Lecture 3: Acoustic Features

Lecturer: Mark Hasegawa-Johnson (jhasegaw@uiuc.edu)
TA: Sarah Borys (sborys@uiuc.edu)
Web Page: http://www.ifp.uiuc.edu/speech/courses/minicourse/

June 27, 2005

1 Where to get Software

Which software should you use for your speech recognizer development? It depends on the amount of customization you want to do. This section will talk about four types of code: commercial off-the-shelf, academic off-the-shelf, command line toolkit, and object libraries. Most of the lectures in this course will focus on the "command-line toolkit" category, and most Illinois speech recognition software to date has been designed around these toolkits. That may soon change.

If you spend time modifying somebody else's source code to achieve your own research goals, you want to know that you will be able to redistribute your work. This section will describe the licensing conditions of each package. A package is considered "Open Source" if the license permits the user to modify the source code, and to redistribute the result, either for free or in the form of a commercial software system. Open Source software includes Sphinx, ISIP, MIT FST, LAPACK, and STL. Some authors distribute source, but do not grant you the right to modify and redistribute: these include QuickNet, HTK, LibSVM, and SVM-light.

Commercial Off-the-Shelf: There are a number of commercial off-the-shelf systems: some are specialized for dictation, some for call-center applications, some for information retrieval. I have only personally tested one commercial dictation system. After adapting the system to individual voices, we achieved 95% word recognition accuracy on really unusual texts. The system adapts to your voice, and to your language usage; it is possible to teach it special vocabulary items. Most other customization is impossible: even scripting (to do batch-mode recognition of multiple files) is only possible using the "professional edition."

Academic Off-the-Shelf: These systems are released with trained recognizer parameters, so that you can use them out of the box to do speech recognition, if your application uses exactly the same speaking style and recording hardware as those used to train the recognizer. All of these also include utilities for re-training the recognizer using new speech data. Two of these are open source, and Berkeley's is semi-open. Any open source system may be used as a "template" if you want to modify the system, or create your own.

1. Sphinx (http://cmusphinx.sourceforge.net/). License: Open Source, similar to Apache. Language: Sphinx 3.5 is in C, Sphinx 4 is in Java. Distributed recognition models: trained using DARPA Hub4, i.e., Broadcast News.

2. ISIP (http://www.isip.msstate.edu). License: Open Source, all uses explicitly permitted. Language: C. Distributed recognition models: Resource Management (DARPA small-vocabulary dialog system).

3. SPRACH/QuickNet (http://www.icsi.berkeley.edu/~dpwe/projects/sprach/sprachcore.html). License: Not quite open source — redistribution and non-commercial use allowed, commercial use and redistribution of modified code not allowed. Code: C. Distributed recognition models: demo is available for the Berkeley Restaurant Project (BeRP) dialogue system.

4. GMTK (http://ssli.ee.washington.edu/~bilmes/gmtk/). License: release 1.3.12 is binaries only; release 2 will be open source. Distributed models: Aurora digit recognition in noise.

5. SONIC (http://cslr.colorado.edu/beginweb/speech_recognition/sonic.html). License: binaries/libraries are downloadable; no source except for the client/server example code. Tutorials are available on-line

for TI-Digits and the Resource Management corpus, both of which are available at UIUC on the IFP network.

Command Line Toolkits: Most of the lectures in this course will focus on these software packages.

1. HTK (http://htk.eng.cam.ac.uk/) is designed to facilitate customized training of your own speech recognition models. Use: (1) compile the toolkit, (2) create configuration files specifying your desired architecture, (3) create bash or perl scripts to run training programs. Advantages: HTK is perhaps the best-documented of all academic recognizers (http://htk.eng.cam.ac.uk/docs/docs.shtml, [5]); unlike other recognizers, many modifications are possible without changing any source code. Disadvantages: if you need to modify the source code, you do not want to use HTK, because (1) the source code is quirky, and violates POSIX style standards, (2) HTK is not open source; the license allows modification but not redistribution.

2. GMTK is designed to be used in the same way as HTK, i.e., system architecture is specified using extensive configuration files. Customization in GMTK is much, much more flexible than HTK.

3. AT&T FSM Library (http://www.research.att.com/sw/tools/fsm/) and decoder Distribution: binaries only, controlled by configuration files and scripting. More on this later.

