0]

The frame size in input level frames (=samples for a pcm/wave input level) (overrides frameSize, if set and > 0).

frameStepFrames = <numeric> [Default: 0]

The frame step in input level frames (=samples for a pcm/wave input level) (overrides frameStep, if set and > 0).

frameCenter = <numeric> [Default: 0]

The frame center in seconds, i.e. where frames are sampled (0=left), see ‘frameCenterSpecial’ for examples on how the frame center options affect the sampling of frames.

frameCenterFrames = <numeric> [Default: 0]

The frame sampling center in input level frames (overrides ‘frameCenter’, if set), (0=left), see ‘frameCenterSpecial’ for examples on how the frame center options affect the sampling of frames.

frameCenterSpecial = <string> [Default: ‘left’]

The frame sampling center (overrides the other ‘frameCenter’ options, if set). The available special frame sampling points as strings are:

left = sample at the beginning of the frame (the first frame will be sampled from 0 to frameSize)

mid = sample in the middle of the frame middle (the first frame will be sampled from -frameSize/2 to frameSize/2; values at negative indices are padded with zeros)

right = sample at the end of the frame (the first frame will be sampled from -frameSize to 0; values at negative indices are padded with zeros, i.e. the first frame will be all 0s)

noPostEOIprocessing = <numeric> [Default: 1]

1 = do not process incomplete windows at the end of the input, i.e. all created frames have been sampled from segments that are exactly ‘frameSize’ in length. Excess data at the end of the input will be discarded. This is only relevant for off-line processing.

functionalsEnabled[] = <string> [Default: ‘(null)’]

A string array that defines the list of enabled functionals. The following functionals are available (sub-components) (*Attention:* the names are case-SENSITIVE!):

	Name of group	Description
1.	Extremes	extreme values (max, min, range, maxPos, minPos, ...)
2.	Means	various mean values (arithmetic, geometric, quadratic, ...)
3.	Peaks	number of peaks and various measures associated with peaks, such as mean of peaks, mean distance between peaks, etc. Peak finding is based on : $x(t-1) < x(t) > x(t+1)$.
4.	Segments	number of segments based on simple delta thresholding
5.	Onset	relative position of the first onset and the last offset based on simple thresholding. Number of onsets and offsets can also be computed.
6.	Moments	statistical moments (standard deviation, variance, skewness, kurtosis)
7.	Crossings	zero-crossing rate, mean crossing rate, dc offset, min, and max value
8.	Percentiles	percentile values and inter-percentile ranges (including quartiles, etc.). This component sorts the input array and then chooses the value at the index closest to $p \cdot \text{segment_length}$ for the p -th percentile ($p=0..1$).
9.	Regression	linear and quadratic regression coefficients and corresponding linear and quadratic regression errors. Linear regression line: $y = mx + t$; quadratic regression parabola: $y = ax^2 + bx + c$. Algorithm: Minimum mean square error, direct analytic solution of system of equations. This component also computes the centroid of the contour.
10.	Samples	sampled values at equidistant frames
11.	Times	up- and down-level times + rise and fall, left- and right-curve times, duration, etc.
12.	DCT	Discrete Cosine Transformation (DCT) (type-II) coefficients

nonZeroFuncs = <numeric> [Default: 0]

If this is set to 1, functionals are only applied to input values unequal 0.

If this is set to 2, functionals are only applied to input values greater than 0. For the default value of this option (0) functionals are applied to all input values.

funcNameAppend = <string> [Default: '(null)']

Specifies a string suffix to append to the functional name (which is appended to the input feature name)

masterTimeNorm = <string> [Default: 'segment']

This option specifies how all components should normalise times, if they generate output values related to durations. You can change the 'norm' parameter of individual functional components to overwrite this master value. You can choose one of the following normalisation methods:

segment (or: 'turn') normalise to the range 0..1, the result is a relative length as a fraction of the turn length.

second absolute time in seconds.

frame absolute time in number of frames of the input level.

