**18-551 Digital Communications and Signal Processing**
**System Design**
**Spring 2002**
**Professor Casasent**

# Group #5: Language Identification

**Abel Villca Roque (avr)**
**Velik Bellemin (vbellemi)**
**William Y. Liu (wyl)**

## The Problem

In today's diverse society, people come from a wide variety of backgrounds and therefore speak a wide variety of languages. It is no longer safe to assume that everybody here in the United States speaks English, just like it is no longer safe to assume that everybody in another nation such as France, for example, speaks French. At times, communication between individuals can be difficult due to the language barrier. Granted this problem is more apparent in everyday human communications, but it also is applicable to various forms of speech technology.

Many systems in the speech technology field are limited to working with one and only one language. This restriction may eventually become a hurdle in making speech technology systems practical and useful in societies where not everybody speaks the particular languages that they were designed around. The U.S. is an example of such a society; clearly, not all people in this country can claim to be a proficient and fluent speaker of English. However, many speech technologies seem to make this assumption and simply cannot be of much use to the segment of the population who are not fluent in English.

Recognizing this problem, we have done our course project on attempting to implement a simple language identification system. Our vision was to create a system that could perform real-time language identification of speech limited to the scope of the 3 primary languages of our group members: English for William, French for Velik, and Spanish for Abel. We set a goal of creating this language identification system to be able to receive a relatively brief sample of digitally recorded speech and to be able to accurately and reasonably quickly generate an indication of which of the 3 aforementioned languages applies to the sample. This was the focus of our team's efforts for this semester.

## The Evolution of Our Project

At first, we proposed this project believing that it was going to be quite challenging to realize a system that could perform language identification as we had envisioned. Though we now know that this belief we had was well justified, we also have learned that there are well-known techniques and algorithms in the field of speech processing that make it more feasible than we initially had thought.

In the early stages of this project, all of us devoted many hours to reading a large collection of literature about speech processing and speech recognition. We found papers from numerous researchers in this field, including Marc A. Zissman and Rabiner and Juang, authors of an important book, Fundamentals of Speech Recognition [1]. From this research, we acquired an adequate background about the field to make some decisions about what kinds of techniques and algorithms we wanted to try for our project.

However, reading the literature alone was not the only preparatory work we did prior to rolling up our sleeves and getting involved in the actual implementation efforts. We also were able to meet with Professor Richard Stern of the ECE and CS Departments here at CMU, who has research experience in the field of speech technology. We scheduled meetings with him regularly to discuss our ideas about how we wanted to implement our system, and he offered us valuable advice and feedback concerning these matters. Professor Stern also recommended certain sources of literature that aided this preparatory work. For this, we owe much thanks to Stern for his time and his guidance on this project.

Following these early stages of learning about speech technology methods, we moved to an intermediate phase of deciding upon the techniques and algorithms that we wanted to implement for our language identification system. We considered the time constraints (only about slightly over 2 months for the whole project), the availability of sources of programming code, the difficulty of writing code for techniques and algorithms we would not be able to locate code for, and the effectiveness of these techniques and algorithms based on speaking with Stern and our own reading.
We weighed all these factors into deciding which techniques and algorithms to apply to our system. Later on in this report, we will discuss in further depth the actual methods we employed in our system and motivations for choosing each method.

Once our group made these decisions, it was a matter of working on the implementation. Since we did not want to write code for all of the chosen techniques and algorithms, we spent much time searching for any relevant software that we could find for our project. Scouring the web for these, we managed to find a variety of public domain source codes for all kinds of speech processing applications. Some of these sources on the web include the Center for Spoken Language Understanding at the Oregon Graduate Institute [CSLU at OGI] (http://cslu.cse.ogi.edu/) and the speech group here at CMU (www.speech.cs.cmu.edu/) to name a couple. We ended up using an LBG Vector Quantization codebook generation algorithm implemented by a PhD candidate at University of California San Diego (UCSD) named Aldebaro Barreto da Rocha Klautau Jr. (http://speech.ucsd.edu/aldebaro/). We also used a package of codes for computing Mel Frequency Cepstral Coefficients which was supplied to us courtesy of Rita Singh, a researcher here at CMU with ties to the fairly renown Sphinx speech project. With some modifications, we were able to make use of these source codes.

Once we tested these codes out on PCs, we needed to consider what we wanted the TI C6701 EVM, our DSP board, to do. Since one of the requirements of this project is to make use of the EVM, we had to decide which part of the process would be accomplished by the EVM. We decided that all of the front-end processing (conversion from raw speech data to Mel Frequency cepstral coefficients) would be the job of the EVM, and we set out to adapt the codes from the Sphinx project to suit our needs. This involved writing code for communication between PC and EVM, and also, getting the entire signal processing codes to run on the EVM side. Obviously, some significant changes to the Mel Frequency cepstral analysis codes were made, and we managed to streamline them, eliminating the irrelevant parts that we did not need.

While this EVM side work was being done, we also concurrently made some minor changes to the LBG VQ codebook generation code as well as the VQ classification code. Both of these, as mentioned before, were from Aldebaro at UCSD.
The changes were simply to the input and output aspects, nothing drastic in other words.