4. Support vector machines are usually trained using similar configuration + scripting methods. LibSVM (http://www.csie.ntu.edu.tw/ cjlin/libsvm/) and SVM-light (http://svmlight.joachims.org/) have complementary functionality, described in the next lecture. NEITHER are open source!! Source code is available, but licenses for both toolkits prohibit redistribution prohibits redistributionor commercial use.

5. PVTK (Periodic Vector Toolkit: http://www.ifp.uiuc.edu/speech/software). License: Open Source, Apache license — but HTK must be installed in order for PVTK to compile. This toolkit contains three command line programs useful for concatenating and transforming spectral vectors, extracting spectral vectors to train a neural net or SVM, and applying a bank of SVMs to a huge speech database in batch mode. Code is messy and nonstandard, and will change soon (if you'd like to volunteer to help, please do!), but we will try to maintain the same command-line interface.

Object Libraries: These are libraries of functions designed to be used inside of your own program. The two most important object libraries are not speech libraries at all, but general purpose time-savers: LAPACK and STL. In the speech domain, any open-source recognizer is a potential "object library," but (to my knowledge) only three have been designed that way on purpose.

1. LAPACK (http://www.netlib.org/lapack95, http://www.netlib.org/clapack, http://www.netlib.org/lapack++, http://www.cs.utk.edu/java/f2j/). License: open source, similar to LaTeX, meaning that if you modify and redistribute any function, you should change the filename. Use: LAPACK is a standard for efficient linear algebra routines (matrix inverse, minimum-norm pseudo inverse, eigenvalues, SVD, et cetera). Parallel implementations exist.

2. Standard Template Library (STL: http://www.sgi.com/tech/stl/). Provides templates (header files) that enrich the C++ type system with hash tables, iterators, vectors, strings, et cetera.

3. MIT Finite State Toolkit (alpha release available from Lee Hetherington). License: Open Source. Advantage: an STL-compatible finite state transducer template. Disadvantage: no documentation yet.

4. SpeechLib (http://www.ifp.uiuc.edu/speech/software). License: Open Source, Apache. Use: header files and functions implement Graphs (graphs in general, and Annotation Graph transcription type in particular), PDFs, neural networks, and some linear algebra. Code is clean but nonstandard, and will change soon to better interface with LAPACK and STL (if you'd like to volunteer to help, please do!).

2 Acoustic Features

All acoustic features commonly used in automatic speech recognition are based on the spectrogram. A "spectrogram" is a time-frequency representation of the signal, $X(t, f)$, where t may be in seconds and f may be in Hertz. Specifically, $X(t_0, f)$ is the log magnitude of the Fourier transform, at frequency f, of the signal multiplied by a short window $w(t)$ ($w(t)$ is usually 20-30ms in length):

$$X(t_0, f) = |\mathcal{F}\{x(t + t_0)w(t)\}| \quad (1)$$

$$\text{SPEC}(t_0, f) = \log X(t_0, f) \quad (2)$$

Phonetically readable spectrogram displays compute $X(t_0, f)$ once per 2ms, and humans can hear acoustic events with a temporal precision of about 2ms, but in order to save memory and computational complexity, speech recognition algorithms usually only compute $X(t_0, f)$ once per 10ms. The only speech events not well represented by a 10ms skip are stop bursts. Perhaps more important, a 10ms skip-time makes it difficult to deal with impulsive background noise (clicks).

2.1 Mel Spectra

Human hearing is characterized by two different semi-logarithmic frequency scales: the Bark scale, and the mel scale. First, the inner ear integrates spectral energy in bands of width equal to one Bark; for this reason, if the total noise power within 1/6 octave on either side of a tone is more than about 4dB above the power of the tone, the tone will be masked. One Bark is equal to about 1/6 octave (two semitones, or 12%), with a minimum value of about 312Hz. Second, the just-noticeable-difference between the frequencies of two pure tones is about 3 mels. One mel is equal to about 0.1% (about 1/700 octave), with a minimum value of about 0.6Hz.

Both of these scales suggest that the ability of humans to discriminate two sounds is determined by the spectral energy as a function of semilog-frequency (Bark frequency or mel frequency), not linear frequency (Hertz). Most speech recognition features average the spectral energy within bandwidths of 1 Bark, or about 90-120 mels, in order to eliminate distinctions that depend on the difference between two frequencies that are within 1 Bark of each other. This frequency-dependent smearing or smoothing process substantially increases the accuracy of a speech recognizer.

