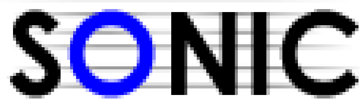
SONIC: The University of Colorado Continuous Speech Recognizer



Technical Report TR-CSLR-2001-01

Bryan Pellom and Kadri Hacıoğlu
{pellom, hacioglu}@cslr.colorado.edu

Center for Spoken Language Research
University of Colorado, Boulder

March 2, 2001
Revised: May 31, 2005

Table of Contents

NON-COMMERCIAL SOFTWARE LICENSE AGREEMENT 1

1 OVERVIEW 2

2 WHAT'S NEW IN 2.0-BETA5? 2

3 REVISION HISTORY 3

4 RECENT PAPERS PUBLISHED USING SONIC 3

5 FEATURES OF SONIC 2.0-BETA5 5

6 INSTALLATION 6

6.1 Setting up your Installation configuration 6
6.2 Unix/Linux Setup (Sun Solaris, Mac OS X, Red Hat Linux / Fedora) 6
6.3 Microsoft Windows Setup 7
6.4 Byte Ordering and Machine Architecture Compatibility Issues 7

7 DIRECTORY ORGANIZATION & SOFTWARE 8

8 THE SONIC SPEECH RECOGNIZER 9

8.1 Feature Representation 9
8.2 Acoustic Model 11
8.3 Language Model 12
8.4 Phonetic Symbol Set 12
8.5 Token-Pass Search Algorithm 13

9 PERFORMANCE ON STANDARD TASKS 13

10 USAGE GUIDE 14

10.1 Overview 14
10.2 Audio file Format 14
10.3 Phoneme Configuration File Format 15
10.4 Statistical N-gram Language Model Format 15
10.5 Pronunciation Lexicon Format 15
10.6 Acoustic Model Format 16
10.7 SONIC Configuration File 16
10.8 Output Format 22

11 ACOUSTIC MODEL TRAINING 22

11.1 Data Preparation 22
11.2 Feature Extraction 23
11.3 State-Based Alignment of training data 23
11.4 Master Label File Specification 28
11.5 Data Extraction from Master Label File 29
11.6 HMM Model Estimation 30
11.7 Generating the Final Acoustic Model 32
11.8 Advanced Feature: Distributed Computer HMM Training 32

12 KEYWORD SPOTTING AND GRAMMAR RECOGNITION 33

13 WORD-LEVEL CONFIDENCE ANNOTATION 34

14 ADAPTING TO SPEAKER AND ENVIRONMENT 34

14.1 Vocal Tract Length Normalization (VTLN) 34
14.2 Maximum Likelihood Linear Regression (MLLR) 36
14.3 Constrained Maximum Likelihood Linear Regression (CMLLR) 37
14.4 Structured Maximum a Posterior Linear Regression (SMAPLR) 39
14.5 Cepstral Variance Normalization 40

15 VOICE ACTIVITY DETECTION 40

16 FAST-MATCH SEARCH ALGORITHM 41

17 LETTER-TO-SOUND PREDICTION ALGORITHM 42

18 CLIENT-SERVER AND DISTRIBUTED RECOGNITION INTERFACE 44

18.1 Server Implementation 44
18.2 Client Implementation 45
18.3 Speech Compression Client Interface 47

19 APPLICATION PROGRAMMING INTERFACE (API) 48

19.1 Creating a new application 48
19.2 The SONIC Speech Recognition API 49
19.3 Example Use of the SONIC API 52
19.4 Error Message Handling 54

REFERENCES 55

NON-COMMERCIAL SOFTWARE LICENSE AGREEMENT

Center for Spoken Language Research
University of Colorado, Boulder
Copyright (c) 2005
All Rights Reserved

The University of Colorado ("CU") hereby grants to you an irrevocable, nonexclusive, nontransferable, perpetual, royalty-free and worldwide license to use this source code solely for educational, research, personal use, or evaluation. Limitations of Use are described below. A commercial license can be obtained only through expressed written permission of The Center for Spoken Language Research, University of Colorado at Boulder.

Non-Commercial License Limitations and Use Requirements

Limitations on Use: The License is limited to noncommercial use. Noncommercial use relates only to educational, research, personal use, or evaluation purposes. Any other use is commercial use. You may not use the Software in connection with any business activities. *You may not redistribute the software*.

Conditions of Use: This work is furnished to you under the following conditions:

1. The code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Any modifications must be clearly marked as such.
3. Original authors' names are not deleted.
4. The authors' names are not used to endorse or promote products derived from this software without specific prior written permission.
5. You agree to acknowledge CU-Boulder's Center for Spoken Language Research with appropriate citations in any publication or presentation containing research results obtained in whole or in part through the use of the Software.
6. You agree to provide all updates, algorithm enhancements, modifications, bug fixes, integrated into the SONIC source code to the University of Colorado, Center for Spoken Language Research for use and redistribution without restriction.

Term of License: The License is effective upon receipt by you of the Software and shall continue until terminated. The License will terminate immediately without notice by CU if you fail to comply with the terms and conditions of this Agreement. Upon termination of this License, you shall immediately discontinue all use of the Software provided hereunder, and return to CU or destroy the original and all copies of all such Software. All of your obligations under this Agreement shall survive the termination of the License.

DISCLAIMER: THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Copyright (c) 2005 Bryan Pellom

Permission is hereby granted, free of charge, to the University of Colorado *who has obtained a copy of this software from Bryan Pellom* and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

1 Overview

SONIC is a complete toolkit for research and development of new algorithms for continuous speech recognition. The software has been under development at CSLR since March of 2001. SONIC represents our test bed for integrating new ideas and for supporting research activities that include speech recognition as core components at the University of Colorado. The toolkit is far from complete and is not meant as a general purpose HMM toolkit (as is the case for the HTK toolkit from Cambridge). SONIC is specifically designed for speech recognition research with careful attention applied for speed and efficiency needed for real-time use in live applications.

SONIC is based on continuous density hidden Markov model (CDHMM) technology. The acoustic models are decision-tree state-clustered HMMs with associated gamma probability density functions to model state-durations. The recognizer implements a two-pass search strategy. The first pass consists of a time-synchronous, beam-pruned Viterbi token-passing search through a lexical prefix tree. Cross-word acoustic models and trigram or four-gram language models are applied in the first pass of search. During the second pass, the resulting word-lattice is converted into a word-graph. Longer span language models can be used to rescore the word graph using an A* algorithm or to compute word-posterior probabilities to provide word-level confidence scores.

The recognizer toolkit consists of a core speech recognition engine and programming interface (API). The current implementation allows for two modes of speech recognition:

1. Keyword / Grammar Decoding – continuous speech recognition constrained by a finite-state grammar. This mode also allows for keyword and grammar spotting capabilities.
2. N-gram Decoding – speech recognition based on statistical n-gram language models

SONIC incorporates speaker adaptation and normalization methods such as Maximum Likelihood Linear Regression (MLLR), Vocal Tract Length Normalization (VTLN), and cepstral mean & variance normalization. In addition advanced language-modeling strategies such as concept language models are also incorporated into the toolkit.

With this manual you will learn how to obtain, install, and use the SONIC recognizer and embed recognition functions into working applications. In addition, this document will teach you the basic skills needed to retrain acoustic models for new tasks and make necessary adjustments to configuration parameters to optimize the recognition performance.

2 What's New in 2.0-beta5?

SONIC 2.0-beta5 has been optimized for run-time speed and acoustic model footprint size compared to version 2.0-beta3. The token-passing search as well as code related to recognizer initialization for large vocabularies has been studied using the GCC profiler. On most tasks 2.0-beta5 executes approximately 2 to 2.5 faster than the previous release. The acoustic model trainer in SONIC now can build decision-tree state-clustered HMMs using splitting questions which are automatically derived from data. The resulting acoustic models are now stored in a compressed format space efficiency without resulting in a loss in accuracy. For most tasks, the acoustic models of SONIC are 6 times smaller than those built using version 2.0-beta3. The program "lm_encode" has been optimized for quick compression very large n-gram ARPA formatted files. Finally, in this release, we have optimized the front-end feature extraction in SONIC for speed and efficiency and we have once again switched back to a new and improved implementation of our Mel-Cepstral features.

3 Revision History

Updates By:	Date	Reason For Changes	Version
Bryan Pellom	03/02/01	Initial Technical Specification	1.0
Bryan Pellom	07/21/01	Software Update	1.0-beta1
Bryan Pellom	11/05/01	Software Update	1.0-beta2
Bryan Pellom & Kadri Hacioglu	01/29/02	Software Update	1.0-beta3
Bryan Pellom & Kadri Hacioglu	02/25/02	Software Update	1.0-beta4
Bryan Pellom & Kadri Hacioglu	04/07/02	Software Update	1.0-beta5
Bryan Pellom & Kadri Hacioglu	07/01/02	Software Update	1.0-beta6
Bryan Pellom & Kadri Hacioglu	08/11/02	Software Update	1.0-beta7
Bryan Pellom & Kadri Hacioglu	01/23/03	Software Update	1.0-beta8
Bryan Pellom & Kadri Hacioglu	05/15/03	Software Update	1.0-beta9
Bryan Pellom & Kadri Hacioglu	07/05/03	Software Update	2.0-beta1
Bryan Pellom & Kadri Hacioglu	02/06/04	Software Update	2.0-beta2
Bryan Pellom	06/21/04	Software Update	2.0-beta3
Bryan Pellom	03/16/05	Software Update (unreleased)	2.0-beta4
Bryan Pellom	05/31/05	Software Update	2.0-beta5

4 Recent Papers Published using SONIC

Several research projects at the University of Colorado utilize the SONIC speech recognizer as a core component application technology. Such projects have included DARPA SPINE (speech recognition in noisy environments), CU-Move (in-vehicle dialog and speech recognition), pronunciation modeling (funded by NSF), and the Foundations to Literacy program which provides interactive digital books for children (funded by NSF, Dept. of Ed, NIH). Below is a list of recent papers that have incorporated research results using SONIC. Each paper is available from the CSLR website (http://cslr.colorado.edu):

[1] Andreas Hagen and Bryan Pellom, "A Multi-Layered Lexical-Tree Based Token Passing Architecture for Efficient Recognition of Subword Speech Units", in 2nd Language & Technology Conference, Poznan, Poland, April, 2005

[2] Andreas Hagen, Bryan Pellom, et al., "Advances in Children`s Speech Recognition within an Interactive Literacy Tutor", in HLT NAACL 2004, Boston, May, 2004

[3] Bryan Pellom, Kadri Hacioglu, "Recent Improvements in the CU SONIC ASR System for Noisy Speech: The SPINE Task", in Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP), Hong Kong, April, 2003.

[4] Kadri Hacioglu, Bryan Pellom, "A Distributed Architecture for Robust Automatic Speech Recognition", in Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP), Hong Kong, April, 2003.

[5] Ayako Ikeno, Bryan Pellom, Dan Cer, Ashley Thornton, Jason Brenier, Dan Jurafsky, Wayne Ward, William Byrne, "Issues in Recognition of Spanish-Accented Spontaneous English", in ISCA & IEEE Workshop on Spontaneous Speech Processing and Recognition, Tokyo, Japan, April, 2003.

[6] O. Salor, B.L. Pellom, T. Ciloglu, K. Hacioglu, M. Demirekler, "On Developing New Text and Audio Corpora and Speech Recognition Tools for the Turkish Language", International Conference on Spoken Language Processing (ICSLP), vol. 1, pp. 349-352, Denver, Colorado USA, September 2002.

[7] B. Pellom, "SONIC: The University of Colorado Continuous Speech Recognizer," Technical Report TR-CSLR-2001-01, University of Colorado, March 2001.

**If you use the SONIC speech recognizer in your own research and find it useful, we ask that you kindly consider referencing papers [3] and [7] as a general references to SONIC.

5 Features of SONIC 2.0-beta5

- *Phonetic Aligner*
  - Provides word, phone, and HMM state-level boundaries for acoustic training.
  - Decision-tree based trainable letter-to-sound prediction module
  - Multilingual lexicon support
- *Phonetic Decision Tree Acoustic Trainer*
  - Estimates parameters of state-clustered continuous density Hidden Markov Models
  - Incorporates phonetic position & context questions
  - Distributed / parallel acoustic trainer (multiple machines, operating systems, CPUs)
- *Core Recognizer*
  - Token-passing based recognition using a lexical-prefix tree search
  - Cross-word triphone models & up to 4-gram language model in first-pass
  - HMM state durations modeled using Gamma distributions
  - N-best list output; Lattice dumping, and second-pass rescoring functionality
  - Word confidence computed from word-posteriors of word-graph
  - Class-based, word-based, and concept-based n-gram language models
  - Dynamically switched statistical language models (dialog state-conditioned LMs)
  - Keyword & regular expression based grammar spotting with confidence
  - Phonetic fast-match constrained ASR search for improved decoding speed
  - MFCC feature representation
- *Speaker Adaptation*
  - (Confidence Weighted) Maximum Likelihood Linear Regression (MLLR)
  - Constrained Maximum Likelihood Linear Regression (CMLLR)
  - Lattice-based MLLR (Lattice-MLLR)
  - Maximum a Posterior Linear Regression (MAP-LR)
  - Vocal Tract Length Normalization (VTLN)
  - Cepstral mean and variance normalization
- *Live-Mode Recognition*
  - API for streaming audio for keyword/grammars and continuous speech recognition
- *Voice Activity Detection*
  - Internal, HMM-based
- *Language Portability*
  - Aligner, trainer, recognizer designed to incorporate new phone sets and foreign vocabularies. SONIC has been successfully ported to over 15 languages
- *Speech Compression Interface*
  - Libraries and APIs for utilizing the Speex subband CELP coding system are provided within SONIC. SONIC in server mode accepts both raw PCM and compressed bitstreams.
- *Application Programming Interface (API)*
  - API environment for linking and designing speech enabled applications.
  - Batch-mode and simulated live-mode example source code
  - Socket-based Tcl/Tk client / C-code server example
- *Supported Operating Systems*: Linux, SunOS, MS Windows, and Mac OS X

6 Installation

The SONIC distribution provides a feature extraction module, an automatic speech time-alignment module, and an acoustic model building application. The recognizer is distributed with an API interface and an example batch-mode application (`sonic_batch`) and client/server application (`sonic_server`) to illustrate how API commands can be used to build speech-enabled applications.

The current version of SONIC is 2.0-beta5. SONIC was originally designed for Red Hat Linux (and currently Fedora) running on Intel-based processors. It has also been successfully compiled and tested on Sun Solaris, Microsoft Windows, and Mac OS X. Recognizer speed may not be as fast in Windows.

6.1 Setting up your Installation configuration

Compile options for SONIC are provided in (`sonic/2.0-beta5/Makefile.defines`) for Unix installations and in (`sonic/2.0-beta5/Makefile.defines.Win32`) for Windows installation. If you are compiling on a Unix platform (Sun Solaris, Linux, Mac OS X), and have the Open Sound System (OSS) installed on your computer, then edit the line "`OSS_COMPILE`" and "`OSS_PATH`". The OSS compile option builds executables for playing and recording audio files on your computer. In general, compile options are provided for GNU GCC for Unix installations and Microsoft's Visual Studio C++ (cl) compiler for Windows.

6.2 Unix/Linux Setup (Sun Solaris, Mac OS X, Red Hat Linux / Fedora)

Recompile Method

SONIC is typically not distributed in source code form. However, if you have a licensed source distribution it can be easily recompiled. If not, skip to the path setup section. To compile SONIC, uncompress the distribution, (`tar zxf sonic-2.0-beta5.tar.gz`). You can now compile the distribution. SONIC can be compiled either using GCC or the Intel C++ Compiler version 7.0. Edit the file (`sonic/2.0-beta5/Makefile.defines`) if you wish to use a compiler other than GCC. Next, compile the distribution,

```
cd sonic/2.0-beta5
gmake clean
gmake
```

Path Setup:

Based on your machine's operating system (OS) and architecture (ARCH), the binary executables will be installed within a unique machine-dependent directory. For example, the following directories assuming (ARCH = i686) and (OS = Linux):

```
sonic/bin/i686-Linux
sonic/contrib/bin/i686-Linux
```

It is highly recommended that you include this directory into your Unix environment path. In addition, for proper installation of the software, the following Unix environment variable should be set,

```
setenv ALIGN_BASE $install_path/sonic/2.0-beta5
```

where `$install_path` is the Unix file path where the software has been uncompressed. The examples in the following sections assume that this environment variable and binary executable directory are in your Unix path.

6.3 Microsoft Windows Setup

Recompile with Method

SONIC is typically not distributed in source code form. However, if you have a licensed source distribution it can be easily recompiled. If not, skip to the path setup section. To compile SONIC in MS Windows you must have the Microsoft Visual Studio C++ 6.0 (or higher) compiler. Our installation is based on the using the command-line driven compiler (`cl`). To compile in MS Windows, execute a DOS shell window (i.e., select "start", "run") and then type "`cmd`". Then assuming the Visual Studio environment is in your shell path, execute the following commands,

```
cd sonic\2.0-beta4
compile-Win32.bat
```

Path Setup: (Required for all users of SONIC):

The MS Windows binary executables for the SONIC recognizer will be installed within a unique machine-dependent directory. For example,

```
sonic\2.0-beta5\bin\Windows-NT
sonic\2.0-beta5\contrib\bin\Windows-NT
```

It is highly recommended that you include this directory into your Windows environment path. In addition, for proper installation of the software, the following environment variable should be set,

```
SET ALIGN_BASE=$install_path\sonic\2.0-beta5
```

where `$install_path` is the file path where the software has been uncompressed.

6.4 Byte Ordering and Machine Architecture Compatibility Issues

SONIC has been designed for interoperability between computer platforms that differ in byte ordering (i.e., Intel vs. Sun platforms). To deal with byte ordering issues, all extracted features, acoustic models, and compiled language models are stored in little endian format.

The exception to this philosophy is in audio formats. It is always assumed that audio is stored in raw (headerless) PCM samples *in the native byte ordering of the machine on which SONIC is executed*. Microsoft wav (RIFF) formatted audio is currently not supported.