Also while EVM side work was being done, we wrote our own codes to generate the probability matrices for the 3 languages. These probability matrices will be defined and elaborated upon later on in this report. Along with this probability matrix formation code, we also had to write our own code to do the final scoring part of our system. This part would utilize the probability data from the matrices to calculate three scores for an input speech sample, one per language. The best score would be the indication of which language the system identifies. More on the details of this procedure will be included later in this report, as well.

The final part of the implementation stage of our project was to integrate all the separate codes, both PC side and EVM side. This proved to be quite challenging since we wanted to implement a real-time system where the input speech could be entered into the EVM directly via its codec, using a microphone. Up to this point, we simply recorded speech files on the PC in .wav format and worked with those rather than getting speech data directly into the EVM with the codec. Issues concerning the C67's speed and memory arose during these efforts. These issues will be dealt with at a later point in this paper.

## Prior Work

From the available final reports for the projects in this course during the past few years, we really have not been able to find a similar project done in the recent past. So rather than discuss any projects that are only remotely close to ours, we will describe the prior work of some researchers in the area of automated language identification. The work of these researchers has been the subject of numerous technical papers that we have read.

One of the most relevant technical papers that we came across is "Comparison of Four Approaches to Automatic Language Identification of Telephone Speech" by Marc A. Zissman [2]. In his paper, Zissman describes 4 different approaches toward the task of automated language identification and compares the results that he obtained from testing these 4 different approaches. These approaches are what Zissman calls: 1) Gaussian mixture model classification (GMM), 2) single-language phone recognition followed by language-dependent, interpolated n-gram language modeling (PRLM), 3) parallel PRLM, and 4) language-dependent parallel phone recognition (PPR). The interested reader is encouraged to read this paper, if they should like to learn about the details of each method. Zissman found that the simpler GMM approach was inferior to the other approaches in terms of performance, as the other approaches involve phone recognition while GMM does not. The best-performing approach, according to Zissman, was parallel PRLM, which makes use of phone recognition based on multiple languages.

Although this paper by Zissman describes actual methods employed in research for automated language identification, we ended up not attempting to implement any of these approaches, at least not in a direct sense. The paper did give us some possibilities for our system, and we explored these possibilities in our discussions with Professor Stern. We eventually settled on an approach that was simpler and more practical for implementation given our time constraints as well as other constraints concerning available hardware and software.

Another technical paper describing approaches toward the problem of automated language identification is "Language Identification with Language-Independent Acoustic Models" by C. Corredor-Ardoy, J.L. Gauvain, M. Adda-Decker, and L. Lamel [3]. The researchers who published this paper compared an approach using language-independent acoustic models to an approach using a parallel language-dependent phone architecture. They found that the two approaches achieved comparable results in the tests that they performed. Again, the interested reader is advised to read this paper should they wish to obtain more in-depth description of the underlying concepts behind these approaches.

The two aforementioned papers discuss approaches that utilize phone recognizers, acoustic models, and the GMM Gaussian mixture model. However, other approaches for language classification include the Hidden Markov Model (HMM) and neural networks. From our readings, we could see that HMM seems to be a reasonably popular approach in many modern speech technology systems. A paper that is specifically about using HMMs for speech classification is "Selective Training for Hidden Markov Models with Applications to Speech Classification" by Levent M. Arslan [4]. HMMs are somewhat similar to the concept of finite state machines to give a very basic idea of how it works. The state transitions are based upon probabilistic data collected from training data.

Another approach for automated language identification is the usage of neural networks. We did not ever give much consideration to the neural network approach, but we did notice that it was mentioned from time to time in the literature that we encountered. For a discussion of the application of neural networks in the area of automated language classification, the reader is referred to the paper of Jerome Braun and Haim Levkowitz at the Computer Science Department of the University of Massachusetts Lowell, titled "Automatic Language Identification with Recurrent Neural Networks" [5]. Neural networks are meant to model the way humans learn and acquire knowledge, so intuitively, they may be quite effective for the task of language identification.

As the approaches of HMMs and neural networks seemed rather complicated to implement given our limited time frame, we decided not to use them. But from our research, they seem to be fairly effective methods, and if we had more time for this project, we might have actually attempted to do either a Hidden Markov Model or a neural network for our system.

For a solid theoretical background on virtually all of these approaches, we recommend Fundamentals of Speech Recognition [1]. It is an excellent overall resource for the theories behind speech recognition technology.

## Discussion of Algorithms and Methods

In the next several pages of this report, we will discuss the different algorithms and techniques that we applied for our system. Our system can be divided into 3 major divisions: the front-end that does the signal processing functions and converts raw speech data to a more useful parametric form; the vector quantization codebook generation and the associated vector quantization classification algorithms to represent input speech as feature vectors that are in the codebook ; and last but not least, the creation of probability matrices for all 3 languages of interest – English, French, and Spanish – and the decision making that is done based on the probability data in these matrices. We will attempt to clearly and concisely describe the algorithms and techniques that we implemented for all 3 of these divisions of our project.