The following paragraphs describe the individual functional extractor components' options. These components are called 'cFunctionalXXXX' in openSMILE, e.g. 'cFunctionalExtremes',

etc. For simplicity we only refer to the last part of the name (the ‘XXXX’ part), e.g. ‘Extremes’. This is also the name of the configuration option type. This means to configure the options of the ‘cFunctionalExtremes’ component in a ‘cFunctionals’ component you must use the syntax `Extremes.option = value`.

Crossings . The ‘cFunctionalCrossings’ component computes the zero-crossing and mean-crossing rates, and the arithmetic mean (optionally).

Crossings.zcr = <numeric> [Default: 1]
1/0=enable/disable output of zero crossing rate. [field name suffix: **zcr**].

Crossings.mcr = <numeric> [Default: 1]
1/0=enable/disable output of mean crossing rate (the rate at which the signal crosses its arithmetic mean value, same as zcr for mean normalised signals). [field name suffix: **mcr**].

Crossings.amean = <numeric> [Default: 0]
1/0=enable/disable output of arithmetic mean. [field name suffix: **amean**].

DCT . The ‘cFunctionalDCT’ component applies a discrete cosine transformation of type-II to the input contour. The output field has the suffix ‘DCTn’ appended (where n is the index of the DCT coefficient). The range of the DCT coefficients can be configured via the configuration options:

DCT.firstCoeff = <numeric> [Default: 1]
The first DCT coefficient to compute (coefficient 0 corresponds to the DC component).

DCT.lastCoeff = <numeric> [Default: 6]
The last DCT coefficient to compute.

DCT.nCoeffs = <numeric> [Default: 6]
An alternative option to ‘lastCoeff’ (this option overwrites ‘lastCoeff’, if it is set): the number DCT coefficient to compute ($\text{lastCoeff} = \text{firstCoeff} + \text{nCoeffs} - 1$).

Extremes . The ‘cFunctionalExtremes’ component computes extreme values and associated attributes. The list of available configuration options is the documentation of the attributes that can be computed:

Extremes.max = <numeric> [Default: 1]
1/0=enable/disable output of maximum value. [field name suffix: **max**]

Extremes.min = <numeric> [Default: 1]
1/0=enable/disable output of minimum value. [field name suffix: **min**]

Extremes.range = <numeric> [Default: 1]
1/0=enable/disable output of range (max-min). [field name suffix: **range**]

Extremes.maxpos = <numeric> [Default: 1]
1/0=enable/disable output of position of maximum value (relative to the input segment length, in seconds, or in frames, see the ‘norm’ option or the ‘masterTimeNorm’ option of the cFunctionals parent component). [field name suffix: **maxpos**]

Extremes.minpos = <numeric> [Default: 1]
 1/0=enable/disable output of position of minimum value (relative to the input segment length, in seconds, or in frames, see the ‘norm’ option or the ‘masterTimeNorm’ option of the cFunctionals parent component). [field name suffix: **minpos**]

Extremes.amean = <numeric> [Default: 0]
 1/0=enable/disable output of the arithmetic mean. [field name suffix: **mean**]

Extremes.maxameandist = <numeric> [Default: 1]
 1/0=enable/disable output of (maximum value minus arithmetic mean). [field name suffix: **maxameandist**]

Extremes.minameandist = <numeric> [Default: 1]
 1/0=enable/disable output of (arithmetic mean - minimum value). [field name suffix: **minameandist**]

Extremes.norm = <string> [Default: ‘frames’]
 This option specifies how this component should normalise times (see also the ‘masterTimeNorm’ option in cFunctionals):

segment (or: ‘turn’) normalise to the range 0..1, the result is the relative length wrt. to the segment length.

second absolute time in seconds.

frame absolute time in number of frames of the input level.