7 Directory Organization & Software

The SONIC recognizer contains several modules that are compiled as libraries. These libraries constitute the core of the recognition engine and API. Below is a description of the current software layout:

Directory	Description
`bin/`	Binary executables
`config/`	Phonetic aligner configuration and models
`doc/`	Documentation, Tutorials & Examples
`lib/`	Precompiled libraries
`src/`	Program Source code (if provided)
`adapt/`	*Speaker & channel adaptation routines*
`aligner/`	*Viterbi aligner for acoustic training*
`app/`	*Example applications built with SONIC library* (a) *continuous speech [batch mode] application* (b) *continuous speech [server mode] application*
`decoder/`	*Main decoder functions (tree search, lexicon, API)*
`enhance/`	*Speech Enhancement API (will be integrated in 2.0-beta5)*
`features/`	*Feature extraction modules*
`lattice/`	*Lattice & word-graph processing routines, confidence*
`lexicon/`	*Lexicon / Phonetic dictionary support routines*
`lm/`	*Language Model encoding and access routines*
`misc/`	*Miscellaneous support libraries (signal processing, HMMs, etc.)*
`text2phone/`	*Decision tree letter-to-sound module (API, training programs)*
`trainer/`	*Phonetic decision tree Acoustic Trainer*
`voip/`	*Speech Compression libraries from Speex-1.0.1 (www.speex.org)*

When compiled, SONIC will create the following binary executable programs in addition to providing a set of linkable libraries with programming interfaces:

- `align` — *phonetically aligns audio data for acoustic training*
- `cmllr` — *Constrained Maximum Likelihood Linear Regression adaptation*
- `fea` — *Extracts features for acoustic training*
- `gram2vocab` — *Converts finite-state grammars into a list of unique words*
- `hmm_train` — *Hidden Markov Model (HMM) training routine*
- `hmm_train_ctl` — *Distributed HMM training support routine*
- `lex_encode` — *Compresses an ASCII formatted lexicon into a binary file*
- `lm_encode` — *Compresses an ASCII formatted language model into a binary*
- `mlf2bin` — *Extracts features and alignments prior to acoustic training*
- `mllr` — *Maximum Likelihood Linear Regression speaker adaptation*
- `sonic_batch` — *Batch-mode recognition application*
- `sonic_server` — *Live-Mode Server application*
- `sspell` — *"Sonic Spell", a phonetic dictionary access tool*
- `t2p_fea` — *Text2Phone feature extractor. Used to build letter2sound rules*
- `t2p_train` — *Text2Phone decision tree training module.*
- `tree_maplr` — *MAP based Linear Regression speaker adaptation*
- `varnorm` — *Cepstral Variance Normalization support routine*
- `voip_server` — *Audio Codec server based on CELP algorithm*
- `vtln` — *Vocal Tract Length Normalization Warp computation*

8 The SONIC Speech Recognizer

SONIC is based on continuous density hidden Markov model (CDHMM) technology. The acoustic models are decision-tree state-clustered HMMs with associated gamma probability density functions to model state-durations. The recognizer implements a two-pass search strategy. The first pass consists of a time-synchronous, beam-pruned Viterbi token-passing search through a lexical prefix tree. Crossword acoustic models and trigram or four-gram language models are applied in the first pass of search. During the second pass, the resulting word-lattice is converted into a word-graph. Longer span language models can be used to rescore the word graph using an A* algorithm or to compute word-posterior probabilities to provide word-level confidence scores.

8.1 Feature Representation

While Sonic supports the use of any arbitrary feature representation as input, the recognizer has been developed to natively use a standard 39-dimensional feature vector consisting of 12 Mel Frequency Cepstral Coefficients (MFCCs) and normalized frame energy along with the first and second order derivatives of the features. The input feature vector is calculated once every 10 ms from a sliding window of 20 ms of audio. This 39-dimensional feature representation is used by most commercial speech recognition systems.

Observation vectors are computed every 10 ms from a 20 ms short-time windowed segment of audio data. For each frame of audio, a 13-dimensional observation vector is extracted consisting of 12 Mel Frequency Cepstral Coefficients (MFCC) [1] and one normalized log-energy term. A block diagram of the MFCC feature extraction process is illustrated in Fig. 1.

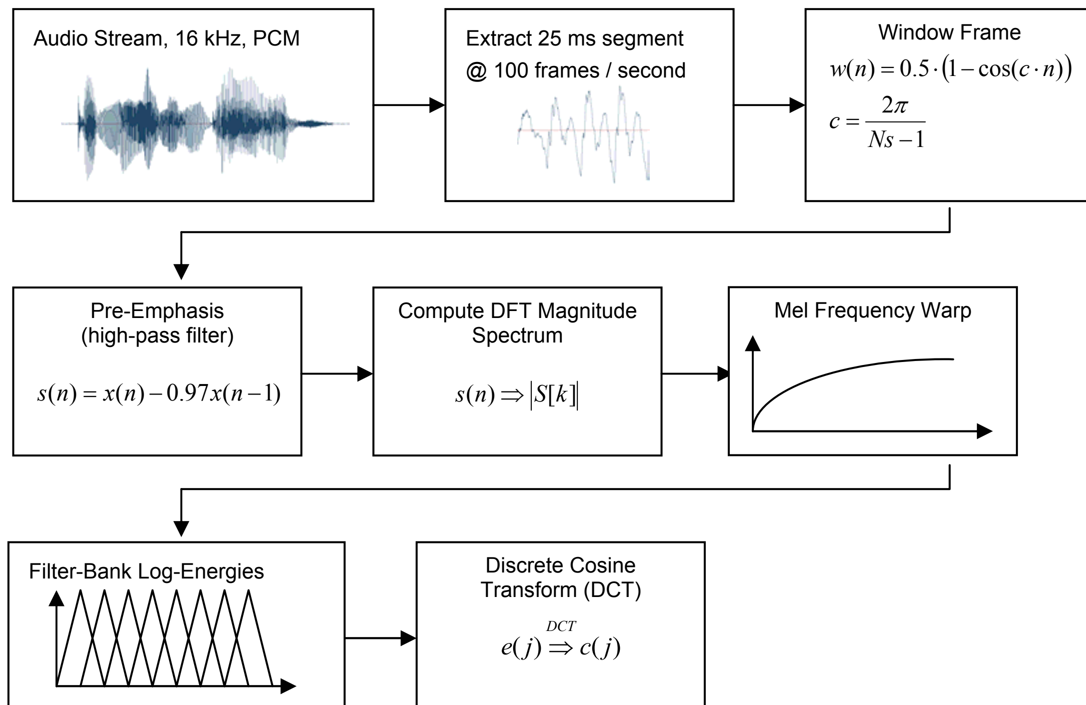


Figure 1: Block Diagram of Mel-Frequency Cepstral Coefficient (MFCC) calculation

Feature extraction begins by pre-emphasizing the audio to remove glottal and lip radiation effects. The pre-emphasis operation is implemented as a first order Finite Impulse Response (FIR) filter given by[1]

$$H(z) = 1 - 0.97z^{-1}$$

where z represents a one sample delay. Next, the magnitude spectrum of the waveform is computed using the Discrete Fourier Transform (DFT). The linear frequency axis is then warped onto the Mel scale in order to take into account the relationship between frequency and "perceived" pitch. The mapping between the linear frequency scale and Mel scale is given by

$$f_{mel} = 2595 \log_{10}\left(1 + \frac{f_{linear}}{700}\right)$$

The warped magnitude spectrum is next passed through a bank of triangular-shaped filters that uniformly partition the Mel frequency scale into P regions. For 16 kHz sampled audio, 20 filters (P=20) are used. The filter outputs generate a discrete set of P log-energy terms, ($e[j]$, j=1..P). Let $w_j[k]$ represent the weight of the jth filter to the kth discrete frequency of the sampled signal $s(n)$ and let $|S_{mel}[k]|$ represent the DFT magnitude spectrum of $s(n)$ warped onto the Mel frequency scale. Assuming an N point DFT of the signal, the log-energy within the jth filter bank is given by,

$$e[j] = \log_2\left(\sum_{k=0}^{N-1} w_j[k] \cdot |S_{mel}[k]|\right) \quad \text{for j = 1, 2, … } P$$

Finally, the 12 MFCCs ($c_t[i]$, $i=1..12$) are computed for each excised frame of audio by decorrelating the filter outputs using the discrete cosine transform (DCT),

$$\tilde{c}_t[i] = \sqrt{\frac{2}{P}} \sum_{j=1}^{P} \left(e[j] \cdot \cos\left(\frac{\pi i}{P}(j - 0.5)\right)\right)$$

Finally the cepstral mean is removed from the features. The final 12 MFCCs are given by

$$c_t[i] = \frac{1}{T} \sum_{t=1}^{T} \tilde{c}_t[i]$$

The 12 dimensional vector is augmented with a normalized log-energy component, which is calculated for each frame of data. Finally, the log-energy term is calculated by first taking the log of the sum of the squared data samples. Let $s_t(n)$ represent the nth sample from the tth excised frame of audio. Assuming Ns samples per frame of audio, an initial frame-based energy term is computed as follows,

$$\tilde{e}_t = \log_2\left(\sum_{n=1}^{Ns} s_t^2(n)\right)$$

The energy outputs are normalized to range between –5.0 and +1.0 and are augmented as the 20th feature vector element. Finally, the first and second order derivatives of the 13 features are computed resulting in a final 39-dimensional feature vector.

---

[1] Note that in the time-domain, the resulting signal is given by: $y(n) = s(n) - 0.97s(n-1)$ where $y(n)$ represents the pre-emphasized signal and $s(n)$ represents the input signal.

8.2 Acoustic Model

Most speech recognition systems today model sound units, or phonemes, by a sequence of connected Hidden Markov Model (HMM) states. Speech is analyzed at 100 frames per second and spectral parameters are extracted at each time instant to produce a 39-dimensional feature. This feature representation, known as cepstral parameters, efficiently models the spectral content, frame energy as well as the velocity and acceleration of the audio at each time instant. For each analyzed time frame, t, the HMM-based recognizer is assumed to transition from state i to state j with probability a_{ij} and emit an observation symbol (effectively a 39-dimensional feature) with probability density $b_j(o_t)$. Each phoneme is typically modeled using anywhere between 3 to 5 HMM states depending on the recognizer implementation (SONIC uses 3 states per phoneme to represent the begin, middle, and end-part of the sound unit) and each state is modeled by a weighted set of M multivariate Gaussian distributions. Acoustic modeling in SONIC makes the assumption that the feature elements are independent and therefore able to be modeled using a diagonal covariance matrix rather than a full-covariance matrix. The emission probability, $b_j(o_t)$, therefore becomes,

$$b_j(o_t) = \sum_{m=1}^{M} \frac{w_m}{(2\pi)^{D/2} \sqrt{\prod_{d=1}^{D} \sigma_m^2[d]}} \exp\left\{ -\frac{1}{2} \sum_{d=1}^{D} \frac{(o_t[d] - \mu_m[d])^2}{\sigma_m^2[d]} \right\}$$

where $o_t[d]$ represents the dth feature dimension (D=39) for the tth frame of data. The acoustic model therefore consists of M Gaussians which each have dimension of 39 (i.e., 39 elements to represent the Gaussian mean and 39 elements to represent the diagonal covariance terms, $\sigma^2_m[d]$). Each Gaussian is weighted by a factor, w_m, such that the weights sum to unity.

In order to model speech accurately, HMM states are clustered in various ways depending on their triphone context (the immediate phoneme to the left and immediate phoneme to the right of the current phonetic unit). For a language such as English there are 45^3 possible triphone units. Many of these contexts never appear in the training data and therefore most modern recognition systems use some sort of unsupervised clustering method to obtain a primitive set of about 5000-6000 clustered states. Each state models a unique phonetic quality in the speech data. Typically between 8 and 32 Gaussians ($8 <= M <= 32$) are used to model the state distributions. For a typical speech recognition system with 5000 clustered states and 24 Gaussians per state (each of which are a dimension of 39), the resulting recognition parameters stored as floating point value would require 5000*24*(2*39+1)*(4 bytes/float) = 36.6MB of storage space. Typically the resulting model size is much greater since the triphone contexts modeled by each clustered state must also be stored for lookup at run-time. Many advanced recognition systems build gender-dependent acoustic models which effectively double the storage requirements.

There are several ways of reducing the storage complexity of HMM-based recognizers and the University of Colorado SONIC speech recognition system implements several techniques. Obviously one could reduce the number of HMM-states, reduce the number of Gaussians per HMM state, or even attempt to reduce the dimensionality of the feature representation. All of which result in loss in system accuracy. In our approach HMM model parameters are efficiently stored in 8-bit (1 byte) representation rather than in floating point form. We have found that, when carefully implemented, this quantization reduces the model storage requirements by a factor of 4 without any measurable loss in system accuracy. In SONIC, each HMM feature dimension is scalar quantized to map the range of values for each Gaussian parameter into a quantized 256 element space. At run-time, the models are loaded off of disk and uncompressed as needed back into the floating point representation.

We have also design and developed a method for quantizing all the system Gaussian distributions using Vector Quantization (VQ) methods. For this approach, we gather all Gaussian mean vectors into a pool and quantize the pool of vectors into a set of C code vectors. Typically, around 10,000 code vectors are required to maintain accuracy in a system which originally contains more than 100,000 Gaussian components (recall that for a 5000 state system with 24 Gaussians per state, there will be 120k Gaussian distributions). The VQ method has been designed to allow for separate codebooks for the Gaussian means and covariance matrices, each of which can be optimized for size and accuracy resolution.

Combining approaches, we have demonstrated in the laboratory that an original HMM model of nearly 60MB in size can be compressed to approximately 3MB with no measurable loss in accuracy and even further down to 1MB with some degree of accuracy tradeoff. In summary, the acoustic modeling for the SONIC recognizer consists of decision tree state-clustered continuous density HMMs. The acoustic models have a fixed 3-state topology. Each HMM state can be modeled with variable number of multivariate mixture Gaussian distributions. The acoustic model trainer uses the Viterbi algorithm for model estimation. This substantially reduces the amount of CPU effort needed to train acoustic models compared with forward-backward training methods. The training process therefore consists of first performing state-based alignment of the training audio followed by an expectation-maximization (EM) step in which decision tree state-clustered HMMs are estimated. Acoustic model parameters (means, covariances, and mixture weights) are estimated in the maximum likelihood sense. The training process can be iterated between alignment of data and model estimation to gradually achieve adequate parameter estimation. The final model parameters are efficiently quantized for acoustic model storage.

8.3 Language Model

Support is provided for standard word-based and class-based backoff n-gram language models. Currently unigram, bigram, trigram, and fourgram models can be applied during the first-pass of recognition. SONIC can process language models that have been estimated using both the CMU/Cambridge Statistical Language Modeling Toolkit and the SRI language modeling toolkit. Support for finite-state regular grammar based speech recognition is also provided. For n-gram language models, SONIC uses integer representation for word-id's (therefore SONIC can handle vocabularies exceeding 64k words). Probabilities and Backoff weights for n-gram models are stored in short integer (16-bit) format. We have found that such quantization of the language model parameters (from 32 bit to 16 bit) does not alter the end recognition performance.

8.4 Phonetic Symbol Set

SONIC is capable of using any arbitrary set of phonetic symbols (silence is always assumed to have the symbol `SIL`). Currently, for English we use the 55-phoneme symbol set adopted by earlier versions of the CMU Sphinx-II speech recognizer. The current US English phoneme symbol set is shown below in Table 1.

Phone	Example	Phone	Example	Phone	Example	Phone	Example
AA	father	DX	butter	KD	talk	GD	mug
AE	mad	DH	them	JH	Jerry	SH	show
AH	but	EH	bed	K	kitten	T	tot
AO	for	ER	bird	L	listen	TH	thread
AW	frown	EY	state	M	manager	UH	hood
AX	alone	F	friend	N	nancy	UW	moon
AXR	butter	G	grown	NG	fishing	V	very
AY	hire	HH	had	OW	cone	W	weather
B	bob	IH	bitter	OY	boy	Y	yellow
CH	church	IX	roses	P	pop	Z	bees
D	don't	IY	beat	R	red	ZH	measure
PD	top	BD	tab	S	sonic	SIL	silence
TD	lot	DD	had	TS	bits	br	<breathe>
ls	<lipsmack>	lg	<laughter>	ga	<garbage>		

Table 1: American English phoneme set used by SONIC. Additional symbols are used for breathe, lipsmack, laughter, and garbage events.

8.5 Token-Pass Search Algorithm

Search within SONIC is based on the Token Passing model for speech recognition [14]. The recognizer supports keyword/grammar spotting and large vocabulary continuous speech recognition. Statistical n-gram decoding is accomplished using tree-structured lexicon. In our implementation, tokens are propagated through a static lexical tree (i.e., we do not perform tree copying). The tokens themselves contain links to word history information so that long-span n-gram models can be used. To improve search efficiency, we merge hypotheses that share the same two-word histories. When a token enters the root node of a lexical tree, we predict the best-case value for the trigram probability (by considering the previous two words and all possible follower words that are possible given the tree's root node). Upon exiting a leaf in the tree, we insert the word hypothesis into a lattice and recover the true trigram probability by subtracting the log-probability estimate (from entrance at the root node) with the corrected trigram probability (considering the exact 3 or 4 word sequence of the propagated token). The search utilizes crossword acoustic models in the first pass in an efficient manner. Efficiency is further improved by applying beam pruning. Specialized search beams are associated with the entrance into the root nodes of the tree, nodes located within the tree itself, as well as for states near the leaf of the tree. Histogram pruning is also applied to limit the maximum number of tokens propagated at word-ends (leaf nodes) or globally during search.

The result of the first-pass of n-gram decoding is a word lattice. The word lattice can be converted into a word graph and rescored using alternative language models. In fact, SONIC supports the integration of the Phoenix semantic parser into the second pass search. This allows for word graphs to be converted into concept graphs and for concept n-gram probabilities to be interpolated with word-based language models.