Frequency-dependent smearing can be done using sub-band filters or wavelets [6], but is most often done by, literally, adding up the Fourier transform coefficients in bands of width equal to about 90 mels (resulting in roughly 32 bands between 0Hz and 8000Hz). The resulting coefficients are called mel-frequency spectral coefficients, or MFSC:

$$\tilde{X}(t_0, k) = \sum_{f=0}^{4000} X(t_0, f) H_k(f) \quad (3)$$

$$\text{MFSC}(t_0, k) = \log \tilde{X}(t_0, k) \quad (4)$$

The weighting functions, $H_k(f)$, are usually triangular, centered at frequencies f_k that are about 90 mels apart [2].

2.2 Mel Cepstra

Neighboring spectral coefficients are highly correlated: for example, if there is a spectral peak near center frequency f_k, then $\text{MFSC}(t, k)$ is often large, but usually, so is $\text{MFSC}(t, k+1)$. Probabilistic models such as diagonal-covariance Gaussians work best if the features are uncorrelated. It has been shown that the MFSCs can be pretty well decorrelated by transforming them using a discrete cosine transform or DCT:

$$\text{MFCC}(t_0, m) = \mathcal{C}\{\text{MFSC}(t_0, k)\} \quad (5)$$

The DCT, $\mathcal{C}$, is just the real part of a Fourier transform. Thus, in effect, the MFSCs are the inverse Fourier transform of the log of the frequency-warped spectrum. $\text{MFCC}(t, 0)$ is similar to the average spectral energy; $\text{MFCC}(t, 1)$ is similar to the average spectral tilt. The peak corresponding to F1 occurs at $\text{MFCC}(t, F_s/F_1)$.

Eq. 5 is useful for decorrelating the MFSCs, but it actually doesn't change the perceptual distance between two sounds. Because the DCT is a unitary transform, the distance between two MFSC vectors is exactly the same as the distance between two MFCC vectors:

$$\sum_k \left(\mathrm{MFSC}_1(k) - \mathrm{MFSC}_2(k)\right)^2 = \sum_m \left(\mathrm{MFCC}_1(m) - \mathrm{MFCC}_2(m)\right)^2 \tag{6}$$

Thus it's possible to think of Eq. 5 as a meaningless mathematical convenience — it simplifies the probabilistic model, but doesn't change the representation of human perception.

2.3 LPCC

Humans are most sensitive to the frequencies and amplitudes of spectral peaks. There is some evidence that humans completely ignore zeros in the spectrum, except to the extent that the zeros change the amplitudes of their surrounding peaks. Thus it may be reasonable to smooth the spectrum in some way that carefully represents the frequencies and amplitudes of peaks, while ignoring inter-phoneme differences in the amplitudes of spectral valleys. It has been shown [4] that focusing on the peak can be accomplished by using a parameterized spectrum $A(t_0, f)$, and by choosing its parameters according to the following constrained minimization:

$$A(t_0, f) = \arg\min \sum_f \left(\frac{X^2(t_0, f)}{A^2(t_0, f)}\right) \quad \text{subject to } \sum_f X^2(t_0, f) = \sum_f A^2(t_0, f) \tag{7}$$

When the vocal folds clap together, the resulting impulse-like sound excites the resonance of the vocal tract (formants); the sound pressure in the vocal tract continues to ring like a bell for a few milliseconds afterward. During this time, the waveform looks like a sum of exponentially decaying sine waves. Once you have figured out the frequencies and decay rates of these sine waves (formants), you can predict each sample of the speech signal from its $2N$ previous samples (N is the number of formants; $2N$ because you need to know both the frequency and decay rate). The prediction coefficients are called linear prediction coefficients, or LPC [1]. The LPC coefficients a_i specify a model of the spectrum:

$$A(t_0, f) = \left| \frac{1}{1 - \sum_{i=1}^{2N} a_i e^{j2\pi i f F_s}} \right| \tag{8}$$

where F_s is the sampling frequency. If a_i are chosen according to Eq. 7 (as they usually are), then $A(t, f)$ is a smoothed spectrum that accurately represents the amplitudes and frequencies of spectral peaks, possibly at the cost of more

The linear predictive cepstral coefficients (LPCC) are computed by taking the DCT of $\log A(t_0, f)$, in order to decorrelate it:

$$\mathrm{LPCC}(t_0, m) = \mathcal{C}\left\{\log A(t_0, f)\right\} \tag{9}$$

The LPC smoothing process (Eq. 8) changes the implied perceptual distance between two sounds: sounds with different spectral valleys become more similar, while sounds with different spectral peaks become less similar. The LPC→LPCC transformation (Eq. 9) doesn't change the distance between two sounds; it just decorrelates the features.