Means . The ‘cFunctionalMeans’ component computes various mean values. The list of available configuration options is the documentation of the available mean values:

Means.amean = <numeric> [Default: 1]
 1/0=enable/disable output of arithmetic mean. [field name suffix: **amean**].

Means.absmean = <numeric> [Default: 1]
 1/0=enable/disable output of arithmetic mean of absolute values. [field name suffix: **absmean**].

Means.qmean = <numeric> [Default: 1]
 1/0=enable/disable output of quadratic mean. [field name suffix: **qmean**].

Means.nzamean = <numeric> [Default: 1]
 1/0=enable/disable output of arithmetic mean (of non-zero values only). [field name suffix: **nzamean**].

Means.nzabsmean = <numeric> [Default: 1]
 1/0=enable/disable output of arithmetic mean of absolute values (of non-zero values only). [field name suffix: **nzabsmean**].

Means.nzqmean = <numeric> [Default: 1]
 1/0=enable/disable output of quadratic mean (of non-zero values only). [field name suffix: **nzqmean**].

Means.nzgmean = <numeric> [Default: 1]
 1/0=enable/disable output of geometric mean (of absolute values of non-zero values only). [field name suffix: **nzgmean**].

Means.nnz = <numeric> [Default: 1]

1/0=enable/disable output of number of non-zero values (relative to the input segment length, in seconds, or in frames, see the 'norm' option or the 'masterTimeNorm' option of the cFunctionals parent component). [field name suffix: **nnz**].

Means.norm = <string> [Default: 'frames']

This option specifies how this component should normalise times (see also the 'masterTimeNorm' option in cFunctionals):

segment (or: 'turn') normalise to the range 0..1, the result is the relative length wrt. to the segment length.

second absolute time in seconds.

frame absolute time in number of frames of the input level.

Moments . The 'cFunctionalMoments' component computes the statistical moments, arithmetic mean, standard deviation, variance, skewness, and kurtosis.

Moments.variance = <numeric> [Default: 1]

1/0=enable/disable output of variance. [field name suffix: **variance**]

Moments.stddev = <numeric> [Default: 1]

1/0=enable/disable output of standard deviation. [field name suffix: **stddev**]

Moments.skewness = <numeric> [Default: 1]

1/0=enable/disable output of skewness. [field name suffix: **skewness**]

Moments.kurtosis = <numeric> [Default: 1]

1/0=enable/disable output of kurtosis. [field name suffix: **kurtosis**]

Moments.amean = <numeric> [Default: 0]

1/0=enable/disable output of arithmetic mean. [field name suffix: **amean**]

Onset . The 'cFunctionalOnset' component finds the first onset and the last offset in the contour, that is the position of the first value which is above a specified threshold, and the position of the first value which is below a given threshold and all values following this value up to the end of the segment are also below this threshold. Moreover, the number of onsets in total can be computed (the number of times when the signal changes from below to above the threshold).

Onset.threshold = <numeric> [Default: 0]

The absolute threshold used for onset/offset detection (i.e. the first onset will be where the input value is above the threshold for the first time).

Onset.thresholdOnset = <numeric> [Default: 0]

A separate threshold only for onset detection. This will override the 'threshold' option, if set.

Onset.thresholdOffset = <numeric> [Default: 0]

A separate threshold only for offset detection. This will override the 'threshold' option, if set.

Onset.useAbsVal = <numeric> [Default: 0]

1/0=yes/no : apply thresholds to absolute input value instead of original input value.

Onset.onsetPos = <numeric> [Default: 0]

1/0=enable/disable output of relative position (relative to the input segment length, in seconds, or in frames, see the ‘norm’ option or the ‘masterTimeNorm’ option of the ‘cFunctionals’ parent component) of first onset found. [field name suffix: **onsetPos**].