9 Performance on Standard Tasks

SONIC has been benchmarked on several standard speech recognition tasks. Below are the word error rates and real-time factors (on a 2.8 GHz Pentium 4 Xeon). Results obtained from SONIC are comparable to state-of-the-art results obtained in competitive research laboratories.

Task	Vocabulary Size	Word Error Rate	Real-Time Factor
TI-Digits Continuous Telephone Speech	11 Digits (0-9, oh)	0.2% (0.5% first-pass, 0.2% after speaker adaptation)	0.02 x RT
DARPA Communicator Spontaneous Telephone Speech	2.1k	10.9%	0.40 x RT
Wall Street Journal Read Microphone Speech	5k	3.0% (4.2% first-pass, 3.0% after speaker adaptation; wide beams)	0.56 x RT (first-pass only)
Wall Street Journal Read Microphone Speech	20k	8.6% (10.1% first-pass, 8.6% after speaker adaptation; narrow beams)	0.45 x RT (first-pass only)
Switchboard Spontaneous Conversational Telephone Speech	35-40k	36.0% (NIST 1998 Eval)	~3-4 x RT (first-pass only)

Table 2: Word Error rates for several standard speech recognition benchmark tasks using the SONIC speech recognizer.

10 Usage Guide

10.1 Overview

Speech-enabled applications can be built by calling functions within the SONIC API. An example application called `sonic_batch` is provided as a template to building new applications. `sonic_batch` can be used to run speech recognition experiments by simulating batch and live-mode operations. In general, a configuration file is defined by the user in order to describe the experiment setup. The program reads the configuration file and processes audio files while writing output and logging information to external files. The batch-mode program may be executed with the following arguments,

```
sonic_batch -c config.txt [-l]
```

The last argument [-l] can be used to simulate a live-mode interaction. During live-mode, program will send small chunks of audio to the recognizer causing it to perform live speech activity detection and live cepstral mean subtraction on the extracted features.

In order to use the recognizer you will need,

- One or more audio files to process
- A phoneme configuration file which defines the phoneme units in your language
- A language & acoustic model
- A lexicon containing word pronunciations for words contained in the language model

Next, we will discuss the format of each if these inputs and finally describe how the various knowledge sources are specified in the SONIC configuration file.

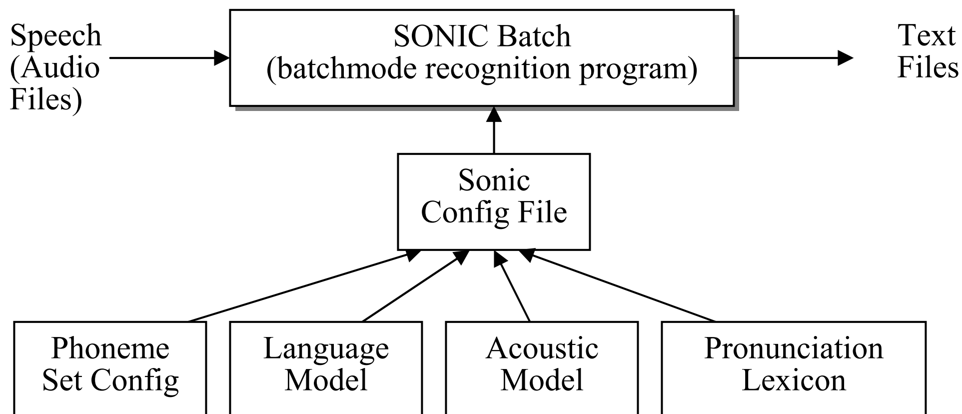


Figure 2: Example inputs required to perform batch-mode recognition in SONIC. A single configuration file is used to setup the recognition job. The SONIC configuration file provides the listings of the decoder settings, phoneme set, language model, acoustic model, and pronunciation lexicon.

10.2 Audio file Format

The recognizer assumes that the input audio is in 16-bit linear PCM format (no header). The audio files should have their bytes swapped in the native format of the machine running the recognizer. If you are working with Microsoft wav files, you can strip the header from the file using the "sox" program for Windows or Unix:

```
sox infile.wav –w –s infile.raw
```

10.3 Phoneme Configuration File Format

This file defines the phoneme set used by the recognizer. The phoneme configuration file begins with an ASCII header and then lists the valid phoneme symbols to be used by the recognizer. The phoneme configuration file ends with the `</phonelist>` marker.

```
<numphones>  55
<phonelist>
AA
AE
AH
...
</phonelist>
```

An example phoneme configuration file for US English is provided here:

```
sonic/2.0-beta5/doc/examples/phoneset.cfg
```

10.4 Statistical N-gram Language Model Format

Support is provided for standard word-based and class-based backoff n-gram language models. Currently up to 4-gram language models can be applied during the first-pass of recognition. SONIC can process language models that have been estimated using both the CMU/Cambridge Statistical Language Modeling Toolkit and the SRI language modeling toolkit.

For class-based language models, a class definition file is required to list which words belong to each defined class. An optional probability for each word given the class can be supplied. A uniform distribution is assumed whenever probabilities are omitted for words within a class.

Language models must first be converted into a compressed binary format before being input into the recognizer. Included is a program called `lm_encode` for generating binary encoded language models. The encoding process takes an ASCII text back-off language model as input and converts it to binary format,

```
lm_encode my_ascii_lm.arpa my_binary_lm.bin
```

Note that for class-based language models, the class-definition file should be provided to the recognizer text format while the word-based back-off LM containing the classes should be compressed into binary format. Several example language models are provided,

`sonic/2.0-beta5/doc/examples/lm.arpa`	Example n-gram language model
`sonic/2.0-beta5/doc/examples/lm.def`	Example class definition file

10.5 Pronunciation Lexicon Format

The lexicon contains the pronunciations for the words in the recognition vocabulary. Words are listed followed by their phoneme sequence. Alternate pronunciations are denoted using open parenthesis followed by an integer alternate number followed by a closing parenthesis. Optional normalized log-probabilities can be assigned to pronunciation variants using bracketed expressions. For example,

```
[0.00000] ACCIDENTAL           AE K S AX D EH N AX L
[-0.22185] ACCIDENTAL(2)       AE K S AX D EH N T AX L
```

denotes that the word "accidental" has 2 pronunciations. The first pronunciation is the most frequently observed in the training data while the alternate is less likely. Note that the log-prob for the most frequent word is normalized to 0 while all the alternates have negative log-probs (less likely to occur. Lexicons can also be constructed without pronunciation probabilities. For example,

```
ACCIDENTAL            AE K S AX D EH N AX L
ACCIDENTAL(2)         AE K S AX D EH N T AX L
```

An example lexicon is provided in,

```
sonic/2.0-beta5/doc/examples/lm.lex
```

For speed of access by the recognizer, one can compress an ASCII text version of the lexicon into a binary file. This can be accomplished using the "`lex_encode`" program provided with SONIC. The program requires the phoneme configuration file (previously described) as input:

```
lex_encode –p phonemeset.cfg –i mylex.txt –o mylex.bin
```

10.6 Acoustic Model Format

The acoustic trainer is used to generate decision tree state clustered HMMs. The trainer outputs binary files containing the clustered-state parameters (e.g., mean vectors, variances, mixture weights, etc.). The files are output by the trainer in files with the name,

`<phoneme>.<context>` for example `AA.f`

where `<phoneme>` represents the base phone for the decision tree, and `<context>` can be either 'l', 'r', 'f', 'n' depending on whether decision tree splitting questions are applied to only the left-context, right-context, full-context, or no-context. A left-context model implies that decision tree questions are only asked with respect to phonemes to the left of the current base phone. Prior to utilizing the final acoustic models, the output models for each base phone, state, and context should be concatenated to produce a single binary model. This can be accomplished using the "`cat`" command in Unix,

```
cat *.? > my_acoustic_model.bin
```

10.7 SONIC Configuration File

A configuration file is used to establish the basic settings of the recognizer for decoding. It is an ASCII text file that has a set of parameters followed by arguments. In general, the SONIC configuration file contains the following items,

- Location of the acoustic model to be used during recognition
- Location of the language model or finite state grammar to use
- Location of the pronunciation lexicon
- (optionally) a pointer to a control file containing a list of audio files to process
- Recognizer settings such as search beams, pruning settings, language model scale factors

At run-time, SONIC loads this configuration file and interprets the file to set it's internal state. Below is a table that describes the possible arguments and values. One example configuration file is provided in,

```
sonic/2.0-beta5/doc/example/sonic.cfg
```

SONIC Configuration File

Option	Required	Typical Values	Description
Batch Processing & Logging Options			
`-batch_file`	Y	filename	An ASCII file which lists the files to be recognized. Each line of the file should contain the filename to be decoded with a unique identifier following it. The identifier is used by the NIST scoring package to uniquely identify the utterance with it's reference transcription. This is a required parameter for "`sonic_batch`", but not required when SONIC is used in server mode, "`sonic_server`".
`-decode_range`	N	<int begin file> <int end file>	Given a batch file, you might want to decode only utterances 23 through 75 in the file. The decode_range option allows one to specify subranges within the control file.
`-output_file`	Y	filename	The output file to which the hypothesis from the decoder will be written. The output format is a string of text followed by the file identifier within parenthesis.
`-log_dir`	N	directory	Output directory to store waveforms and hypotheses from the decoder during batch-mode processing. Output files are numbered 0 through N with a .raw file extension (raw PCM samples). Hypotheses output is written to the log_dir with a .txt extension.
`-expand_compound_words`	N	1 or 0	Set to 1 to convert output words merged with underscore to separate words (e.g., austin_texas → austin texas).
`-output_pronunciations`	N	1 or 0	Set to 1 to output which pronunciation alternative the recognizer selected (e.g., austin(2)).
`-live_mode`	N	1 or 0	Simulate live interaction. Sends samples of audio in small chunks to the recognizer. Cepstral mean normalization applied in real-time rather than computed over the entire signal.
`-align_word_hypothesis`	N	1 or 0	The resulting 1-best hypothesis will be aligned at the HMM-state level using an internal Viterbi aligner. If –log_dir is set, the resulting state-level alignment will be dumped into an ascii file (.sta) to the directory specified by –log_dir. The resulting file can be used for adaptation or retraining.
Phoneme Set and Lexicon Specification			
`-phone_config`	Y	filename	Phoneme set configuration
`-dictionary`	Y	filename	Lexicon of word pronunciations
Acoustic & N-gram Language Model Specification Options			
`-acoustic_mod`	Y	filename	Filename which contains binary encoded acoustic models
`-langmod_file`	Y	filename	Binary encoded version of an n-gram language model

`-lm_class_def`	N	filename	ASCII class definition file for class-based language models. Defines the words in each class and their probabilities.
`-lmctlfn`	N	filename	File containing coded list of language model(s). This function allows the recognizer to load multiple n-gram language models that can be dynamically switched during recognition. Currently, we assume that each LM has the same unigram vocabulary.
`-max_vocabulary_size`	N	(integer)	Used to restrict the size of the vocabulary to Y words (where Y is less than or equal to the number of unigrams in the language model). Decoder will use top-Y words that are most likely based on unigram statistics.
`-filler_file`	N	filename	File containing list of filler words. Fillers are additional words added to the vocabulary (e.g., silence, breath, lipsmack, etc.). Each filler should contain a pronunciation in the user specified lexicon.
`-filler_penalty`	N	-20.0 – 0.0	A log-scale penalty applied to filler words
Feature Extraction Options			
`-sample_rate`	N	8000	Input sampling rate of audio in Hz. Default is 8000.
`-feature_type`	N	MFCC, MFCC_C0 PMVDR	Used to select a valid feature type for decoding. Current features include MFCC, MFCC with 0th cepstrum (MFCC_C0) and PMVDR cepstral coefficients.
`-warp_factor`	N	0.80 to 1.20	Frequency warping factor to be used for vocal tract length normalization
`-fea_apply_var_norm`	N	1 or 0	Apply feature variance normalization to the input feature stream (reads a file containing standard deviation normalization terms that are then applied to the features through division).
`-fea_var_norm_file`	N	(filename)	File containing feature variance normalization factors for each feature dimension. Should contain the standard deviations of the features measures across a session (e.g., a telephone call in the Switchboard corpus).
`-fea_lt_file`	N	(filename)	File containing linear feature transform (matrix plus bias vector) to be applied to the features. This ASCII input file is computed using the CMLLR (cmllr) program (see speaker adaptation section of this manual).
`-cmn_buffer_size`	N	(int)	Number of frames for cepstral mean normalization buffer. Default is 500 frames or 5 seconds.
`-cmn_buffer_file`	N	(filename)	SONIC will store the current estimate of the cepstral mean buffer (12 floating point values) as users utilize the system. This option can be used for initializing the cepstral mean-buffer for live-mode applications.
First-Pass Search Options			

-word_end_beam	N	60-120	Beam threshold computed from the most-likely word-end state at each frame.
-word_entry_beam	N	60-120	Beam threshold for first state in each word
-state_beam	N	120-220	Beam threshold for states after the first state in each word
-lm_scale	N	20-30	Scale factor for language model during first-pass
-max_active_states	N	12000 - 40000	Maximum number of active states allowed at any given time frame.
-max_word_ends	N	50-1000	Maximum number of word-end tokens allowed to propagate at each frame during token passing. Small values impose more constraint and increase speed (with tradeoff in accuracy).
-word_trans_penalty	N	-10	Log-penalty for transitioning from one word to another word during search.
-state_dur_scale	N	1.0	Scale factor applied to the log-probabilities emitted from the state-dependent Gamma duration densities during search.
-short_word_penalty	N	5.0	A log-scale penalty applied to words with 3 phonemes or less.
-use_only_monophones	N	1 or 0	If set to 0, decoder will only use context-independent monophones for decoding.
-min_log_prob	N	-125	Used to constrain the minimum log-probability emitted from any state. Similar to smoothing observation probability with a uniform distribution.
-pronounce_scale	N	40	Scale factor for pronunciation probabilities if incorporated into the lexicon
-online_adapt	N	1 or 0	Perform incremental adaptation single regression class MLLR. SONIC will incrementally estimate a transform matrix between utterances and apply the current transform to the subsequent utterance (can be used with the sonic_batch example application when statistical n-gram LMs are used).
-fastmatch	N	1 or 0	Enable fast-match search for n-gram based recognition
-fastmatch_beam	N	10.0 - 15.0	Beam threshold for fast-match. Smaller values result in tigher beams and faster decoding at the expense of higher word error rates.
-dynamic_lm	N	1 or 0	When using the language model control file (-lmctlfn) multiple LMs can be loaded into memory provided they each have the same vocabulary. SONIC will maintain a history buffer of recognized words and will select the language model with the highest probability to decode the next utterance. This option is used for reading-tracking for speaker-adaptation or for interactive books.
-crossutt_word_history	N	1 or 0	When set to 1, SONIC will retrain the last N words from the previous utterance and initialize the search for the next utterance by initializing tokens with <s> or

			with the previous recognized words as possible word histories. Therefore, the words from the current utterance might be recognized as a continuation of words from the previous utterance.
Second-Pass Search, Lattice Processing & Logging Options			
`-lattice_rescore`	N	1 or 0	Set to 1 to invoke lattice rescoring (rather than computing the maximum likelihood path from the first-pass search)
`-rescore_lm_scale`	N	10 - 30	Language model scale factor for second pass search
`-posterior_rescoring`	N	1 or 0	Rescores the word-graph using word-level posterior probabilities computed from the graph
`-lattice_dir`	N	(directory)	Directory to read / write lattice files out to
`-dump_lattice`	N	1 or 0	set to 1 to write lattices into files. Lattices are written in AT&T FSM format
`-dump_model_marked_lattice`	N	1 or 0	Writes a state-level aligned lattice to the directory specified by –log_dir
`-read_lattice`	N	1 or 0	set to 1 to read lattices from files
`-lattice_prune_beam`	N	100-300	Beam threshold for paths in the lattice. Used to prune the lattice.
N-Best List Generation			
`-nbest`	N	N desired	Compute N-best alternative hypotheses
`-nbest_dir`	N	Directory	Directory to output the N-best lists
Confidence Options			
`-confidence`	N	1 or 0	Tag words with confidence. Confidence is output from 0 to 1 using word-graph posterior probabilities
`-confidence_lm_scale`	N	1.0	Language model scale used for confidence annotation
`-confidence_am_scale`	N	10.0 to 30.0	Inverse of Acoustic Model scale used in confidence annotation. Ideally should be equal to the language model scale factor used in the first-pass of decoding.
`-filler_delete_penalty`	N	-10	Penalty of filler words deleted for parsing (used only if concept language models are applied).
Grammar Recognition & Keyword Spotting Options			
`-keyword_spot`	N	1 or 0	Put the recognizer into keyword spotting mode. This should be used with grammar based speech recognition (-network_name option).
`-keyword_threshold`	N	0.0 to 1.0	Threshold for detecting / rejecting spotted words. If this option is set, you must set –confidence to 1 to enable computation of word confidence for keyword thresholding.
`-network_search`	N	0 or 1	Perform recognition using a finite-state regular grammar (setting –network_name value will automatically set –network_search to a value of 1).
`-network_name`	N	Filename	File containing finite-state regular grammar

Concept Language Modeling Options			
-concept_lm_rescoring	N	0 or 1	Enable concept language modeling in second pass
-clangmod_file	N	(file)	Specify a single concept-based language model.
-clmctlfln	N	(file)	Specify a control file that allows the recognizer to load multiple concept-based language models. Each concept language model is assigned a unique name. LMs can be dynamically selected during runtime (for dialog state dependent concept language modeling).
-clm_scale	N	(float)	Scale factor for concept language model
-frames_file	N	(filename)	File containing list of concepts
-grammar_dir_name	N	(directory)	The directory in which the stochastic context free grammar (SCFG) file resides.
-dict_file	N	(filename)	The name of the SCFG dictionary. It includes all the words used in the grammar.
-grammar_file_name	N	(filename)	The name of the SCFG file.
-glm_scale	N	(float)	Scale factor for SCFG model
-int_wt	N	(float)	Interpolation factor between word/class based language model and concept language model (1.0 = CLM only)
Voice Activity Detection and Push-to-Talk Options			
-auto_end_point	N	1 or 0	Automatically detect the begin / end of speech activity
-end_point_padding	N	125	Number of milliseconds of silence to pad signal with after speech activity end-point detection.
-vad_sil_to_speech_penalty	N	-3.0	Penalty for transitioning from silence to speech
-vad_speech_to_sil_penalty	N	-3.0	Penalty for transitioning from speech to silence
-vad_min_speech_frames	N	6	Number of frames classified as speech before begin of speech is assumed to start
-vad_min_silence_frames	N	12	Number of frames classified as silence before end of speech is detected
-vad_num_speech_mixtures	N	4	Number of top-N weighted speech mixtures per context-independent monophone that are used to construct the internal speech Gaussian Mixture Model for classification
-vad_sil_penalty	N	0.0	Log-prob penalty for detecting silence. Negative values (-20 → -10) make speech detection easier, while positive values encourage the system to detect silence.
-push_to_talk	N	0 or 1	Enable push-to-talk mode. Applications interacting with this configuration must call functions such as “decoder_set_push_to_talk” to set the status of the push-to-talk button. Enabling push_to_talk allows applications to force detection of speech and override VAD decisions through user keypress.