2.4 Perceptual LPC (PLP)

Perceptual LPC combines together the frequency-dependent smoothing of MFSC with the peak-focused smoothing of LPC [3]. The spectral model $\tilde{A}(t_0, k)$ is chosen according to

$$\tilde{A}(t_0, k) = \arg\min \sum_k \left(\frac{\tilde{X}^2(t_0, f)}{\tilde{A}^2(t_0, f)}\right) \quad \text{subject to } \sum_k \tilde{X}^2(t_0, k) = \sum_k \tilde{A}^2(t_0, k) \tag{10}$$

PLP coefficients are never used directly, except for the purpose of displaying a pretty smoothed spectrum. In speech recognition, they are always DCT'd:

$$\mathrm{PLP}(t_0, m) = \mathcal{C}\left\{\log \tilde{A}(t_0, k)\right\} \tag{11}$$

3 Computing Acoustic Features with HTK

The HTK program `HCopy` can convert a waveform into any of the acoustic feature types described in Sec. 2.

If you are on a machine that doesn't have HTK already installed, download the source code and/or binaries from http://htk.eng.cam.ac.uk. You will have to register (free). While you're there, download the HTK book. If you plan to do anything serious, you will need chapters 4-17.

3.1 Using HCopy

The `HCopy` program copies a speech file. In the process of copying, it can (1) convert from one feature type to another, (2) convert from other file formats to HTK file format, (3) extract subsegments corresponding to labeled phonemes and/or specified start and end times, or (4) concatenate multiple files. `HCopy` is usually called (from the unix or DOS command line) as

```
HCopy  -S scriptfile.scp -T 1 -C config.txt
```

The `-T 1` option tells any HTK program to print trace (debugging) information to standard error. Higher trace settings print more information; `-T 1` will just print the name of each speech file as it is converted.

An HTK "script file" is a list of the files to be processed. All HTK tools can take a script file using the `-S` option. The content of a script file is literally appended to the command line; thus, for example, the command shown above could also be implemented by typing

```
HCopy  -S scriptfile.scp
```

and by including, as the first line in the script file, the characters `-T 1 -C config.txt`. In general, it is considered good form to specify options like `-T` and `-C` on the command line, and to use the script file only to specify the filename arguments.

The filename arguments of `HCopy` come in inputfile-outputfile pairs, e.g., the file might include

```
timit/TRAIN/DR1/FCJF0/SA1.WAV data/fcfj0sa1.plp
timit/TRAIN/DR1/FCJF0/SA2.WAV data/fcfj0sa2.plp
...
```

The line breaks are optional; you can specify all of `HCopy`'s filename arguments on the same line, but your script file will be less readable that way. A script file like the one above could be generated using the following bash script:

```
#!/bin/bash
foreach d ‘ls timit/TRAIN‘; do
  foreach s ‘ls timit/TRAIN/$d‘; do
    foreach f ‘ls timit/TRAIN/$d/$s/*.WAV‘; do
      output=‘echo $f | sed ’/WAV/s/.*\//data\//;/WAV/s/WAV/plp/;’‘;
      echo $f $output;
    done
  done
done
```

All of the processing to be performed is specified in the configuration file, which could be called `config.txt`. Here is an example:

```
SOURCEFORMAT = NIST
SOURCEKIND = WAVEFORM
TARGETKIND = PLP_E_D_A_Z_C
TARGETRATE = 100000.0
WINDOWSIZE = 250000.0
NUMCHANS = 32
CEPLIFTER = 27
NUMCEPS = 12
```

Here are what the lines of the configuration file mean:

- `SOURCEFORMAT` specifies the file format of the input. This can be `NIST` (for NIST or LDC distributed SPHERE files), or `WAV` (for Microsoft WAV files), or `HTK` (for HTK-format files).
- `TARGETFORMAT`, if specified, would declare the desired format of the output. Only waveform data can be output in any form other than HTK. Since HTK is the default, TARGETFORMAT doesn't need to be specified.
- `SOURCEKIND` specifies that the input is waveform data. This line is optional, since WAVEFORM is the default input data kind.
- `TARGETKIND` specifies that the outputs are PLP cepstral coefficients (Sec. 2.4). Other useful options include `MELSPEC` ($\tilde{X}(t_0, k)$, Sec. 2.1), `FBANK` (MFSCs, Sec. 2.1), `MFCC` (Sec. 2.2), and `LPCEPSTRA` (Sec. 2.3). The modifiers have the following meanings:
  - `_E`: Append a normalized energy to each PLP vector. The "normalized energy" is the log Energy, normalized so that the highest value in each utterance file is exactly 1.0.
  - `_D`: Append the delta-PLPs and delta-energy to each PLP vector.
  - `_A`: Append delta-delta-PLPs and delta-delta-energy to each PLP vector.
  - `_Z`: Perform cepstral mean subtraction: from each cepstral vector, subtract the average cepstral vector, where the average is computed over the entire file. This option is most useful when the recognizer will be tested with a different microphone, or in a different kind of room, than was used to train the recognizer. Under other conditions, it may hurt recognition performance.
  - `_C`: Compress the coefficients. Compressed HTK files are half as large as uncompressed files, and compressed files may actually have *better* precision than uncompressed files.
- `WINDOWSIZE` specifies the length of the window $w(t)$, in 100ns units. The example specifies a 25ms window ($25ms = 25000\mu s = 250000 \times 100ns$).
- `TARGETRATE` specifies the frame skip parameter, in 100ns units. The example specifies one frame per 10ms ($10ms = 10000\mu s = 100000 \times 100ns$).
- `NUMCHANS` specifies the number of mel-frequency filterbanks to use. Thus, in Eq. 10, k goes from 1 to 32.
- `NUMCEPS` specifies the number of cepstra to keep. Thus the total acoustic feature vector will have dimension 39: 12 cepstra, then the energy, then the deltas of all 13 coefficients, then their delta-deltas.
- `CEPLIFTER` specifies that the cepstrum should be "liftered," in order to slightly de-emphasize the lower-order cepstra. Liftering is useful because the low-order cepstra (small m) are usually much larger in amplitude than the high-order cepstra (large m). Without liftering, speech recognition distance measures would completely ignore the high-order cepstra. Liftering multiplies the cepstra by a lifter of length $L = 27$:

$$\mathrm{PLP}'(t, m) = \mathrm{PLP}(t, m) \times \left(1 + \frac{L}{2} \sin \frac{\pi m}{L}\right) \tag{12}$$

3.2 HTK File Format

The HTK file format is one of the best available formats for the compact but precise storage of periodic real-valued spectral or cepstral vectors (e.g., one vector every 10ms for 2 seconds, or one vector per day for 365 days, or any other such file). The main advantage of HTK format is the fast, fixed-length, high-accuracy data compression scheme. The main disadvantage is the unnecessarily restrictive type checking.

An HTK file is a 12-byte header, followed optionally by compression information, followed by sample vectors stored in either 4-byte floating point or 2-byte short integer format. All data are stored in IEEE standard floating point, with big-endian data order (high-order byte stored first), thus if you are writing C

code to read or write an HTK file on a little-endian machine (e.g., Intel or AMD), you will need to implement byte swapping. The header contains the following variables:

nSamples	Number of sample vectors in the file (4 byte integer)
sampPeriod	Sample period in 100ns units (4 byte integer)
sampSize	Number of bytes in each sample vector (2 byte integer)
parmKind	Two-byte code specifying feature kind

The `parmKind` code specifies the base kind (PLP, MFCC, etc), plus all modifiers (_E_D_A...) using a code specified on p. 66 of the HTK book. The `sampSize` specifies the number of *bytes* per vector, not the number of `dimensions:` thus if vectors are compressed, `sampSize` is twice the number of dimensions, otherwise four times.

If the file is uncompressed, the rest of the file is a series of parameter vectors, stored in 4-byte floating point format (IEEE standard big-endian).