Onset.offsetPos = <numeric> [Default: 0]

1/0=enable/disable output of position of last offset found (relative to the input segment length, in seconds, or in frames, see the ‘norm’ option or the ‘masterTimeNorm’ option of the ‘cFunctionals’ parent component). [field name suffix: **offsetPos**]

Onset.numOnsets = <numeric> [Default: 1]

1/0=enable/disable output of the number of onsets found. [field name suffix: **numOnsets**]

Onset.numOffsets = <numeric> [Default: 0]

1/0=enable/disable output of the number of offsets found (this is usually redundant and the same as ‘numOnsets’, use this only for special applications where it may make sense to use it). [field name suffix: **numOffsets**]

Onset.norm = <string> [Default: ‘segment’]

This option specifies how this component should normalise times (see also the ‘masterTimeNorm’ option in cFunctionals):

segment (or: ‘turn’) normalise to the range 0..1, the result is the relative length wrt. to the segment length.

second absolute time in seconds.

frame absolute time in number of frames of the input level.

Peaks . The ‘cFunctionalPeaks’ component computes the number of peaks and various measures associated with peaks, such as mean of peaks, mean distance between peaks, etc. Peak finding is based on $x(t-1) < x(t) > x(t+1)$. The list of available configuration options is the documentation of the available peak associated measures:

Peaks.numPeaks = <numeric> [Default: 1]

1/0=enable/disable output of the number of peaks. [field name suffix: **numPeaks**]

Peaks.meanPeakDist = <numeric> [Default: 1]

1/0=enable/disable output of mean distance between peaks (relative to the input segment length, in seconds, or in frames, see the ‘norm’ option or the ‘masterTimeNorm’ option of the ‘cFunctionals’ parent component). [field name suffix: **meanPeakDist**]

Peaks.peakMean = <numeric> [Default: 1]

1/0=enable/disable output of arithmetic mean of peaks. [field name suffix: **peakMean**]

Peaks.peakMeanMeanDist = <numeric> [Default: 1]

1/0=enable/disable output of (arithmetic mean of peaks - arithmetic mean of all values). [field name suffix: **peakMeanMeanDist**]

Peaks.norm = <string> [Default: ‘frames’]

This option specifies how this component should normalise times (see also the ‘masterTimeNorm’ option in cFunctionals):

segment (or: ‘turn’) normalise to the range 0..1, the result is the relative length wrt. to the segment length.

second absolute time in seconds.

frame absolute time in number of frames of the input level.

Percentiles . The ‘cFunctionalPercentiles’ component can compute quartiles, inter-quartile ranges, and an arbitrary number of user-defined percentiles and percentile ranges. In order to efficiently compute percentiles, the values in the input segment are sorted once using a quicksort algorithm.

Percentiles.quartiles = <numeric> [Default: 1]
1/0=enable/disable output of all quartiles (overrides individual settings quartile1, quartile2, and quartile3).

Percentiles.quartile1 = <numeric> [Default: 0]
1/0=enable/disable output of quartile1 (0.25). [field name suffix: **quartile1**].

Percentiles.quartile2 = <numeric> [Default: 0]
1/0=enable/disable output of quartile2 (0.50). [field name suffix: **quartile2**].

Percentiles.quartile3 = <numeric> [Default: 0]
1/0=enable/disable output of quartile3 (0.75). [field name suffix: **quartile3**].

Percentiles.iqr = <numeric> [Default: 1]
1/0=enable/disable output of all inter-quartile ranges (overrides individual settings iqr12, iqr23, and iqr13).

Percentiles.iqr12 = <numeric> [Default: 0]
1/0=enable/disable output of inter-quartile range 1-2 (quartile2-quartile1). [field name suffix: **iqr12**].

Percentiles.iqr23 = <numeric> [Default: 0]
1/0=enable/disable output of inter-quartile range 2-3 (quartile3-quartile2). [field name suffix: **iqr23**].

Percentiles.iqr13 = <numeric> [Default: 0]
1/0=enable/disable output of inter-quartile range 1-3 (quartile3-quartile1). [field name suffix: **iqr13**].