Table 3: SONIC configuration file options.

10.8 Output Format

The output format of the batch-mode application (`sonic_batch`) is structured for use with the NIST Sclite Scoring Package (`http://www.nist.gov/speech`). Sentence hypotheses are listed one at a time with the sentence identifier surrounded by parentheses. For example,

```
OCTOBER TWENTY THIRD (sls-20000628-003-003)
LATE MORNING AFTER NINE (sls-20000628-003-004)
```

11 Acoustic Model Training

Acoustic model training consists of three basic steps: feature extraction, Viterbi state-based alignment, and model estimation. The outcome of the training process is a set of decision-tree state-clustered Hidden Markov Models. The last two steps (alignment, model estimation) are iterated until speech recognition errors are minimized on a development test set. In general, you will need a set of audio files (16 bit linear PCM sampled audio with no header) and associated text-based transcriptions along with a pronunciation dictionary (for languages other than English). The process of retraining the acoustic models is described in the next sections. A block diagram of the process is shown in Figure 3.

11.1 Data Preparation

It is assumed that your audio files are stored in 16-bit linear PCM audio format in the byte-swapped format of the machine used for acoustic training. Audio files are assumed to not contain headers (Microsoft wav files have a 44 byte header typically). If you are working with Microsoft wav files, you can strip the header from the file using the "sox" program for Windows or Unix:

```
sox infile.wav –w –s infile.raw
```

Each audio file should have a corresponding word-level text transcription. All punctuation (commas, semicolons, etc.) should be removed from the transcriptions prior to training since the alignment utility does not perform text normalization.

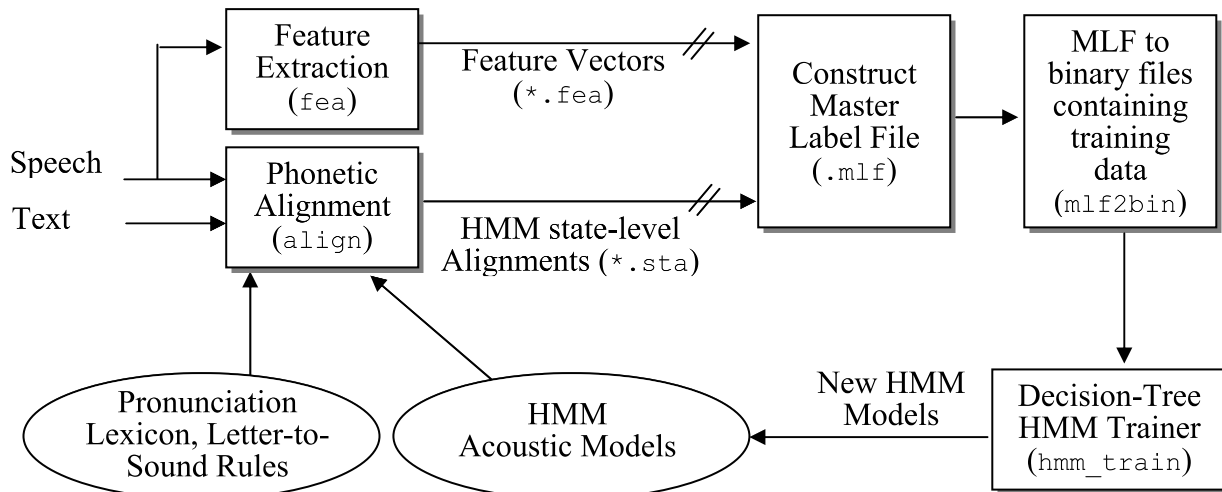


Figure 3: Block diagram of the acoustic model training process. The steps include (1) feature extraction and phonetic alignment of speech files with provided text transcription. During this phase an initial acoustic model is used to provide the Viterbi force-alignment at the HMM state-level. The feature vectors and corresponding state alignments are grouped into a single Master Label File (MLF). This text file is then

processed to generate a single binary data stream (`mlf2bin`) prior to acoustic model training (`hmm_train`). Finally, the new acoustic models are used to realign the training data and the process is repeated (sequentially obtaining improved data alignment and more accurate acoustic models).

11.2 Feature Extraction

A feature extraction routine is provided to extract the standard 39-dimensional feature vectors consisting of 12 Mel Frequency Cepstral Coefficients (MFCCs) and normalized frame energy. The 13 static parameters are augmented with their first and second order derivatives to give a resulting 39-dimensional floating-point vector (100 vectors are extracted for every second of audio). The included program fea is used to extract features:

fea [options] speechfile.raw featurefile.fea

Option	Required	Description
`-f <int>`	N	Input audio sampling rate in Hz (default is 8000 Hz)
`-o <int>`	N	Output audio sampling rate for features in Hz (can be less than input rate)
`-b <int>`	N	Cepstral Mean buffer size in frames. If this option is set, the cepstral mean will be estimated over a fixed circular buffer size of N frames.
`-l <file>`	N	Batch-mode feature extraction. The file input argument is an ASCII file containing an input/output pair on each line separated by space or tabs. The feature extraction module will read each audio file listed and output the corresponding features to the output file. This option can be used with the -b flag to simulate live-mode feature extraction. If this option is used, the final fea arguments of an input and output mfc file can be ignored.
`-feature_type <string>`	N	Sets the internal feature type to be output. Current valid feature types include MFCC, MFCC_C0. Default is MFCC feature type.
`-w <float>`	N	Frequency warping factor for use in Vocal Tract Length Normalization. Default is 1.0 (standard frequency scale). Typical values range from 0.88 to 1.12 for most applications using VTLN.
`-a`	N	Output feature vectors in ASCII format

Table 4: `fea` program command-line usage

For example, if you with to extract features for a 16 kHz sampled audio file, you can simply use,

```
fea -f 16000 speechfile.raw featurefile.fea
```

If you are using 16kHz sampled audio but wish to build acoustic models for 11kHz data, one can specify the output sampling rate for the features,

```
fea -f 16000 –o 11025 speechfile.raw featurefile.fea
```

11.3 State-Based Alignment of training data

At the heart of the acoustic model training process is the Viterbi-based phonetic aligner (`align`). This program takes raw audio files with associated text transcriptions as input and produces a time-aligned representation of the data (i.e., an association between frames of input audio to words, phonemes, and HMM-states). Accurate alignment of the audio data is key to training high-quality acoustic models. The phonetic aligner is user configurable and extendable while providing support for multiple languages.

Phonetic alignment requires knowledge sources: (1) an acoustic model and (2) a pronunciation lexicon. We have designed the aligner to be user definable and configurable. The configuration of the aligner is found in,

```
sonic/2.0-beta5/config/aligner.cfg
```

The configuration file for the aligner contains 2 basic tags. These are `<LEXICON>` and `<CONFIG>`. The `<LEXICON>` tag is used to provide specifications for access of phonetic dictionaries, phoneme sets and letter-to-sound modules. In general, this tag is used by a program called (`sspell`) to allow users with a handy access to phonetic pronunciations. We will discuss this tag shortly. The `<CONFIG>` tag is used to specify the behavior of the aligner when processing audio files. An example tag set for US English is shown below:

```
<CONFIG>
   <TAG>      english-si-microphone
   <LANGUAGE> english
   <GENDER>   si
   <SAMPRATE> 8000
   <TYPE>     microphone
   <PHONESET> english/eng-phoneset.cfg
   <MODEL>    english/microphone-8kHz-i.mod
   <FEATURE>  PMVDR
   <LEXICON>  english/eng-lex.bin
   <PHONEMAP> none
   <LET2SND>  english/eng-lex.lts
   <DEFAULT>  true
</CONFIG>
```

In this example, an aligner configuration called "english-si-microphone" has been defined. The language and gender tags are currently not of much importance to the aligner, but will be utilized in future versions of SONIC. The example configuration specifies that the speaker-independent acoustic models are located in (`sonic/2.0-beta5/config/english/microphone-8kHz-i.mod`). The pronunciation lexicon is specified by the `<LEXICON>` subtag. In this example, the lexicon has been compressed into a binary searchable file using the (`lex_encode`) program which is part of SONIC. One can specify a letter-to-sound model which can be trained for any new language or lexicon. We will discuss letter-to-sound modules later in this document. The `<PHONEMAP>` tag is used for bootstrapping SONIC to a new language. This option will be discussed in a tutorial file provided with SONIC ("SONIC Tutorial: Porting to a New Language").

Proper installation of the align program requires that one set the "`ALIGN_BASE`" environment variable in Unix or Windows to point to the directory in which SONIC has been installed. Please refer to the installation section of this document to ensure that the aligner is properly set up. The commandline usage of the align program is provided in Table 5.

```
align [options] speechfile.raw
```

Option	Required	Description
		Input Options
-bat <file>	N	Batch-mode alignment. Batch-mode input file should contain command-line options (one complete command-line per line of the file). The aligner will read this file and process one file at a time as specified in the command-line parameters.
-t <file>	Y	File containing text-transcription to align. Can contain embedded phoneme sequences using the "!" character followed by the phoneme name.
-iw <file>	N	File containing reference input word alignments. If this option is provided, the aligner will force word boundaries to occur at user defined locations. Consequently only the phonemes and state-level alignments will be altered with respect to the hand-labeled word alignments.

-sil	N	Automatic silence detection and insertion. Uses a general heuristic of 170 msec pause between words before detecting and labeling it.
-lts	N	Enable the use of letter-to-sound prediction if a module exists in the aligner configuration.
-f <float>	N	Sampling rate of audio file in Hz.
-audio_rate <float>	N	Set the sampling rate of the input audio file in Hz. Note that this is similar to the option –f
-v	N	Verbose output
-n <int>	N	Number of states to remain active in the search at each time-frame. Default is 60.
-con	N	Use context-dependent acoustic models (use in conjunction with -mod option)
Feature Specification Options		
-var_norm <file>	N	Optional: a file containing variance normalization scaling terms for the features used in alignment. To be used in conjunction with acoustic models trained using variance normalization. This ASCII file contains the standard deviations of each of the feature components. Variance normalization is applied by dividing the features by their standard deviations.
-warp_factor <float>	N	Frequency warping factor for use in Vocal Tract Length Normalization. Default is 1.0 (standard frequency scale). Typical values range from 0.88 to 1.12 for most applications using VTLN.
Built-In Model Selection Options		
-m <tag>	N	Typing "align" from the Unix or Windows commandline will display a set of default configurations of the aligner which are specified in (sonic/2.0-beta5/config/aligner.cfg). Users can select a pre-defined configuration for ease of program use.
User-defined Model Selection Options		
-phone_config <file>	File	Specify a user defined phone set configuration file.
-phone_map <file>	File	Specify a phoneme mapping file. This file is used to map phonemes in a new language to phonemes in an existing language.
-model <file>	N	User defined binary acoustic model as built from the model training process.
-model_rate <float>	N	Set the sampling rate of the input acoustic models (used in conjunction with the –mod option to specify the acoustic models and the model's sampling rate). This option allows the aligner to operate on 22kHz sampled audio while utilizing acoustic models trained at a lower sampling rate (e.g., 8kHz, 16kHz, etc.)
-feature_type	N	Sets the internal feature type. Current valid feature types include MFCC, and MFCC_C0. Default is MFCC feature type.
-lex_file	N	User specified lexicon
-lts_file	N	User specified letter-to-sound model
Output Options		
-pro	N	Output which pronunciation was selected during alignment.
-ow <file>	N	File for output word-level alignments. Contains begin & end sample, word
-op <file>	N	File for output phone-level alignments. Contains begin & sample, and phoneme
-os <file>	N	File for output state-level alignments. Contains begin/end frame for each state followed by phoneme. This file is used during acoustic model training.

Table 5: align program command-line usage

By default, SONIC is shipped with telephone acoustic models for US English (male/female/child) and telephone acoustic models (male/female). By typing, "`align`" from the Unix command-line, the program will display the set of configurations that are available through the `-m` command-line option. For example,

```
Built-In Model Selection Options
-----------------------------------------------
-m     [type] select aligner configuration.

       Available configurations include:

           english-si-telephone
           english-female-telephone
           english-male-telephone
           english-child-microphone
           english-si-microphone (* default)
           english-male-microphone
           english-female-microphone
```

The US English configurations of the SONIC aligner utilize the freely available 125,000-word pronunciation dictionary from Carnegie Mellon University (CMU). During alignment, the align program will first check to see if you have a personalized dictionary. Personalized dictionaries are specified in Unix by setting the environment variable "`ALIGN_USER`" to point to an ASCII lexicon of pronunciations. For example, in the C-shell environment,

```
setenv ALIGN_USER /home/users/pellom/dictionary.txt
```

here the lexicon file might contain entries such as,

```
ACCIDENTAL           AE K S AX D EH N AX L
ACCIDENTAL(2)        AE K S AX D EH N T AX L
```

For the Microsoft Windows environment, the following environment variable can be set,

```
SET ALIGN_USER=C:\home\users\pellom\dictionary.txt
```

In general, if you are starting with a new task domain (for example, let's consider a telephone recorded phrase from the travel domain), you will first get an initial alignment of the data using the existing microphone based acoustic models provided as part of the aligner. So, let's say our audio file was "`example.raw`" with transcription "`example.txt`". The text transcription file contains the phrase,

```
!SIL UH I'D LIKE TO GO FROM DENVER TO SAN DIEGO ON FRIDAY MARCH SIXTEENTH !SIL
```

Alignment would be performed using these options[2],

```
align -t example.txt -ow example.wrd -op example.phn -os example.sta -v
-m english-si-telephone -sil -f 8000 example.raw
```

This example command-line configures the aligner to process the audio file `example.raw` using `example.txt` as the primary transcription (note that `!SIL` is placed around the utterance to force the aligner to insert silence on either end of the audio file). The outputs consist of word-level (.wrd), phone-level (.phn) and state-level (.sta) alignments. The (`-m english-si-telephone`) option configures the aligner to use

---

[2] Data files for this example can be found in: `sonic/2.0-beta1/doc/examples/aligner`

speaker-independent telephone speech models. Finally the `-sil` option configures the aligner to automatically detect and insert silence into the alignment. Running this command would produce the following output,

```
(1) !SIL:  SIL
(1) UH:  AH
(1) I'D:  AY DD
(1) LIKE:  L AY KD
(3) TO:  T UW,  T IX,  T AX
(1) GO:  G OW
(3) TO:  T UW,  T IX,  T AX
(1) SAN:  S AE N
(1) DIEGO:  D IX EY G OW
(1) ON:  AA N
(2) FRIDAY:  F R AY DX IY,  F R AY D EY
(1) MARCH:  M AA R CH
(2) SIXTEENTH:  S IX K S T IY N TH,  S IH K S T IY N TH
(1) !SIL:  SIL

speech parameterization...
automatic silence option on
alignment in progress...
ave. log probability : -31.04595
```

The state-alignment output (.sta file) is used to retrain new acoustic models. However, the aligner also outputs the word-level and phoneme-level alignment of the data. Using the tool "`xs.tcl`" which is part of the Snack Sound Toolkit (`http://www.speech.kth.se/snack`), we can view the word-level alignments,

```
xs.tcl example.raw example.wrd
```

to produce a graphical display as shown in Figure 4. It is a very good idea to inspect some of the word-level alignments using a graphical display tool such as snack before proceeding onto the next step. ***Tip:*** *If you are certain that your speech audio file has silence both before and after speech activity, then insert !SIL into the transcription (see above example) in order to force the aligner to recognize silence in these locations.*

The last example illustrated how to align files. Often you might have an existing set of acoustic models trained for the task at hand that are tuned in some sense to better align your data. You can tell the aligner to use your own acoustic models by (1) adding a new `<CONFIG>` entry into the aligner configuration file (`sonic/2.0-beta5/config/aligner.cfg`), or (2) via the command-line using the (`-mod`) switch. The built-in US English models are context-independent monophone models. To obtain even more accurate alignment of your data, one can align using context-dependent models using the (`-con`) switch. For example, assuming you have trained a set of acoustic models that have been binary encoded with the filename "`comm-male.mod`",

```
align -mod comm-male.mod -model_rate 8000 -feature_type PMVDR -con -t
example.txt -ow example.wrd -op example.phn -os example.sta -v -sil -f 8000
example.raw
```

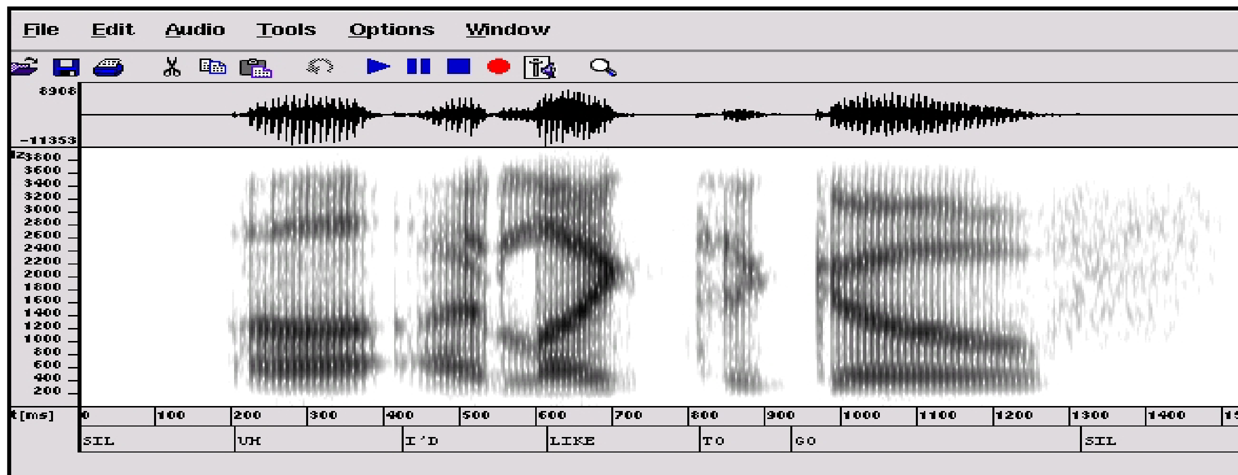


Figure 4: Snack Sound Toolkit Graphical display of aligner word-level output

It is relatively expensive in terms of file I/O to load your context-dependent acoustic models into memory to align one audio file at a time. To alleviate this problem, the aligner has a batch-mode capability. Using batch-mode, the aligner can be executed simply by using this command-line,

```
align -bat batch.txt
```

where `batch.txt` contains a sequence of command-lines to be executed on your data set. For example,

```
-t example1.txt -ow example1.wrd -op example1.phn -os example1.sta -v
      -m english-si-telephone -sil -f 8000 example1.raw
-t example2.txt -ow example2.wrd -op example2.phn -os example2.sta -v
      -m english-si-telephone -sil -f 8000 example2.raw
-t example3.txt -ow example3.wrd -op example3.phn -os example3.sta -v
      -m english-si-telephone -sil -f 8000 example3.raw
-t example4.txt -ow example4.wrd -op example4.phn -os example4.sta -v
      -m english-si-telephone -sil -f 8000 example4.raw
```

The main result from the alignment process is a set of Viterbi state-based alignments of your training data set. In the next section, we create a "Master Label File" which specifies the training feature vectors and state alignments in one file. The acoustic trainer uses the Master Label File to generate the acoustic models.