If the file is compressed, the next $8N_d$ bytes contain $2N_d$ floating point numbers: $A[1] \ldots A[N_d]$, and $B[1] \ldots B[N_d]$, where N_d is the dimension of each sample vector. After the compression information, the sample vectors themselves are stored as 2-byte big-endian integers, $D[t, m]$. The value of the real features are computed as:

$$\mathrm{PLP}(t, m) = \frac{1}{A[m]} \left(D[t, m] + B[m] \right) \tag{13}$$

The values of $A(m)$ and $B(m)$ are chosen as specified on p.67 of the HTK book, in order to make sure that $\max_t D[t, m] = 1$ and $\min_t D[t, m] = -1$. Since the coefficients are scaled up to take full advantage of the 16-bit short integer, the signal to quantization noise ratio (SQNR) of features encoded this way is truly 2^16, or 96dB — better, in some cases, than the SQNR of a naive floating-point encoding.

4 Manipulating HTK Files with matlab, C, or PVTK

If you are doing research on the acoustic or phonetic features of a speech recognizer, you will want to read and write HTK-format files. Using the information in Sec. 3.2, together with pp. 66-67 of the HTK book, you can easily write your own code to read and write HTK files. There are a few oddities to remember: (1) the file should be compressed if and only if bit 11 of the `parmKind` is set (see p. 66 of HTK Book), (2) when writing files, clear bit 13 of the `parmKind`, otherwise HTK tools will expect a checksum at the end of the file, (3) when writing files that do not contain delta or delta-delta coefficients, be sure to clear bits 9 and 10 of the `parmKind`; otherwise HTK will insert zeros on the end of each feature vector in order to ensure that the feature vector dimension is compatible with the presence of delta and/or delta-delta coefficients (i.e., to make sure the feature dimension is a multiple of 3).

Of course you don't need to write your own. matlab code is available in the `speechfileformats.tgz` file available on the course web page. There are three different C implementations available. (1) `PVTK` reads an HTK file as a matrix. The code is in file `PVTKLib.c`, functions `ReadHParm` and `WriteHParm`; the matrix definition is specified in `HTKLib/HMem.h`. (2) `SpeechLib` includes an "observation" data type (file `Observation.h`), and functions that read and write observations from HTK files. SpeechLib is completely self contained, and does not depend on the HTK librares. (3) HTK's code for reading and writing files is contained in the file `HTKLib/HParm.c`. HTK's code is extremly complex, so that it can handle streaming audio input — if you don't need streaming audio, use either the `PVTK` or the `SpeechLib` versions.

4.1 Using VTransform to Transform HTK Vectors

In order to compile PVTK, you must first have HTK compiled. Once HTK is compiled, download the PVTK source. Edit the following lines in the PVTK Makefile, in order to specify the location of the HTK libary on your system:

```
# Where are the HTK libraries?
hlib = htk
HLIBS = $(hlib)/HTKLib.$(CPU).a
```

Then type `make`.

PVTK contains three programs: `VTransform`, `VExtract`, and `VApplySvms`. In order to see a complete manual page, type the name of the program without any arguments, and hit return.

`VTransform` can be used to concatenate multiple HTK vectors together into a single vector, or to implement arbitrary linear or nonlinear transformations of the vectors in an HTK file. `VTransform` could be used to apply a neural network or SVM to an HTK file — but not to train the neural network or SVM!!

For example, consider the following (rather arbitrary) transformation. Suppose you wish to accept pairs of input files, of type MFCC and PLP. You want to concatenate the feature vectors from each input file, and then concatenate together 21 successive frames of the result, creating a very large data matrix A_0. Finally, you want to apply the following transformation, in order to create data matrix A_5:

$$A_5 = \sin(A_0 B + b) - \tanh(A_0 C + c) \tag{14}$$

Applying this transform consists of the following steps. First, create the matrix files Bb.txt and Cc.txt, containing the matrices B and C and the row vectors b and c in the format given below. Matrices are loaded from text files (e.g. Bb.txt, Cc.txt) with the following format. The first line must contain the number of rows (including the offset vector) and the number of columns. The second line contains the offset vector. Remaining lines contain the rows of the transform matrix. Comments are not allowed. A non-square transform matrix will result in a change in the dimension of the feature vector. Here is a short example matrix file, for converting from a 5-dimensional input feature vector to a 4-dimensional output feature vector. The offset vector is here set to zero:

```
6 4
0 0 0 0
1 1 1 1
1 1 1 -1
1 1 -1 -1
1 -1 -1 -1
1 1 -0.5 0.5
```