Percentiles.percentile[] = <numeric> [Default: 0.9]
Array of p*100 percent percentiles to compute. p = 0..1. Array size indicates the number of total percentiles to compute (excluding quartiles), duplicate entries are not checked for and not removed : percentile[n] = p (p=0..1). [field name suffix: **percentile[p]**].

Percentiles.pctlrange[] = <string> [Default: ‘0-1’]
Array that specifies which inter percentile ranges to compute. A range is specified as ‘n1-n2’ (where n1 and n2 are the indices of the percentiles as they appear in the percentile[] array, starting at 0 with the index of the first percentile). [field name suffix: **pctlrange[a-b]**].

Percentiles.interp = <numeric> [Default: 1]
If set to 1, percentile values will be linearly interpolated, instead of being rounded to the nearest index in the sorted input array.

Regression . The ‘cFunctionalRegression’ component computes the parameters of a linear and a quadratic approximation of the segment contour that approximates the contour with the minimal quadratic error. Moreover, the linear and quadratic approximation errors for the linear and quadratic approximations can be computed. As a by-product the centroid (centre of gravity) of the contour is computed.

Regression.linregc1 = <numeric> [Default: 1]
1/0=enable/disable output of slope m (linear regression line). [field name suffix: **linregc1**]

Regression.linregc2 = <numeric> [Default: 1]
1/0=enable/disable output of offset t (linear regression line). [field name suffix: **linregc2**]

Regression.linregerrA = <numeric> [Default: 1]
1/0=enable/disable output of linear error between contour and linear regression line. [field name suffix: **linregerrA**]

Regression.linregerrQ = <numeric> [Default: 1]
1/0=enable/disable output of quadratic error between contour and linear regression line. [field name suffix: **linregerrQ**]

Regression.qregc1 = <numeric> [Default: 1]
1/0=enable/disable output of quadratic regression coefficient 1 (a). [field name suffix: **qregc1**]

Regression.qregc2 = <numeric> [Default: 1]
1/0=enable/disable output of quadratic regression coefficient 2 (b). [field name suffix: **qregc2**]

Regression.qregc3 = <numeric> [Default: 1]
1/0=enable/disable output of quadratic regression coefficient 3 (c = offset). [field name suffix: **qregc3**]

Regression.qregerrA = <numeric> [Default: 1]
1/0=enable/disable output of linear error between contour and quadratic regression line (parabola). [field name suffix: **qregerrA**]

Regression.qregerrQ = <numeric> [Default: 1]
1/0=enable/disable output of quadratic error between contour and quadratic regression line (parabola). [field name suffix: **qregerrQ**]

Regression.centroid = <numeric> [Default: 1]
1/0=enable/disable output of centroid of contour (this is computed as a by-product of the regression coefficients). [field name suffix: **centroid**]

Regression.normRegCoeff = <numeric> [Default: 0]
1/0=enable/disable normalisation of regression coefficients. If enabled, the coefficients are scaled (multiplied by the contour length) so that a regression line or parabola approximating the contour can be plotted over an x-axis range from 0 to 1, i.e. this makes the coefficients independent of the contour length (a longer contour with a lower slope will then have the same ‘m’ (slope) linear regression coefficient as a shorter but steeper slope). In detail the following scaling will be performed (N is the number of elements in the contour,

i.e. the contour length):

$$\begin{aligned}a' &= N^2 \cdot a \\ b' &= N \cdot b \\ c' &= c \\ m' &= N \cdot m \\ t' &= t\end{aligned}$$

Samples . The ‘cFunctionalSamples’ component grabs sample values at pre-defined relative positions within the input segment (such as beginning, middle, end, etc.).

Samples.samplepos = <numeric> [Default: 0]

Array of positions of samples to copy to the output. The size of this array determines the number of sample frames that will be passed to the output. The given positions must be in the range from 0 to 1, indicating the relative position within the input segment, where 0 is the beginning and 1 the end of the segment.