11.4 Master Label File Specification

The next step is to construct a file that lists the training feature file name followed by state-level alignments. We call this file the "Master Label File" (MLF). The MLF provides information regarding the location of the feature files on the hard disk, the phoneme sequence and state-level assignments of the data as well as whether the phoneme is at the beginning ("b"), middle ("m") or end ("e") of a word. For example,

```
/home/rec/segment/com/cu/read_speech/20010621/000/sls-20010621-000-002.fea
   0    1    2   17   18   19 SIL b
  20   20   21   21   22   22 D   b
  23   25   26   30   31   32 AE  m
  33   39   40   41   42   44 L   m
  45   46   47   49   50   51 AX  m
```

```
  52   54   55   66   67   68 S   e
  69   78   79  139  140  144 SIL b
.
/home/rec/segment/com/cu/read_speech/20010621/000/sls-20010621-000-003.fea
   0   14   15   15   16   16 SIL b
  17   22   23   23   24   28 F   b
  29   29   30   32   33   35 R   m
  36   43   44   47   48   51 AE  m
  52   53   54   54   55   58 N   m
  59   64   65   73   74   76 S   e
  77   98   99  123  124  128 SIL b
.
```

Each entry in the MLF contains a feature file (`.fea`) followed by the file's state-level alignment as output from the alignment tool. In this example (first file), the phoneme "`AE`" state 1 consists of frames 23 through 25, state 2 consists of frames 26 through 30 and state 3 consists of frames 31 and 32. From the MLF, the training code can determine which frames are assigned to which states of the acoustic model. It also knows the phonetic context in which the frames appear (for example, `AE` preceded by `D` and succeeded by `L`). Each entry of the MLF is separated by a period (.). MLFs can be constructed using some very simple Unix C-shell code. For example assuming there are many files with .sta and .fea extensions,

```
rm -rf my.mlf
touch  my.mlf
foreach f (`ls -1 *.sta`)
   set mfc = {$f:r}.fea
   echo $mfc >> my.mlf
   cat $f >> my.mlf
   echo "." >> my.mlf
end
```

Master label files may also contain an optional floating point value after the 'b','m','e' phone position field to denote the relative weight of the features during training. This functionality is used to support future machine learning techniques such as Boosting. The training system currently supports weighted training.

11.5 Data Extraction from Master Label File

The acoustic trainer is based on decision-tree state-clustered Hidden Markov Models (HMMs). State-based alignments of the training data determined using the provided Viterbi aligner are first converted into binary feature files for training (using `mlf2bin`). The decision tree trainer (`hmm_train`), is then used to internally construct the decision trees and compute mixture Gaussians for each leaf node within the tree. The clustered states for each base phone and then assembled into the final acoustic model for recognition. The next sections describe how to extract the data for training (using `mlf2bin`) and how to build the acoustic models (using `hmm_train`).

Acoustic model training consists of two basic steps. First, we extract features into temporary files (one temp file for each state of each base phone). A utility called "`mlf2bin`" is included for this operation and has the following command-line structure,

mlf2bin [options]

Option	Required	Default	Description
`-c <file>`	Y	-NA-	Phoneme Configuration File
`-s <int>`	N	3	Number of states
`-d <int>`	N	39	Feature vector length

`-w <float>`	N	1.0	Weight for this data source (can be used to increase the contribution of the data when multiple corpora are used to train acoustic models)
`-l <file>`	Y	-NA-	Master Label File
`-D`	N	Current directory	Output directory for temporary files (these can be huge!!). Defaults to using the current directory for output temp files.
`-C`	N	-NA-	Maximum number of frames per state to extract from master label file. Useful for controlling the amount of disk space and runtime RAM needed to train models. Can result in some loss of training data if not carefully selected. Defaults to using all available training data.

Table 6: `mlf2bin` program command-line usage

The output of the mlf2bin routine is a set of temp files that are then used to train the acoustic models.

11.6 HMM Model Estimation

The acoustic trainer uses the mlf2bin processed Master Label File to associate frames of training data to clustered hidden Markov model states. The trainer can be considered a "Viterbi"-based trainer since the frame-to-state alignments are considered fixed during each iteration of the Expectation-Maximization EM algorithm. Actually, we assume here that the alignments are fixed during the entire training process and realign the data using each newly constructed set of acoustic models. The trainer code estimates models for one base phone at a time. Consequently, the trainer can be loaded in parallel on a multi-processor machine for improved training performance. The trainer utility is called "`hmm_train`" and has the following command-line structure,

Option	Required	Default	Description
			Standard Options
`-c <file>`	Y	-NA-	Phoneme Configuration File
`-r <file>`	Y	-NA-	File containing decision tree rules
`-F <int>`	Y	-NA-	Minimum frame count to split a state (controls model size)
`-L <float>`	Y	-NA-	Minimum likelihood change to split a state (controls model size)
`-s <int>`	N	3	Number of states
`-d <int>`	N	39	Feature vector length
`-t <directory>`	Y	-NA-	Directory containing mlf2bin output files
`-o <directory>`	Y	-NA-	Output directory for resulting acoustic models
`-P <string>`	Y	-NA-	Basephone to train. Should be a phoneme listed in the phoneme configuration file
`-PL <file>`	N	-NA-	Rather than providing a single basephone to train using the –P option, users can train models for a set of phonemes using the –PL option. The input file contains the basephone name to train (one per line). This option is typically used for distributed acoustic training.
`-i <int>`	N	10	Number of EM iterations desired
`-a`	N	-NA-	Adapt models using supplied MLF and baseline set of acoustic models located in (-o) output directory. If this option is set, only the mean vectors and mixture weights will be estimated from the training data.
`-p <int>`	N	39	Train acoustic models by estimating mixture occupation likelihoods from first N feature vector elements.

`-e <float>`	N	1.0	Raise likelihoods during training to the Nth power.
`-f <int> <int> <int>`	N	50 6 24	Frames per mixture, minimum mixtures, maximum mixtures per state. This option is used to dynamically set the number of mixtures in each state based on the amount of available training data.
`-v <float>`	N	0.01	Variance threshold. Limit variances to this minimum value during training. Avoids singularities in acoustic models.
`-C`	N	left, right, full, all, none	Apply decision tree questions to either the left-context only, right-context only, full-context, or do not apply questions at all (results in a simple context-independent monophone acoustic model)
Distributed Training Options			
`-D <mode option>`	Y	client / server	This option establishes the process as either a client or server. For server mode operation a temporary directory for training must be provided (e.g., -D server /home/tmp). For client mode operation, a file containing a list of servers to connect to must be provided (e.g., -D client serverlist.txt). This is a required field for distributed training.
`-N <port number>`	N	6500	Default port number for hmm_train in server mode
`-j <string>`	N	Tag	User assigned job tag (for tracking jobs). This is an optional parameter in distributed training
`-m <int>`	N	1,2,3,…	Number of CPUs on the server machine to be used for training. For dual CPU machines, set to 2. The server mode trainer will fork itself out as needed to accomplish the training task.
`-T <file>`	N		In server mode, list of trusted clients (for remote commands like “reset”)
`-A <file>`	N		Client machines to accent jobs from. If this file is not provided, the server will accept jobs from any client [including those from machines outside your network unless you have a firewall].

Table 7: `hmm_train` program command-line usage

During the training process, single-mixture triphones are estimated for each triphone occurrence in the training data. The data is then placed at the root node of the decision tree and splitting questions are evaluated. The question that results in the largest increase in likelihood for the training data is used to split the node. The splitting process continues until the likelihood change falls below a threshold or the number of frames assigned to the clustered state becomes too small. Finally, the data assigned to each leaf node in the tree is then used to estimate the mixture-Gaussian distributions. The HMM parameters (mean vectors, variances, mixture weights, and clustering information) are then written to disk.

The decision tree rules can be specified in an ASCII file. Rules are denoted by a rule-name (a ‘$’ is put before each rule name) followed by a list of phonemes or rules that are included. For example,

```
$nasal            M N NG
$voiced_stop      B D G
$unvoiced_stop    P T K
$stop             $voiced_stop $unvoiced_stop
```

An example comprehensive set of decision tree questions for US English are found in,

```
sonic/2.0-beta5/doc/examples/dt.rules
```

Note that every phoneme specified in the phoneme set configuration file should appear at least once in the decision tree question set.

11.7 Generating the Final Acoustic Model

The trainer outputs binary files containing the clustered-state parameters (e.g., mean vectors, variances, mixture weights, etc.). The files are output by the trainer in files with the name,

`<phoneme>.<context>` for example `AA.f`

where `<phoneme>` represents the base phone for the decision tree and `<context>` can be either 'l', 'r', 'f', 'n' depending on whether questions are applied to only the left-context, right-context, full-context, or no-context. A left-context model implies that decision tree questions are only asked with respect to phonemes to the left of the current base phone. Prior to utilizing the final acoustic models, the output models for each base phone, state, and context should be concatenated to produce a single binary model. This can be accomplished using the "cat" command in Unix,

```
cat *.*-* > my_acoustic_model.bin
```

At this point, the resulting binary model file will be ready for use within the SONIC recognizer.

11.8 Advanced Feature: Distributed Computer HMM Training

It is well-known that the estimation of hidden Markov model parameters (acoustic training) is a computer-intensive task. To deal with computational issues, SONIC implements Viterbi styled training in which frames of data are explicitly assigned to hidden Markov model states using the Viterbi algorithm. As mentioned in the previous sections, this process iterates between time-alignment and decision tree training. To further reduce training time, SONIC implements a client/server distributed training architecture. In this mode of operation a single client instance of `hmm_train` is loaded on a central machine containing the training data (extracted using the mlf2bin routine). Many machines (including Windows, Linux, Sun, and Mac OS X machines) can be applied to train acoustic models in parallel.

The procedure for distributed training is as follows:

- The program `hmm_train` is loaded in server mode on several computers. The server mode instance of `hmm_train` can be loaded by any user on the machine and assumes that users can freely assess TCP/IP ports on those machines. The server is started on a machine using,

  ```
  hmm_train -N $portnum -D server $tempdir -m $numcpu -A $clientlist.txt
  ```

  where `$portnum` represents the port on the server machine where the server will receive training requests (default port is 6500) and `$tempdir` represents a temporary directory for storing intermediate results during training. The temporary directory is assumed to exist and be readable and writable by the user who loads the server instance. The server can also be loaded with a `-m` option to specify the number of CPUs on the server machine that can be used for training. In addition, users can limit the set of clients which can submit jobs to the server by providing the `-A` option with a file containing a list of client machines.

- Once servers are loaded on several machines, the client instance can be executed to train the acoustic models using the following command line,

  ```
  hmm_train -D client $serverlist.txt $HMM_OPTS
  ```

where `$serverlist.txt` is a text file containing the names of server machines which can accept training processes (one machine name per line). The `$HMM_OPTS` signifies the standard training options which should be applied to build the acoustic models. Typically training in distributed mode utilizes the –PL option to specify a list of phonemes to train in addition to the –C all option that allows the client machine to distribute training of all phonetic contexts to all available servers.

12 Keyword Spotting and Grammar Recognition

SONIC supports first-pass recognition using finite state grammars expressed in terms of a regular expression. The format of the regular expressions is similar to that used by the Cambridge HTK recognizer. Regular expressions contain words, expansion rules, and meta characters,

`|`	alternatives
`[ ]`	optional expressions
`{ }`	zero or more repetitions
`< >`	one or more repetitions
`%`	used to denote words pronounced as individual phonemes

Below is an example grammar for command and control. Note that the overall grammar network is specified by enclosed parenthesis. This example grammar accepts utterances containing silence only as well as commands surrounded by silence (e.g., “turn on the tv”, “go to channel twenty two).

```
$digit    = one | two | three | four | five | six | seven | eight | nine;
$teens    = ten | eleven | twelve | thirteen | fourteen | fifteen | sixteen |
            seventeen | eighteen | nineteen;
$tens     = twenty | thirty | forty | fifty | sixty | seventy | eighty | ninety;
$channel  = $digit | $teens | $tens [$digit];
$device   = tv | dvd | vcr | stereo;
$power_state  = on | off;
$cmd      = turn $power_state [the] $device | $device $power_state |
                  [go] [to] channel $channel;
 ( < $cmd | $filler > )
```

Figure 5: Example finite-state regular expression grammar in SONIC

In SONIC, the grammar term `$filler` is a reserved symbol to represent a network containing all monophone units. Grammar-based recognition can be evoked by using the following configuration settings when using the `sonic_batch` or `sonic_server` application,

```
-network__name          <filename>
```

where `<filename>` is the name of an ASCII file containing the network to use. An example grammar is provided in the directory `sonic/2.0-beta5/doc/examples/example.gram`. A typical use of grammar recognition is to perform keyword and phrase spotting. A simple grammar for spotting zero or more of the words “apples”, “oranges” and “pears” is shown below:

```
( < apples | oranges | pears | $filler > )
```

The filler network ($filler) will provide some rejection capabilities. However, further rejection can be gained by estimating the confidence of each recognized keyword and thresholding the word based on confidence score. This can be done using the following configuration settings,

```
-confidence             1
-keyword_spot           1
-keyword_threshold      $T
```

where `$T` is a floating point value ranging from 0 to 1. The larger the threshold, the greater the keyword rejection rate. Confidence calculation is described in the next section.

13 Word-Level Confidence Annotation

Confidence annotation is performed by converting the internal word-lattice into a graph structure. Word-posterior probabilities are then computed directly from the word-graph using the forward-backward algorithm. Details of this approach can be found in [13]. Words are tagged with confidence by including the "-confidence 1" option in the decoder configuration file. With this option set, the decoder converts the word-lattice into a word graph and computes word-posterior probabilities for each word found in the best-path search. Filler words are assigned a confidence of 0 since they are skipped during confidence processing. The confidence for each word is assigned into the returned hypothesis data structure (see API section for details). The confidence scoring can be controlled using "`-confidence_lm_score`" and "`-confidence_am_score`" settings. These control the scalings used to weight the language model and acoustic-model scores during confidence calculation. In fact, "`-confidence_am_score`" is the inverse of the power factor used to normalize the acoustic model scores. It should be set to be roughly equal to the language model scale factor used in the first-pass of decoding.

14 Adapting to Speaker and Environment

Speaker adaptation algorithms are provided within SONIC as executable command-line programs. These programs are compiled within the directory (`sonic/2.0-beta5/src/adapt`). SONIC supports vocal tract length normalization (VTLN), cepstral mean and variance normalization, Maximum Likelihood Linear Regression (MLLR), and Maximum a Posterior Linear Regression (MAP-LR).

14.1 Vocal Tract Length Normalization (VTLN)

The recognizer supports the integration of vocal tract length normalized acoustic models into acoustic search [9,l0,11]. This is implemented by warping the frequency scale during feature extraction. The basic process for incorporating VTLN normalized models consists of the following:

Training:

1. Align data and train standard multiple-mixture triphone acoustic models for the task

2. Train single-mixture (possibly gender dependent) versions of the models in step (1).

3. For each speaker in the training set, determine the VTLN frequency-warping factor that maximizes the likelihood of the training speaker's data acoustic the models trained in step (2). Command-line usage for `vtln` is provided below. Determine the warp factor that maximizes that speaker's data and extract features at that warp scale. Repeat for all speakers in the training set.

4. Using the VTLN-normalized data determined in step (3), retrain standard multiple-mixture acoustic models for the task. These are referred to as VTLN-normalized models.

Decoding: (for each test speaker):

1. Decode speaker's utterances using standard acoustic models (step 1 from training) and frequency warp factor of 1.0 (no warp). Get initial hypothesis for each utterance.

2. Align data, create master label file, and determine best frequency warp for the test speaker using the `vtln` routine (similar to step 3 above).