Once you have created the files Bb.txt and Cc.txt, then call

```
VTransform -S TRAIN.scp -c 21 -m Bb.txt 0 -n sin 1
     -m Cc.txt 0 -n tanh 3 -n subtract 2 4
```

The `-S` option works just as in HTK tools. If there were only one inputfile per outputfile, then the file `TRAIN.scp` would contain inputfile-outputfile pairs, e.g.,

```
data/fcjf0sa1.plp transformed/fcjf0sa1.vtt
data/fcjf0sa2.plp transformed/fcjf0sa2.vtt
...
```

In this example, however, we want to concatenate feature vectors from two different input files: an MFCC file, and a PLP file. This is done by separating the two input files using the `-h` option (specifying "horizontal" concatenation of the two data matrices). The file `TRAIN.scp` therefore contains:

```
data/fcjf0sa1.mfc -h data/fcjf0sa1.plp transformed/fcjf0sa1.vtt
data/fcjf0sa2.mfc -h data/fcjf0sa2.plp transformed/fcjf0sa2.vtt
...
```

The `-c` option specifies that, at this point, `VTransform` should create a new data matrix that is 21 times as wide as the existing matrix. This is done by concatenating frames. If the MFCC and PLP input files each contained N_f frames and N_d dimensions, then the new data matrix A_0 will be of size $N_f \times 42N_d$:

$$A_0(t,:) = [\text{MFCC}(t-10,:), \text{PLP}(t-10,:), \cdots, \text{MFCC}(t+10,:), \text{PLP}(t+10,:)] \tag{15}$$

The output matrix has exactly as many rows as the input matrix; Eq. 15 assumes that frames $t \leq 0$ are identical to frame $t = 1$, and that frames $t > N_f$ are identical to frame $t = N_f$.

The remaining five command line options specify a series of five linear and nonlinear transformations of the data. Linear transformations are specified using `-m`; nonlinear options are specified using `-n`. Data is transformed in the following stages:

1. $A_1 = A_0 B + b$ — linear transformation.
2. $A_2 = sin(A_1)$ — element-wise nonlinear transformation.
3. $A_3 = A_0 C + c$ — linear transformation. Notice that the option specification (`-m Cc.txt 0`) specifies that the input to stage 3 should come from A_0.
4. $A_4 = \tanh(A_3)$
5. $A_5 = A_2 - A_4$.

If `VTransform` does not contain the nonlinear transform that you are looking for, it is easy to add it. Open the file `VTransform.c`, and find the function `FunctionToDVectors`. Here there is a long list of string comparisons followed by arithmetic operations:

```
...
    else if (!strcmp(func,"exp"))  z[i] = exp(x[i]);
    else if (!strcmp(func,"cos"))  z[i] = cos(x[i]);
    else if (!strcmp(func,"sin"))  z[i] = sin(x[i]);
...
```

add your favorite nonlinear function to this list, and give it a name. Now, at the command line, type `make`; then when the program has compiled, you can specify your new function with the `-n` option to `VTransform`.

References

[1] B. S. Atal and Suzanne L. Hanauer. Speech analysis and synthesis by linear prediction of the speech wave. *Journal of the Acoustical Society of America*, 50(2):637–655, 1971.

[2] Steven Davis and Paul Mermelstein. Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences. *Trans. ASSP*, ASSP-28(4):357–366, August 1980.

[3] Hynek Hermansky. Perceptual linear predictive (plp) analysis of speech. *Journal of the Acoustical Society of America*, 87(4):1738–1752, 1990.

[4] Lawrence R. Rabiner and Ronald W. Schafer. *Digital Processing of Speech Signals*. Prentice-Hall Inc., New Jersey, 1978.

[5] Steve Young, Gunnar Evermann, Thomas Hain, Dan Kershaw, Gareth Moore, Julian Odell, Dave Ollason, Dan Povey, Valtcho Valtchev, and Phil Woodland. *The HTK Book*. Cambridge University Engineering Department, Cambridge, UK, 2002.

[6] Tong Zhang, Mark Hasegawa-Johnson, and Stephen E. Levinson. Extraction of pragmatic and semantic salience from spoken language. In review.