Segments . The ‘cFunctionalSegments’ component detects continuous segments based on delta-thresholding. The beginning of a segment is thereby detected by a rise in the values (difference of the current value minus a long term moving average (computed over segmentSize/(0.5 * maxNumSeg) frames) which is above a threshold relative to the range of the values in the input segment.

Segments.maxNumSeg = <numeric> [Default: 20]

Maximum number of segments to detect.

Segments.rangeRelThreshold = <numeric> [Default: 0.2]

The segment threshold relative to the signal’s range (max-min).

Segments.numSegments = <numeric> [Default: 1]

1/0=enable/disable output of the number of detected segments (the output value is in the range 0..1, due to normalisation with maxNumSeg). [field name suffix: **numSegments**]

Segments.meanSegLen = <numeric> [Default: 1]

1/0=enable/disable output of the mean segment length (relative to the input segment length, in seconds, or in frames, see the ‘norm’ option or the ‘masterTimeNorm’ option of the ‘cFunctionals’ parent component). [field name suffix: **meanSegLen**]

Segments.maxSegLen = <numeric> [Default: 1]

1/0=enable/disable output of the maximum segment length (relative to the input segment length, in seconds, or in frames, see the ‘norm’ option or the ‘masterTimeNorm’ option of the ‘cFunctionals’ parent component). [field name suffix: **maxSegLen**]

Segments.minSegLen = <numeric> [Default: 1]

1/0=enable/disable output of the minimum segment length (relative to the input segment length, in seconds, or in frames, see the ‘norm’ option or the ‘masterTimeNorm’ option of the ‘cFunctionals’ parent component). [field name suffix: **minSegLen**]

Segments.norm = <string> [Default: ‘frames’]

This option specifies how this component should normalise times (see also the ‘masterTimeNorm’ option in cFunctionals):

segment (or: ‘turn’) normalise to the range 0..1, the result is the relative length wrt. to the segment length.

second absolute time in seconds.

frame absolute time in number of frames of the input level.

Times . The ‘cFunctionalTimes’ component computes various time statistics, such as up- and down-level times (i.e. the time the signal is above or below a certain percentage of its total range).

Times.upleveltime25 = <numeric> [Default: 1]
(1/0=yes/no) compute time where signal is above (0.25*range+min). [field name suffix: upleveltime25]

Times.downleveltime25 = <numeric> [Default: 1]
(1/0=yes/no) compute time where signal is below (0.25*range+min). [field name suffix: upleveltime25]

Times.upleveltime50 = <numeric> [Default: 1]
(1/0=yes/no) compute time where signal is above (0.50*range+min). [field name suffix: upleveltime50]

Times.downleveltime50 = <numeric> [Default: 1]
(1/0=yes/no) compute time where signal is below (0.50*range+min). [field name suffix: downleveltime50]

Times.upleveltime75 = <numeric> [Default: 1]
(1/0=yes/no) compute time where signal is above (0.75*range+min). [field name suffix: upleveltime75]

Times.downleveltime75 = <numeric> [Default: 1]
(1/0=yes/no) compute time where signal is below (0.75*range+min). [field name suffix: downleveltime75]

Times.upleveltime90 = <numeric> [Default: 1]
(1/0=yes/no) compute time where signal is above (0.90*range+min). [field name suffix: upleveltime90]

Times.downleveltime90 = <numeric> [Default: 1]
(1/0=yes/no) compute time where signal is below (0.90*range+min). [field name suffix: downleveltime90]

Times.risetime = <numeric> [Default: 1]
(1/0=yes/no) compute time during which the signal is rising. [field name suffix: risetime]

Times.falltime = <numeric> [Default: 1]
(1/0=yes/no) compute time during which the signal is falling. [field name suffix: falltime]

Times.leftctime = <numeric> [Default: 1]
(1/0=yes/no) compute time during which the signal has left curvature. [field name suffix: leftctime]