3. Re-decode speaker's utterances using VTLN-normalized acoustic models and frequency warping factor determined in step 2.

vtln program command-line usage

Option	Required	Description
-c <file>	Y	Phoneme set configuration file used to train the acoustic models
-l <file>	Y	Master label file for adaptation data (contains the feature file names and state alignments)
-i <file>	Y	Binary encoded HMM acoustic model for the task
-f <float>	Y	Sampling rate of the raw audio data in Hz
-b <float>	Y	Lower bound of warp factor (typically set to 0.88)
-e <float>	Y	Ending bound of warp factor (typically set to 1.12)
-s <float>	Y	Step-size for testing various frequency warp factors
-o <file>	Y	Output file for displaying the results. Results are appended to this file.
-feature_type <string>	Y	Type of feature vector to compute. Possibilities include: MFCC, MFCC_C0, RC, RC_C0 for Mel Frequency Cepstral Coefficient vectors or Root Cepstral Coefficient Vector types.

Table 8: `vtln` program command-line usage

The VTLN program described above actually simply calculates the total path score for the data given the alignments provided using the (-l) option. Here is an example Unix C-shell script that illustrates how VTLN is applied (assuming you have calculated the state-alignments of your data and placed them into a file called "mlf"). Note here that the MLF for VTLN computation consists of a raw audio filename followed by the state-based alignment of the data,

```
# Construct Master Label File from RAW audio file and state-alignment (either
# output from the recognizer or obtained by performing state-alignment using
# "align" on the 1-best hypothesis output from the recognizer.

rm -rf $ vtln vtln-mlf
touch vtln vtln-mlf

foreach f (`ls -1 *.sta`)
   echo {$f:r}.raw >> vtln-mlf              # .raw file is the audio file
   cat $f >> vtln-mlf                       # .sta file is the state-alignment
   echo "." >> vtln-mlf
end

# Compute the best VTLN warp factor for 8kHz sampled audio.  Test warps ranging
# from 0.88 to 1.12 with a step increment of 0.02.  Use the HMM model
# $vtln_test_model which contains single Gaussian mixture per HMM state.  The
# feature type is MFCCs.  Output is written to the file VTLN

vtln -c phoneset.cfg -f 8000 -b 0.88 -e 1.12 -s 0.02 -l vtln-mlf \
     -i $vtln_test_model –feature_type MFCC -o vtln
set warp = `cat vtln`;
echo "Best VTLN warp was found to be $warp"
```

14.2 Maximum Likelihood Linear Regression (MLLR)

An implementation of MLLR based adaptation has been provided with SONIC. MLLR is based on estimation of a linear transformation matrix (or class of matrices) that transform the mean vectors of the system Gaussians based on a set of adaptation data. The purpose of MLLR is to estimate a transformation matrix ***A*** such that the next system mean vector is given by,

$$\hat{\mu} = A\mu$$

Where ***A*** is an (nx1) by n matrix. Technical details regarding MLLR are given in [12]. The MLLR source code is contained in the directory,

```
sonic/2.0-beta5/src/adapt
```

The main executable is "`mllr`" and takes the following arguments,

mllr program command-line usage

Option	Required	Default	Description
`-c <file>`	Y	-NA-	Phoneset configuration file
`-l <file>`	N	-NA-	Master label file with state-based alignments of the adaptation data
`-L <file>`	N	-NA-	For Lattice-based MLLR, this file contains a list of feature files followed by model-marked lattice files (fea file, lat file, fea file, lat file, …). The lattice files are obtained from the recognizer using the –dump_model_marked_lattice option. Lattices are logged by the recognizer and are marked with HMM-state level aligments and word-posterior scores.
`-I <int>`	N	1	Number of EM iterations used to estimate the MLLR transform matrices
`-i <modelfile>`	Y	-NA-	Input binary encoded HMM model file
`-o <modelfile>`	Y	-NA-	Output binary encoded HMM model file
`-m <file>`	N	-NA-	MLLR class configuration file
`-M <float>`	N	-NA-	Perform MAP reestimation of the model mixture mean vectors. The floating value input is tau which controls the contribution of the MLLR adapted mean vector compared to that of the adaptation data.
`-v`	N	0	Set to 1 to apply variance adaptation to the data
`-d`	N	-NA-	Vector length used to estimate state occupation probabilities. Can be less than the model vector length

Table 9: `mllr` program command-line usage

The MLLR adaptation code takes as input a binary encoded HMM model, a phoneset configuration file (same format as used in the decoder), and a specification of the state-level alignments of the adaptation data. Currently there are 3 methods for generating state-level alignments:

- Method 1: Log the audio files (.raw) and 1-best hypothesis with confidence to a directory and use the "align" routine to generate the state-level alignments (-sta option). In this method, the input to the MLLR routine (-l option) is a master label file as defined in Section 12.4.

- Method 2: Generate the state-level alignments with confidence during decoding. This is accomplished in the example program sonic_batch by adding the (`-align_word_hypothesis`) option into the configuration file. When a `-log_dir` logging directory is specified, the state level alignments (`.sta`

files) are automatically generated by the recognizer at run-time. In this method, the input to the MLLR routine (`-l` option) is a master label file as defined in Section 12.4.

- Method 3: Generate the state-level alignments using model-marked lattices. In this method, the configuration option (`-dump_model_mark_lattices`) is set to 1. Lattices are output into the logging directory and contain state-level alignments of the audio file along with word-posteriors. This method results in *Lattice-based MLLR* adaptation. For this method, one should use the `-L` option to specify the lattices and extracted features. The file input for this option contains feature file / lattice file pairs (feature file on line one, lattice file on line two, second feature file on line three, second lattice file on line four, etc.).

For all methods listed above, each audio file is assumed to have been processed by the feature extraction module (fea routine). In general, MLLR is applied iteratively by recognizing the input data (usually several utterances from a single speaker). With the state alignments and extracted features, the MLLR routine estimates improved model parameters. The new models are used to obtain an improved hypothesis of the data, improved state-alignments, etc. Typically 2-3 iterations of MLLR are required before the output hypothesis converges to a best-estimate. Over iterating MLLR can cause performance degradations.

MLLR can operate on the adaptation data using one or more regression classes. Classes can be defined using a configuration file specified using the (-m option). An example MLLR configuration file contains,

```
<num_mllr_classes> 3
AA    1
AE    1
Aw    2
AX    3
…
```

In this example, 3 regression matrices will be computed. The phoneme AA and AE are assigned to class 1 while AW and AX are assigned to classes 2 and 3 respectively. The MLLR code can also be used to estimate a global variance scaling term for each feature component. By using the `-v` switch in the `mllr` routine, the following variance scaling term is estimated,

$$R_l = \frac{\sum_{t=1}^{T}\sum_{s=1}^{S}\sum_{m=1}^{M}\gamma_{s,m,t}(o_t)\left(\frac{\left(o(t)-\hat{\mu}_{s,m,l}\right)^2}{\sigma^2_{s,m,l}}\right)}{\sum_{t=1}^{T}\sum_{s=1}^{S}\sum_{m=1}^{M}\gamma_{s,m,t}(o_t)}$$

where T is the number of adaptation tokens, S is the number of states, M is the number of model mixture components. The term γ() denotes the probability of the *t*th adaptation token being observed in the *m*th mixture of the *s*th state of the HMM model. After the update, the model parameter variances are scaled using,

$$\hat{\sigma}^2_{s,m,l} = R_l \sigma^2_{s,m,l}$$

14.3 Constrained Maximum Likelihood Linear Regression (CMLLR)

CMLLR estimates a linear transform in which the same transform is applied to both the means and variances of the system parameters. CMLLR has the advantage of being applicable to the feature-space rather than the model space. Hence, the computational overhead is reduced to a simple feature-space transform. SONIC implements the feature-space implementation of CMLLR. Using a set of adaptation data the linear transform and bias vector can be computed. The new observation is computed as,

$$\hat{o}_t = Ao_t + b$$

Where ***A*** is an (nxn) matrix and b is a bias vector. Technical details regarding CMLLR are given in [22]. The CMLLR source code is contained in the directory,

```
sonic/2.0-beta5/src/adapt
```

The main executable is "`cmllr`" and takes the following arguments,

cmllr program command-line usage

Option	Required	Default	Description
`-c <file>`	Y	-NA-	Phoneset configuration file
`-l <file>`	N	-NA-	Master label file with state-based alignments of the adaptation data
`-L <file>`	N	-NA-	For Lattice-based MLLR, this file contains a list of feature files followed by model-marked lattice files (fea file, lat file, fea file, lat file, …). The lattice files are obtained from the recognizer using the –dump_model_marked_lattice option. Lattices are logged by the recognizer and are marked with HMM-state level aligments and word-posterior scores.
`-I <int>`	N	1	Number of EM iterations used to estimate the CMLLR transform matrices
`-i <modelfile>`	Y	-NA-	Input binary encoded HMM model file
`-o <modelfile>`	Y	-NA-	Output model transform file
`-t`	N	-NA-	Apply transform to features from Master label file (-l) or lattice file (-L)

Table 10: `cmllr` program command-line usage

The CMLLR adaptation code takes as input a binary encoded HMM model, a phoneset configuration file (same format as used in the decoder), and a specification of the state-level alignments of the adaptation data. As in MLLRl, there currently there are 3 methods for generating state-level alignments:

- Method 1: Log the audio files (.raw) and 1-best hypothesis with confidence to a directory and use the "align" routine to generate the state-level alignments (-sta option). In this method, the input to the CMLLR routine (-l option) is a master label file as defined in Section 12.4.

- Method 2: Generate the state-level alignments with confidence during decoding. This is accomplished in the example program sonic_batch by adding the (`-align_word_hypothesis`) option into the configuration file. When a `–log_dir` logging directory is specified, the state level alignments (`.sta` files) are automatically generated by the recognizer at run-time. In this method, the input to the CMLLR routine (`-l` option) is a master label file as defined in Section 12.4.

- Method 3: Generate the state-level alignments using model-marked lattices. In this method, the configuration option (`-dump_model_mark_lattices`) is set to 1. Lattices are output into the logging directory and contain state-level alignments of the audio file along with word-posteriors. This method results in *Lattice-based CMLLR* adaptation. For this method, one should use the `–L` option to specify the lattices and extracted features. The file input for this option contains feature file / lattice file pairs (feature file on line one, lattice file on line two, second feature file on line three, second lattice file on line four, etc.).

Typically we estimate a single CMLLR transform for each training speaker and adapt the features of this speaker using the –t option. Once all speakers have been transformed, a new canonical acoustic model is

trained. This procedure is known as Speaker Adaptive Training (SAT). During decoding, we must estimate the CMLLR transform for the input speaker. First, we obtain a first-pass hypothesis using non-SAT models. Then, we estimate the CMLLR transform using the state alignments from the recognizer (see discussion above). Finally, we transform the features during recognition using the following configuration option,

```
-fea_lt_file <cmllr-transform-file>
```

Below we illustrate an example of computing the CMLLR transform from a training speaker's audio data,

```
# Construct Master Label File (mlf) from state-alignment (either directly
# output from the recognizer or obtained by performing state-alignment using
# "align" on the 1-best hypothesis output from the recognizer).

rm -rf $ mlf
touch mlf

foreach f (`ls -1 *.sta`)
     set raw = {$sta:r}.raw               # assume raw audio file is directory
     fea -f $samprate $raw {$sta:r}.fea   # extract features
     echo {$sta:r}.fea >> mlf
     cat  $sta         >> mlf
     echo "."          >> mlf
end

# Estimate CMLLR transform and apply transform to features
# the resulting ASCII file (cmllr.transform) will contain the linear
# feature-space transform estimated for the speaker

cmllr –c $phoneset –l mlf –i $acoustic-model –t –o cmllr.transform
```

14.4 Structured Maximum a Posterior Linear Regression (SMAPLR)

In some situations lack of adaptation data can lead to poor performance for the MLLR adaptation algorithm. For large amounts of adaptation data, the use of fixed regression classes may not provide optimal performance either. Because of these two drawbacks to MLLR we have implemented the MAP-LR algorithm proposed in [16] and [17] to allow for MAP based adaptation using dynamic regression class trees. The input/output of the `tree_maplr` routine is similar to that of MLLR with the exception of the parameters that control the number of regression classes generated during adaptation. The number of dynamically generated regression classes is controlled using the `–T` and `–s` option

tree_maplr program command-line usage

Option	Required	Default	Description
`-c <file>`	Y	-NA-	Phoneme set configuration file
`-l <file>`	N	-NA-	Master label file with state-based alignments of the adaptation data
`-L <file>`	N	-NA-	For Lattice-based MLLR, this file contains a list of feature files followed by model-marked lattice files (fea file, lat file, fea file, lat file, …). The lattice files are obtained from the recognizer using the –dump_model_marked_lattice option. Lattices are logged by the recognizer and are marked with HMM-state level aligments and word-posterior scores.
`-I <int>`	N	1	Number of EM iterations used to estimate the MLLR transform

			matrices
-T <int>	N	1750	Frame count threshold for regression classes
-s <int>	N	1750	Silence count threshold
-r <float>	N	1.0	Regularization parameter
-i <modelfile>	Y	-NA-	Input binary encoded HMM model file
-o <modelfile>	Y	-NA-	Output binary encoded HMM model file
-M <float>	N	-NA-	Perform MAP reestimation of the model mixture mean vectors. The floating value input is tau which controls the contribution of the MLLR adapted mean vector compared to that of the adaptation data.
-v	N	0	Set to 1 to apply variance adaptation to the data
-d	N	-NA-	Vector length used to estimate state occupation probabilities. Can be less than the model vector length

Table 11: `tree_maplr` program command-line usage

14.5 Cepstral Variance Normalization

An implementation of session-side cepstral variance normalization is provided within the SONIC toolkit. The objective of cepstral variance normalization is to provide some robustness of the parameters to noise and channel by normalizing features to have unit variance. Given a list of feature vectors extracted from a speaker-session (e.g., an entire telephone conversation), the function `varnorm` can be used to compute the feature-dependent standard deviations (the normalized features are computed by subtracting the cepstral mean and dividing by the expected value of the standard deviation). For example,

```
varnorm –l featurefilelist.txt –o output_stddev.txt –d 39
```

The last parameter is the feature vector length. The input file (featurefilelist.txt) contains a list of MFCC files extracted by the user over a session. The output file will contain the standard deviations that can be used for recognition. Note that the input features are normalized by this code and rewritten back to disk normalized by the global standard deviation of the data. Variance normalization can be incorporated into the recognizer by adding two additional lines into the recognizer's configuration file,

```
-fea_apply_var_norm  1
-fea_var_norm_file   stddev.txt
```

In this example, the recognizer would apply feature variance normalization and expect to read the normalization terms (feature standard deviations) from the file stddev.txt.

15 Voice Activity Detection

An HMM model-based voice activity detection module has been developed within the SONIC architecture. At run-time, the recognizer constructs two interconnected single-state Hidden Markov Models (also known as Gaussian Mixture Models) to characterize speech vs. silence acoustics. Since components from the acoustic models provided for recognition are also used for voice activity detection, the recognizer can take direct advantage of speaker adapted acoustic models during speech detection and also the entire statistical knowledge base of the trained acoustic models.

The silence model is constructed at run-time by collapsing the mixtures from the available silence and filler HMM models into a single mixture Gaussian state. Currently, phonemes with the following labels are

considered for the silence model: SIL, ga, br, ls, lg (silence, garbage, breathe, lipsmack, laughter). A single-state speech model is constructed by extracting the top N mixture components (by mixture weight) from the states of HMM models representing speech units. By default, the top 4 mixture components are used from each speech HMM state to construct a global speech model.

The two resulting speech / silence HMM states are interconnected as shown in Figure 4 with log-based penalties (A) – (C) applied for transitioning from silence to speech and from speech to silence. The voice activity detection parameters and their configuration options are, (A) `-vad_sil_penalty` , (B) `-vad_sil_to_speech_penalty` , (C) `-vad_speech_to_sil_penalty`. The default values for these parameters are 0.0, -3.0, and –3.0 respectively. Setting `-vad_sil_penalty` to smaller values will encourage the detection of silence. Smaller values of the `-vad_speech_to_sil_penalty` will make detection of the end of speech region more difficult. While larger values of `-vad_sil_to_speech_penalty` will make the detection of the transition from silence to speech more difficult.

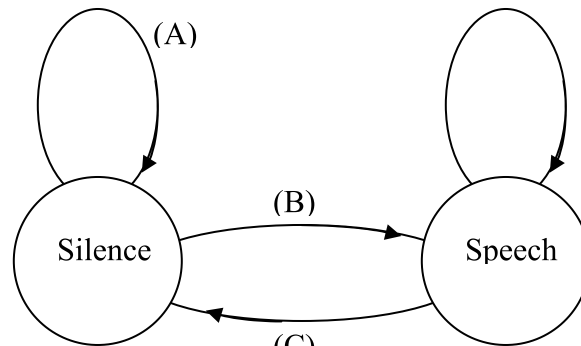


Figure 6: HMM-based voice activity detector.

16 Fast-Match Search Algorithm

Fast-match is a well-known technique for improving the run-time efficiency of large-vocabulary speech recognition systems. The fast-match algorithm in SONIC is based on computing the likelihoods of context-independent monophone HMM models from a look ahead window of 4 frames. Specifically, the probability of observing features at time `t+1` through `t+4` is computed for each context-independent HMM state (3 states) of each phoneme. The maximum log-probability for each phoneme state is determined over the look ahead window. Phonemes whose likelihood scores fall below a user defined beam are labeled as “inactive”. The beam is set based on subtracting a search beam from the highest scoring phoneme model. During search, HMM state transitions into “inactive” phonemes are not considered. This includes transitions into the root nodes of the lexical tree as well as transitions from active phones into inactive phones within the lexical tree. The fast-match technique is similar to that implemented in the CMU Sphinx-II speech recognition system.

Fast-match can be enabled for n-gram based speech recognition tasks. The fast-match module is activated through the SONIC config file by setting (`-fastmatch 1`) and by establishing the fast-match beam threshold. For example (`-fastmatch_beam 15.0`). The source implementation to the fastmatch algorithm can be found in (`src/decoder/fastmatch.c`). For a 20k word Wall Street Journal speech recognition task, fastmatch was found to improve the decoding speed by 20% while resulting in only a 5% relative increase in word error rate.

17 Letter-to-Sound Prediction Algorithm

For most large vocabulary speech recognition tasks, one encounters words in which there does not exist any pronunciation in the hand-made lexicon. Often these occur for proper names (e.g., Pellom, Hacioglu), street names (e.g., Arapahoe, ColFax, etc.), or from words that are fictionally derived (e.g., 'Hogwarts', and 'Hagrid' from the Harry Potter series of books). In SONIC, we have implemented a decision tree based system for prediction of sound units from letter sequences. Our implementation is based on the algorithms described in [19,20]. SONIC provides two such programs (`t2p_fea`) for feature extraction from a pronunciation lexicon and (`t2p_train`) to learn the letter-to-sound mappings from the extracted features. The outcome of the t2p (text-to-phone) software is a decision tree model that can be integrated into the phonetic aligner of SONIC.

The letter-to-sound module operates by aligning each letter within an example input pronunciation lexicon to corresponding phoneme units (or possibly epsilon, or no phoneme). Details of the alignment procedure are provided in [19]. For example, consider the following words from the US English pronunciation lexicon with associated letter-to-sound alignment,

```
ROADWAY     R OW _ D W EY _
RODDING     R AA _ DX IX _ NG
```

The letter-to-sound module extracts vectors of features from the input lexicon consisting of the center letter plus 3 letters of surrounding context and attempts to match the input to a desired phoneme output. For example, consider the following feature vectors for the words "roadway" and "rodding" from above:

```
_ _ _ R O A D R
_ _ R O A D W OW
_ R O A D W A _
R O A D W A Y D
O A D W A Y _ W
A D W A Y _ _ EY
D W A Y _ _ _ _
_ _ _ R O D D R
_ _ R O D D I AA
_ R O D D I N _
R O D D I N G DX
O D D I N G _ IX
D D I N G _ _ _
D I N G _ _ _ NG
```

Here the center letters have been marked in italics while the desired output phoneme has been marked in bold-face type. Based on vectors of input as shown above a decision tree is constructed for each letter of the alphabet (i.e., each possible center letter). Given the letter context surrounding the center letter (L3,L2,L1,R1,R2,R3), questions are asked related to the letter context (e.g., "Is L3 an 'R'?"). At each node in the decision tree, the question that leads to the largest reduction in entropy is considered as the best splitting question for the tree node. Once the tree is constructed, letters can be mapped to phonemes by asking questions at each position in the tree until a leaf node is reached. At the leaf node, the probability of each possible phoneme can be determined. Currently we select the best possible phoneme sequence, but in future versions of SONIC we will generate an N-best representation at output.

Training the decision tree requires two programs (`t2p_fea`) for feature extraction and (`t2p_train`) to build the decision tree. The command-line usage for both programs is shown in Tables 12 and 13.

t2p fea

t2p_fea [options]

Option	Required	Default	Description
-i <file>	Y	-NA-	Input ASCII pronunciation lexicon
-p <file>	Y	-NA-	Phoneme set configuration file for input lexicon
-t <file>	N	-NA-	Use a pre-trained decision tree model to align data
-a <file>	Y	-NA-	Dump letter-to-sound alignments to a file
-o <file>	Y	-NA-	Output training vector file (this is the input to t2p_train)
-c	N		Treat the lexicon as case sensitive for the purposes of building letter-to-sound mappings. Default is case insensitive features.

Table 12: `t2p_fea` program command-line usage.

t2p train

t2p_train [options]

Option	Required	Default	Description
Training Options			
-i <file>	Y	-NA-	Input feature vectors for training
-m <int>	N	1	Minimum feature vector count to split a node
-v	N	-NA-	Verbose output during training
-d <int>	N	50	Maximum depth of decision tree
-e <float>	N	0.02	Minimum change in entropy required to split tree node
-c	N	-NA-	Design resulting decision tree classifier to be letter case-sensitive
-o <file>	Y	-NA-	Output Decision Tree File
Testing Options			
-t <file>	N	-NA-	Input file of words for testing
-T <file>	N	-NA-	Input trained decision tree
-O <file>	N	-NA-	Output hypothesis for test words specified in –t option

Table 13: `t2p_train` program command-line usage

In general, one can train the decision tree using the following example,

```
t2p_fea –i input_lexicon.txt –p phoneset.cfg –o input_lexicon.fea
t2p_train –i input_lexicon.fea –o input_lexicon.lts
```

The tree can be tested by providing a file containing a list of words to predict. For example,

```
t2p_train –t testwords.txt –T input_lexicon.lts –O testwords.hyp
```

The accuracy of the tree can be determined using the NIST sclite scoring utility. For US English, we constructed a decision tree using 90% of the words in the CMU dictionary utilizing the phoneme set from SONIC. We constructed a test set from the remaining 10% of words. The phoneme error rate for US English was found to be 9.8%. For languages such as French and German for which there is a closer correspondence between letters and sounds, the error rates are substantially lower.

Once the decision tree is trained, it can be used within the phonetic aligner (`align`) and phonetic pronunciation extractor (`sspell`) in SONIC. The integration can be accomplished by editing the file, (`sonic/2.0-beta5/config/aligner.cfg`) and inserting the decision tree filename into the `<LET2SOUND>` tag set. For example, to use letter-to-sound prediction in the `sspell` program for the US

English lexicon, we can augment the entry with the letter-to-sound decision tree (`eng-lex.lts`). A similar entry can appear within the `<CONFIG>` tags to allow the aligner to utilize the trained letter-to-sound mapping.

```
<LEXICON>
   <LANGUAGE> english
   <PHONESET> english/eng-phoneset.cfg
   <LEXICON>  english/eng-lex.bin
   <LET2SND>  english/eng-lex.lts
   <DEFAULT>  true
</LEXICON>
```

18 Client-Server and Distributed Recognition Interface

18.1 Server Implementation

An initial implementation of a live-mode client/server interface to SONIC with speech compression support is provided as an example of how to access the API of the recognition engine. The C implementation of the server code can be found in,

```
sonic/2.0-beta5/src/app/sonic_server.c
```

The server waits for connections over a socket port. The server is written in C using the API of SONIC (see section 20). An example client is provided as a Tcl/Tk script using the Snack programming language. The server takes four possible arguments. First, a server configuration can be loaded using the (-C) option. Users must provide the port number for the server to listen for connection on. The port number is specified using the (-p) option. Here is an example command-line which will bring SONIC up in server mode,

```
sonic_server –p 5555 –C server.cfg
```

The server configuration file (`server.cfg`) can be specified using the (–C) option and providing a configuration file. There are two types of configurations, "acoustic-model" and "configuration". Examples of both are shown below for the purposes of illustration,

```
<CONFIG>
   <TAG>          wsj
   <TYPE>         configuration
   <NAME>         Wall Street Journal
   <DESCRIPTION>  Read Newspaper Text
   <FILE>         sonic-test/wsj.cfg
   <LANGUAGE>     American English
   <SAMPRATE>     16000
   <VOCABSIZE>    5000
</CONFIG>

<CONFIG>
   <TAG>          wsj-male
   <TYPE>         acoustic-model
   <NAME>         WSJ - Adult Male
   <FILE>         sonic-test/models/wsj-16k-m.mod
   <LANGUAGE>     American English
   <SAMPRATE>     16000
```

```
   <PHONESET>     demo/wsj/phoneset.cfg
</CONFIG>

<CONFIG>
   <TAG>          wsj-female
   <TYPE>         acoustic-model
   <NAME>         WSJ - Adult Female
   <FILE>         sonic-test/models/wsj-16k-f.mod
   <LANGUAGE>     American English
   <SAMPRATE>     16000
   <PHONESET>     demo/wsj/phoneset.cfg
</CONFIG>
```

Within each configuration of type with the label "`<TYPE> configuration`" one can specify the settings and parameters of SONIC. When selected, SONIC will execute and initialize itself under the settings and prepare for input audio to arrive. Here is an example configuration for a 5k word dictation task,

```
-filler_file            demo/wsj/wsj.filler
-phone_config           demo/wsj/phoneset.cfg
-langmod_file           demo/wsj/wsj.lm
-dictionary             demo/wsj/wsj.lex
-acoustic_mod           models/wsj-16k-i.mod
-word_end_beam          65.0
-word_entry_beam        160.0
-state_beam             160.0
-lm_scale               30.0
-rescore_lm_scale       30.0
-word_trans_penalty     -12.5
-state_dur_scale        2.0
-short_word_penalty     -10.0
-sample_rate            16000.0
-lm_garbage_collect     1
-max_active_states      12000
-auto_end_point         1
-end_point_padding      125
-max_word_ends          175
-filler_penalty         0.0
-live_mode              1
-vad_min_silence_frames 35
-vad_min_speech_frames  35
-push_to_talk            1
```

Here, the file "wsj.filler" is a file containing the filler phoneme for silence, e.g., "<SIL>". The pronunciation lexicon "wsj.lex" contains the words and their pronunciations (including the filler word pronunciations such as the entry "<SIL>   SIL"). The binary compressed language model is "wsj.lm". The phoneset configuration file is provided "phoneset.cfg" along with the binary acoustic model file, "wsj-16k-i.mod".

18.2 Client Implementation

The Tcl/Tk client assumes that Tcl/Tk 8.3 or higher and Snack version 2.1 or higher is installed on the client machine (Snack can be obtained from `http://www.speech.kth.se/snack`). The client graphical interface is located in the following directory,

```
sonic/2.0-beta5/src/app/sonic-client.tcl
```

The client software currently assumes that the acoustic models trained at a 16kHz rate (one can edit the `sonic-client.tcl` program to change this default behavior). The software has the current visual interface,

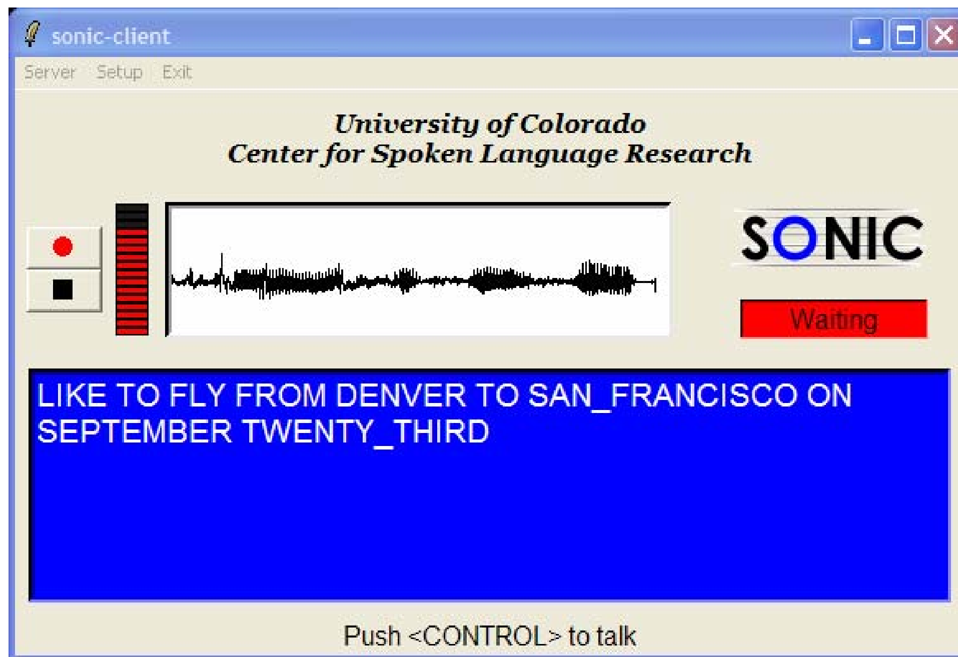


Figure 7: Graphical Tcl/Tk client interface to SONIC

In general, users first select "server" and then "connect". Connections are established by typing in the hostname and port of the sonic server. Once a connection is established, users are prompted with a setup for a task domain to work with. The example dialog box for selecting a configuration is shown below. Note that these entries are provided to the client from the server:

Once a configuration is selected, the user can press the record button. Audio will begin to display in the upper window of the screen. *To send audio to the speech recognition server tap the "Ctrl" button on your PC. When you are finished speaking tap "Ctrl" (control) to stop the recording.* This is the "push-to-talk" mode of the interface. Partial hypotheses will be displayed during speaking within the blue window in Figure 7. The settings of the recognizer can be changed during recognition by selecting any of the setup options (parameters which can be modified during recognition include beam settings, voice activity detection settings, language modeling scaling factors, etc).

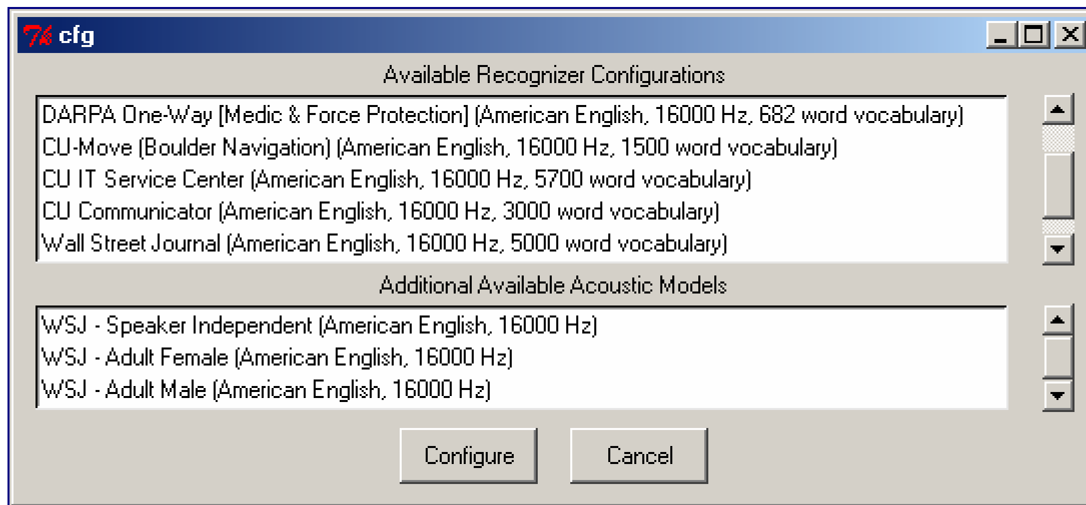


Figure 8: Configuration Selection dialog box

18.3 Speech Compression Client Interface

SONIC integrates the voice compression system developed by Jean-Mark Valin known as Speex 1.1.3. Speex is a fully open source voice compression system based on subband CELP coding (see `http://www.speex.org`). It has the ability to compress speech from 2-44kbps. In general, 16kHz raw PCM audio has a bit rate of 31.25 kB/s. For congested data networks, such a data rate can lead to less-than-realtime speech recognition performance. We have integrated the Speex into the general compiled libraries of SONIC. The Speex libraries and functions are located in,

```
sonic/2.0-beta5/src/voip
```

Based on example programs for encoding and decoding speech using this compression codec, we have developed an API interface for streaming audio. The interface description is found in (`sonic/2.0-beta5/src/voip/speex_api.h`). To illustrate the use of speech compression as a means for lowering the bitrate needed to communicate between a client and server application of SONIC, we have developed a Voice over IP (VoIP) compression server which converts raw PCM sampled audio to Speex encoded bitstreams. SONIC, when loaded into server mode using the `sonic_server` program, can detect and uncompress incoming Speex bitstreams prior to recognition.

The speex compression server is implemented in,

```
sonic/2.0-beta5/src/app/voip_server.c
```

The Tcl/Tk client graphical interface illustrates how to interact with the server. In general use, the client user will load the voip_server on their local machine by executing the command,

```
voip_server -p 7000
```

where `-p 7000` refers to the port on to which the server will listen for connections. Once the Tcl/Tk graphical client is loaded, users can access the services of the compression system by selecting, "Setup" followed by "VoIP Settings". The dialog box shown in Figure 9 will appear. The interface allows one to control the level of speech compression. Users can select the host and port for the compression server along with settings related to the Speex engine (see `www.speex.org` for documentation details).

Currently the `voip_server` is a single connection server (one user can connect to one server instance at a time). We intend to implement multiple connection servers for this piece of software in the future.