- Times.rightctime** = <numeric> [Default: 1]
 (1/0=yes/no) compute time during which the signal has right curvature. [field name suffix: rightctime]
- Times.duration** = <numeric> [Default: 1]
 (1/0=yes/no) compute total contour duration time, in frames (or seconds, if (time)norm==seconds).
 . [field name suffix: duration]
- Times.upleveltime[]** = <numeric> [Default: 0.9]
 compute time where signal is above X*range : upleveltime[n]=X . [field name suffix: upleveltime[n]]
- Times.downleveltime[]** = <numeric> [Default: 0.9]
 compute time where signal is below X*range : downleveltime[n]=X . [field name suffix: downleveltime[n]]
- Times.norm** = <string> [Default: 'frames']
 This option specifies how this component should normalise times (see also the 'masterTimeNorm' option in cFunctionals):
- segment** (or: 'turn') normalise to the range 0..1, the result is the relative length wrt. to the segment length.
 - second** absolute time in seconds.
 - frame** absolute time in number of frames of the input level.

3.4 Feature names

This section will soon contain a table which maps feature names (as generated by default settings), onto a corresponding description or links to the component reference sections where the features are described. Please note, that this information is also contained in the documentation of each component, but is summarised here for easier viewing. You must also be aware that most feature names in openSMILE can be changed via options in the configuration files. Thus, the names listed in this section might not be those you get with your configuration file. However, we consider it a bad practice to use your own names for the features if there is no obvious reason to do so. Using the default names ensures compatibility of feature files and feature selection lists.

openSMILE follows a strict naming scheme for features (data fields). Each component (except the sink components), assigns names to its output fields. All cDataProcessor descendants have two options to control the naming behaviour, namely 'nameAppend' and 'copyInputName'. 'nameAppend' specifies a suffix which is appended to the field name of the previous level. A '-' is inserted between the two names (if 'nameAppend' is not empty or (null)). 'copyInputName' controls whether the input name is copied and the suffix 'nameAppend' and any internal hard-coded names are appended (if it is set to 1), or if the input field name is discarded and only the component's internal names and an appended suffix are used.

The field naming scheme is illustrated by the following example. Let's assume you start with an input field 'pcm' (as the cWaveSource component (section 3.3.3) generates it). If you then compute delta regression coefficients from it, you end up with the name 'pcm-de'. If you apply functionals (extreme values max and min only), then you will end up with two new fields: 'pcm-de-max' and 'pcm-de-min'. Theoretically, if the 'copyInputName' is always set, and a suitable suffix to append is specified, the complete processing chain can be deducted from the field name. In practice, however, this would lead to quite long and redundant feature names,

since most speech and music features base on framing, windowing, and spectral transformation. Thus, most of these components do not append anything to the input name and do only copy the input name. In order to discard the ‘pcm’ from the wave input level, components that compute features such as mfcc, pitch, etc. discard the input name and use only a hard-coded name or a name controlled via ‘nameAppend’.

Chapter 4

Developer's Documentation

The developer's documentation has not yet been included in this document. Fragments of the documentation covering various aspects briefly are found in the `doc/developer` directory.

Writing plugins If you are interested in writing plugins for openSMILE, read the document `doc/developer/implementing_components.txt` and write a component `cpp` and `hpp` file. Then have a look at the Makefiles (Linux) in the `pluginddev` directory for building your plugin on linux, and the Visual Studio Solution files (`openSmilePlugin`) for windows in the `ide/vs05/` folder.

The main source file of a plugin is the `pluginddev/pluginMain.cpp` file. This file includes the individual component files this plugin shall contain, similar to the component list in the `componentManager.cpp` file, which manages the openSMILE built-in components.

Chapter 5

Additional Support

If you have questions which are not covered by this documentation please contact Florian Eyben (fe at audeering.com).

Chapter 6

Acknowledgement

The development of openSMILE and openEAR has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement No. 211486 (SEMAINE).