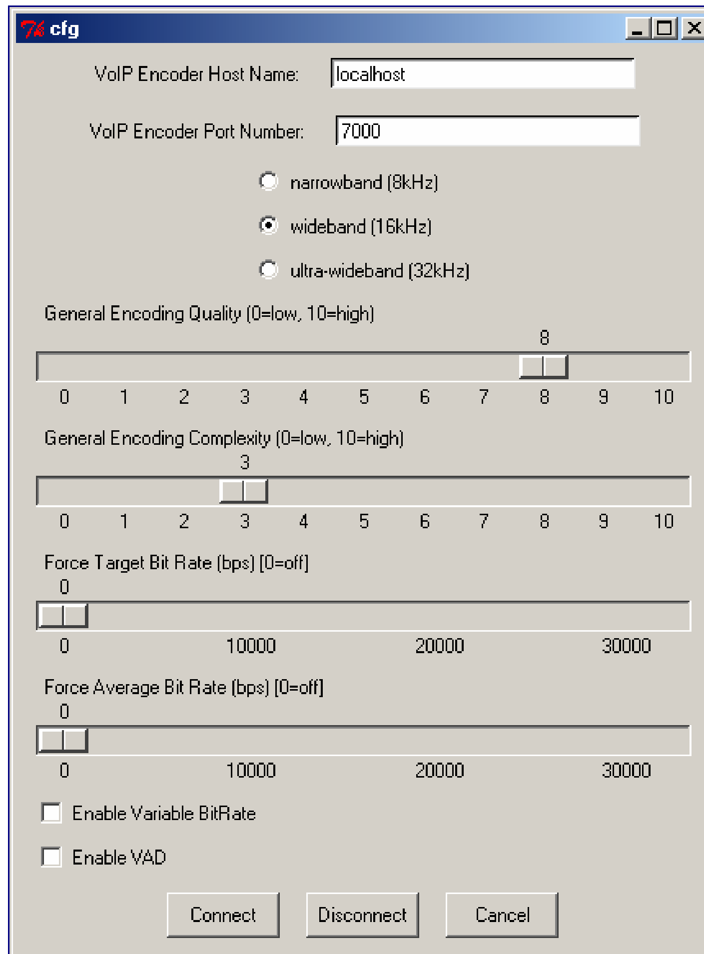


Figure 9: Configuration of Speex Compression Engine

19 Application Programming Interface (API)

19.1 Creating a new application

An example application called "`sonic_batch`" is included in the SONIC download (`sonic/2.0-beta5/src/app`). It provides an example of a Makefile to compile the application along with example code that illustrates using the low-level API calls. A server-mode implementation based on TCP/IP socket connections is provided in the program "`sonic_server`" located within the same directory. These two programs are provided as reference starting points for building new applications using the SONIC recognition API. The application programming interface (API) provides basic high-level functions that allow non-experts to build simple voice-activated applications. In order to build your own custom application your Makefile should include headers the following directories,

```
sonic/2.0-beta5/src/decoder
sonic/2.0-beta5/src/lattice
sonic/2.0-beta5/src/lexicon
sonic/2.0-beta5/src/misc
```

```
sonic/2.0-beta5/src/voip
```

While linking your application, you will need to include the following library directory,

```
sonic/2.0-beta5/lib/$(ARCH)-$(OS)
```

where ARCH and OS refer to the architecture of the machine on which the compile is executed. Linux systems typically have (i686-Linux) as the ARCH-OS value. The following libraries must be linked into your application,

```
-lsonic –lexicon –llattice –lfea -llangmod -lmisc_speech –lvoip -lm
```

Internally, your application should include the following header file at a minimum,

```
#include <sonic_api.h>
```

A simple batch-mode application and Makefile is provided to help illustrate the concepts,

```
sonic/2.0-beta5/src/app/sonic_batch.c
sonic/2.0-beta5/src/app/Makefile
```

A similar example program for socket-based communication in server mode is given in the following example program,

```
sonic/2.0-beta5/src/app/sonic_server.c
```

In the next section, we summarize the basic API level interface to SONIC. If you have been provided source-code level access to SONIC, the implementation details can be found in the files,

```
sonic/2.0-beta5/src/decoder/sonic_api.c
sonic/2.0-beta5/src/decoder/sonic_api.h
```

19.2 The SONIC Speech Recognition API

Function	Description
Basic Commonly Used Continuous Speech Recognition API Commands	
decoder_config_t * decoder_init(char *config_file)	Initializes the recognizer using the configuration options listed in config_file. Returns a pointer to a decoder configuration structure.
void decoder_free();	Frees up memory from the decoder config structure.
void decoder_begin_utt();	Tells the recognizer to get prepared to a new incoming audio segment for recognition. Responsible for resetting structures for each decoded utterance.
int decoder_rawdata(short * audio, int num_samples)	Tells the recognizer to update it's internal hypothesis with samples of audio from the input speech signal. If auto_end_point is set to 1, then this function also returns the state of the voice activity detector. State values are 0 for silence detected, 1 for speech-begin detected and 2 for speech-end detected.
void decoder_features(float **fea, int num_frames)	Same as rawdata function, but allows the recognizer to update it's internal hypothesis with frames of features rather than samples of audio

void decoder_get_partial_hypothesis(decoder_config_t * config, char *hypothesis_string);	Puts the current 1-best hypothesis from the recognizer into a printable string. Can be called during live-mode recognition to see the current best path from the recognizer.
word_hyp_t *decoder_lattice_bestpath();	Called right after all samples have been sent to the recognizer. Returns the best-path sequence of words from the first-pass lattice.
void decoder_free_word_hypothesis(word_hyp_t *);	Frees up memory allocated by the returned word-hypothesis structure malloc'd by the recognizer.
void decoder_end_utt();	Cleans up memory, resets recognizer for next begin_utt signal.
void decoder_abort_utt();	Aborts the recognition and cleans up memory.
Lattice API commands	
void decoder_lattice_dump(char *filename);	Dumps the word lattice to a file
void decoder_lattice_read();	Reads a lattice from disk into memory
word_hyp_t *decoder_A_star(char *filename);	Used to generate an N-best list which will be written to the filename provided. Function returns the first-best hypothesis.
word_hyp_t *decoder_lattice_rescore();	Rescores the word lattice and returns a sequence of recognized words in word_hyp_t structure. Depending on configuration options, performs: word-graph search, concept-lm rescoring, word-posterior rescoring.
void decoder_model_marked_lattice_dump(char *filename);	Dumps model-marked word lattice to a file. Model Marked word lattices contain HMM state timing information (can be used for speaker adaptation).
Grammar Recognition API Commands	
void decoder_load_network();	This function loads a grammar network which is specified in the config->network_name structure. If a grammar network is already loaded, this function will first free up memory from the current network prior to loading the new grammar.
Dynamic Vocabulary API Commands	
int *decoder_get_active_vocabulary(decoder_config_t *config, int *N);	Returns an integer array containing 1's and 0's. The values indicate whether or not the nth word appeared in the word-lattice after the first-pass search. The length of the returned vector should always be equal to the number of unigrams in the provided language model. The length of the array is set in the value N.
void decoder_set_active_vocabulary(decoder_config_t *config, int *vocab, int retain_top_N_words)	Given the short-list of active words from the previous API call, this function can be used to set the active vocabulary to include only words seen in the word lattice. In addition, the user can specify that the top N most frequent words in the vocabulary be retained in the search.
void decoder_set_lm(decoder_config_t *config, char *lm_name);	Given a set of language models specified in –lmctlfn option and loaded at runtime, this API calls allows the recognizer to dynamically swap language models on a per-utterance basis.
Online MLLR Speaker Adaptation API Commands	

void decoder_online_adapt(word_hyp_t *word_hypothesis, decoder_config_t *config);	After a word-hypothesis structure is returned from the recognizer, calling this command will cause SONIC to determine the state-level alignment of the utterance and update parameters of a single-regression class MLLR transform. The recognizer configuration settings should have –online_adapt set to 1 to enable this service. Online speaker adaptation is a new feature of SONIC 2.0-beta5. When used in tandem, SONIC will update an internal MLLR transforms for the current speaker and dynamically adapt HMM states as they are needed during decoding.
void decoder_online_adapt_reset(decoder_config_t *config);	This API command allows SONIC to reset the current single regression class MLLR transform. It is usually called between speakers and sessions.
Voice Activity Detection API Commands	
void decoder_calibrate_vad(short *rawdata, int numsamples);	This API function will adapt the silence model of the voice activity detector with the numsamples of audio in the array rawdata. The silence model is adapted using MAP reestimation of the model mean vector components. This function can be called for live-microphone based recognition systems or to adjust the detection of silence vs. speech in a changing noisy environment.
void decoder_set_push_to_talk(int pressed, decoder_config_t *config);	When –push_to_talk is set to 1 in the recognizer configuration file, this API command is used to tell the recognizer when the user is pushing down on the push-to-talk button. Push to talk internally is initialized to 0. When the user presses the talk key, this function should be called with “pressed” set to 1. When the user releases the talk key, this function should be called again with a value of 0 for pressed. If voice activity detection is turned on, this API command will override the VAD to speech force detection.
Error Message Handling API Commands	
void decoder_print_errors(decoder_config_t *config)	Prints any initialization errors to stderr
void decoder_print_warnings(decoder_config_t *config)	Prints any warning messages to stderr
void decoder_reset_errors(decoder_config_t *config)	Sets the internal config->error variable to 0 and rests the string buffer containing the error messages (config->error_messages)
void decoder_reset_warnings(decoder_config_t *config)	Sets the internal config->warning variable to 0 and resets the string buffer containing the warning messages (config->warning_messages)

Table 14: The Programming Interface to SONIC

19.3 Example Use of the SONIC API

All applications should first begin by calling the initialization function for the recognizer (by passing the filename of the configuration file as input),

```
word_hyp_t *word_hypothesis;
decoder_config_t *config;
config = decoder_init(decoder_config_file);
```

The speech recognizer can be run in either batch-mode (all of the signal is known at run-time) or in live-mode (the signal is streamed into the recognizer). In general batch-mode recognition is always more accurate than live-mode since the entire cepstral mean of the file can be calculated and subtracted during feature extraction.

Batch Mode Recognition

A typical loop in the recognizer run in batch-mode consists of a begin_utt, rawdata, hypothesis determination, end_utt sequence. For example,

```
decoder_begin_utt();
decoder_rawdata(rawdata, num_samples);
word_hypothesis = decoder_lattice_bestpath();
/* print out hypothesis */
decoder_free_word_hypothesis(word_hypothesis);
decoder_end_utt();
```

Here, the `begin_utt()` signal prepares the recognizer to receive the incoming data. The `decoder_rawdata` function passes audio samples (short * data) to the recognizer. This function can be called multiple times with chunks of audio for pipe-lined applications. Finally, the word-lattice is searched and a `word_hypothesis` data structure is returned. Finally, the `end_utt()` call frees up memory and resets the recognizer to receive another utterance.

The recognized words can be printed from the `word_hyp_t` data structure. For example,

```
cur = word_hypothesis;
for(cur = word_hypothesis;cur;cur = cur->next){
   printf("%s ", cur->word);
}
```

The `word_hyp_t` data structure contains much more information than just the words themselves. It can be used to print out their begin/end sample locations, acoustic and language model scores. In the future this data structure will contain word-level confidence information. Assuming the "-confidence 1" switch was set during decoding, we can print the begin sample location, ending sample location, word, and confidence using the following example code,

```
for(cur = word_hypothesis;cur;cur = cur->next){
   printf("%d %d %s %f", cur->begin_sample, cur->end_sample, cur->word,
            cur->confidence);
}
```

More information about the word_hyp_t data structure fields can be found in,

```
sonic/2.0-beta5/src/decoder/sonic_api.h
```

Live-Mode Recognition

A typical loop in the recognizer run in Live-mode consists of a `begin_utt`, sequential streams of rawdata, hypothesis determination, `end_utt` sequence. For example,

```
char hyp[10000];

decoder_begin_utt();

while( [there is samples to send] ) {
    decoder_rawdata(chunk_of_rawdata, small_chunk_of_samples);
    decoder_get_partial_hypothesis(config, hyp);                optional
    printf("the current hypothesis is [%s]\n", hyp);             optional
}
word_hypothesis = decoder_lattice_bestpath();
/* print final hypothesis */
decoder_free_word_hypothesis(word_hypothesis);
decoder_end_utt();
```

For live mode recognition, you should set the internal configuration variable (-live_mode to 1). It is recommended that data be sent in 250 msec chunks. During live-mode, each block of audio sequentially updates the recognition lattice. The final recognition hypothesis is obtained by calling the lattice_bestpath API call.

Live-Mode Recognition with Automatic Voice Activity Detection

For real applications with live microphone input, it is necessary to perform speech detection. A typical loop in the recognizer run in Live-mode with integrated voice activity detection is shown below. It is assumed that (`-live_mode 1`) and (`-auto_end_point 1`) are part of your configuration file.

```
char hyp[10000];
int   vad_state;
word_hyp_t *word_hypothesis;
decoder_config_t *config;

// initialize the recognizer to the current vocabulary

config = decoder_init(decoder_config_file);
vad_state = -1;

while(1) {

   if (vad_state == -1)
      decoder_begin_utt();

   [ get audio from microphone here and place into array ]

   vad_state = decoder_rawdata(chunk_of_rawdata, small_chunk_of_samples);

   switch(vad_state){
      case 0: printf("silence\n");
      case 1: printf("begin of speech detected\n");
      case 2: printf("end of speech detected\n");
```

```
            word_hypothesis = decoder_lattice_bestpath();
            print_hypothesis(word_hypothesis);                 // not provided
            decoder_free_word_hypothesis(word_hypothesis);     // clean up memory
            decoder_end_utt();                                 // reset recognizer
            decoder_begin_utt();                               // start new utt.
            break;
   }
}
```

For live mode recognition, you should set the internal configuration variable (-live_mode to 1). It is recommended that data be sent in 250 msec chunks. During live-mode, the recognition lattice is sequentially updated by each block of audio. The final recognition hypothesis is obtained by calling the lattice_bestpath API call.

19.4 Error Message Handling

Users will inevitably make errors when setting up a recognizer. Missing files, incorrect paths, are just some of the typical mistakes. SONIC provides an error trapping mechanism to provide information to the developer when certain error conditions exist. Below is an example code snippet of trapping error events at recognizer startup.

```
word_hyp_t *word_hypothesis;
decoder_config_t *config;
config = decoder_init(decoder_config_file);
if (config->error){
   printf("error occurred at startup: error message is [%s]\n",
          config->error_message);
   // Or you can use this call,
   decoder_print_errors(config);
}
if (config->warning){
   printf("warning occurred at startup: warning message is [%s]\n",
          config->warning_message);
   // Or you can use this call,
   decoder_print_warnings(config);
}
decoder_reset_errors(config);
decoder_reset_warnings(config);
```

Error messages are used to generally reflect conditions in which the recognizer cannot initialize itself (for example, missing files, incorrect file paths, etc.). Warning messages are not catastrophic events (for example, missing word pronunciations in the lexicon only causes the recognizer not to be able to recognize that particular word in the vocabulary).

References

[1] S. B. Davis and P. Mermelstein, "Comparison of the parametric representations for monosyllabic word recognition in continuously spoken sentences", IEEE Transactions on Acoustics, Speech, and Signal Processing, Vol. ASSP-28, 1980, pp. 357-366.

[2] S. S. Stevens and J. Volkmann, "The relation of pitch to frequency: A revised scale", American Journal of Psychology", Vol. 53, 1940, pp. 329-353.

[3] G. McLaughlin, *Mixture Models*, New York: Marcel Dekker, 1988.

[4] D. Reynolds, R. Rose, "Robust Text-Independent Speaker Identification Using Gaussian Mixture Speaker Models," *IEEE Transactions on Speech and Audio Processing,* Vol. 3, No. 1, pp. 72-83, Jan.1995.

[5] A. Dempster, N. Laird, and D. Rubin, "Maximum Likelihood from incomplete data via the EM algorithm," *J. Royal Stat. Soc.,* Vol. 39, pp. 1-38., 1977.

[6] L. Baum *et al.*, "A maximization technique occurring in the statistical analysis of probabilistic functions of Markov chains," *Ann. Math. Stat.,* Vol. 41, pp. 164-171, 1970.

[7] J. Foote, G. Jones, K. Jones, S. Young, "Talker- Independent Keyword Spotting for Information Retrieval," Proc. Eurospeech, 1995, Vol 3, pp. 2145-2149.

[8] R. Rose, D. Paul, "A Hidden Markov Model Based Keyword Recognition System," Proc. IEEE ICASSP-90, Vol. 1, pp. 129-132.

[9] L. F. Uebel, P.C.Woodland, "An investigation into Vocal Tract Length Normalization", Proceedings of Eurospeech-99, Budapest, Hungary, 1999.

[10] L. Welling, S. Kanthak, H. Ney, "Improved Methods for Vocal Tract Length Normalization", Proceedings of ICASSP-99, Phoenix Arizona, 1999.

[11] D. Pye, P. C. Woodland, "Experiments in Speaker Normalization and Adaptation for Large Vocabulary Speech Recognition," Proceedings of ICASSP-97, Munich Germany, 1997.

[12] C.J. Legetter, P.C. Woodland, "Maximum likelihood linear regression for speaker adaptation of continuous density hidden Markov models," Computer Speech & Language, Vol. 9, pp. 171-185, 1995.

[13] Kadri Hacioglu & Wayne Ward, "A Concept Graph based Confidence Measure", Proc. IEEE International Conference on Acoustic, Speech, and Signal Processing (ICASSP), Orlando Florida, May 2002.

[14] S.J. Young, N.H. Russell, J.H.S. Thornton, "Token Passing: A Simple Conceptual Model for Connected Speech Recognition Systems," Cambridge University Engineering Department Technical Report CUED/F-INFENG/TR.38, July 1989.

[15] O. Salor, B.L. Pellom, T. Ciloglu, K. Hacioglu, M. Demirekler, "On Developing New Text and Audio Corpora and Speech Recognition Tools for the Turkish Language", International Conference

on Spoken Language Processing (ICSLP), vol. 1, pp. 349-352, Denver, Colorado USA, September 2002.

[16] O. Siohan, C. Chesta, and C.-H. Lee, "Joint Maximum a Posteriori Adaptation of Transformation and HMM Parameters," IEEE Trans. on Speech & Audio Proc. Vol. 9, No. 4, pp. 417-428, 2001.

[17] O. Siohan, T. Myrvoll, and C.-H. Lee, " Structural Maximum a Posteriori Linear Regression for Fast HMM Adaptation", Computer, Speech and Language, 16, pp. 5-24, January 2002.

[18] Umit H. Yapanel, John H.L.Hansen, "A new perspective on Feature Extraction for Robust In-vehicle Speech Recognition" Proceedings of Eurospeech'03, Geneva, Sept. 2003.

[19] A. Black, K. Lenzo, V. Pagel, "Issues in Building General Letter to Sound Rules" 3rd ESCA Workshop on Speech Synthesis, pp. 77-80, Jenolan Caves, Australia, 1998.

[20] V. Pagel, K. Lenzo, and A. Black, "Letter to sound rules for accented lexicon compression ICSLP98, vol 5 pp 2015-2020, Syndey, Australia, 1998.

[21] M.N. Murthi and B.D. Rao. "MVDR Based All-Pole Models for Spectral Coding of Speech" *Proceedings of 1999 International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, Phoenix, 1999.

[22] M. J. F. Gales, "Maximum Likelihood Linear Transformations for HMM-Based Speech Recognition," Technical Report CUED/F-INFENG/TR 291, Cambridge University, May 1997.