## Front-end signal processing of speech

Front-end signal processing is a common component in all speech recognition systems. So what does this so-called front-end do? And, what is it for? The answer to the former is that the front-end is the part of the system that performs the conversion of the original speech waveform to a parametric representation. Now, you may ask why do this? What good is this parametric representation? Well, a parametric representation ideally provides an accurate representation of the original speech data requiring less storage requirements. Reduction of storage requirements is not the only desirable effect of front-end processing. In addition, the remaining processes in a speech system can utilize these parametric forms rather than the original speech waveforms, cutting down on the amount of computation as well. These are the motivations for front-end processing: reduction of storage and computation, both quite desirable.

From Fundamentals of Speech Recognition, Chapter 3, titled "Signal Processing and Analysis Methods from Speech Recognition", provides a fairly thorough treatment of two popular approaches for implementing front-ends. According to the authors Rabiner and Juang, the two most popular front-end approaches (and ones that in practice have proven to be quite effective) are the bank-of-filters processor and LPC model analyses. We will just summarize what Rabiner and Juang had to say about these two approaches, as the interested reader is encouraged to find out more on their own. The basic concept of the bank-of-filters processor is to pass an input speech signal through a set of bandpass filters, each covering a different range of frequencies. In general, the ranges do overlap, and for each spectrum produced by a bandpass filter, you can do additional steps like additional lowpass filtering, sampling rate reduction, and amplitude compression that also help to reduce the storage and/or bitrates. Filterbanks can be implemented using the types of FIR and IIR filters that we have discussed during this course, and some implementations make use of the Short Time Fourier Transform, a topic of one of the lectures of this course. Typically, the number of bandpass filters in most practical systems is on the order of 8 to 32.

The other approach to designing a speech system's front-end is called LPC analysis, where LPC is an acronym for Linear Predictive Coding. The basic concept of LPC analysis is to make predictions of what sounds will come based on the previous sounds that have been generated. The predictions are made using fairly simple linear equations that utilize some number of previous speech samples to predict the current speech sample. Obviously, the predictions will not be 100% correct, so errors are computed between actual and predicted values. These error values are what get transmitted rather than the predicted values, since error values are smaller than either the predicted or actual values. This is how the savings in storage and/or bitrates are achieved via LPC analysis. Based on Fundamentals of Speech Recognition, some advantages of LPC analysis include less computation, analytical tractability (meaning implementation of the mathematical equations in either hardware or software is reasonable), and proven performance that meets or exceeds the filterbank approach. The key consideration in implementing LPC is tuning the parameters of the equations to get as small errors as possible.

With this theoretical background in place, we now will discuss the front-end of our system. We utilize MFCCs – Mel Frequency Cepstral Coefficients. From our reading of speech recognition literature and discussions with Professor Richard Stern, we decided to implement our front-end as an MFCC analyzer because it seemed to be an approach that is fairly popular in practical systems and works reasonably well. It seems to fit into the former of the two categories briefly discussed before: filterbank processing and LPC analysis. Our front-end code, in fact, uses a Mel-scaling filterbank to accomplish its task of generating the parametric representation of speech data that are called feature vectors. These feature vectors, in our particular case, are simply sets of MFCCs, 13 per frame of speech.

The process of taking an input speech sample and converting it to a set of feature vectors is complex but not too difficult to understand. First, the input speech data in its raw form goes through a step known as pre-emphasis. Pre-emphasis is done in order to spectrally flatten the signal and to make it less susceptible to finite-precision effects when processed later. Typically, pre-emphasis is handled with a first-order FIR digital filter. Then, the resulting pre-emphasized signal is divided up into frames, where each frame is just a segment of the speech data over a very brief amount of time. Frames, in general, can and do overlap, as they do with our system. The frames are then windowed using a Hamming window to minimize signal discontinuities at the beginning and end of each frame. Following windowing, the frames get Fourier transformed to get their Fourier spectrum. The resulting spectrum is then processed through the Mel-scaling filterbank introduced above. This Mel-scaling warps the frequency domain representation to one that was developed by speech researchers Davis and Mermelstein. Hence, the term Mel was brought into existence. This Mel frequency scale is supposed to be a better representation in the frequency domain for human hearing, as it was created in studies concerning the sensitivity of human hearing. Once the Mel frequency spectra are generated, they are put through cepstral analysis (basically taking the logarithm of the spectra) to finally produce the MFCCs.

So, as the alert reader may have noticed, we did not breathe a word about LPC analysis. Alas, we did not use LPC analysis at all for our front-end, despite all the advantages it has according to Rabiner and Juang. We also noticed that in our search for public domain speech processing codes, there was a good amount out there that performs LPC analysis. Clearly, the popularity of LPC analysis was supported by the availability of codes to do it. However, at that point, we had made up our minds to do MFCC analysis instead, and it did not occur to us to change the front-end design to use LPC analysis instead.

The following are some figures taken from Fundamentals of Speech Recognition to illustrate high-level block diagrams of the filter-bank analysis approach and the LPC analysis approach for speech system front-ends. They can be found in the text on pages 74 and 113, respectively.

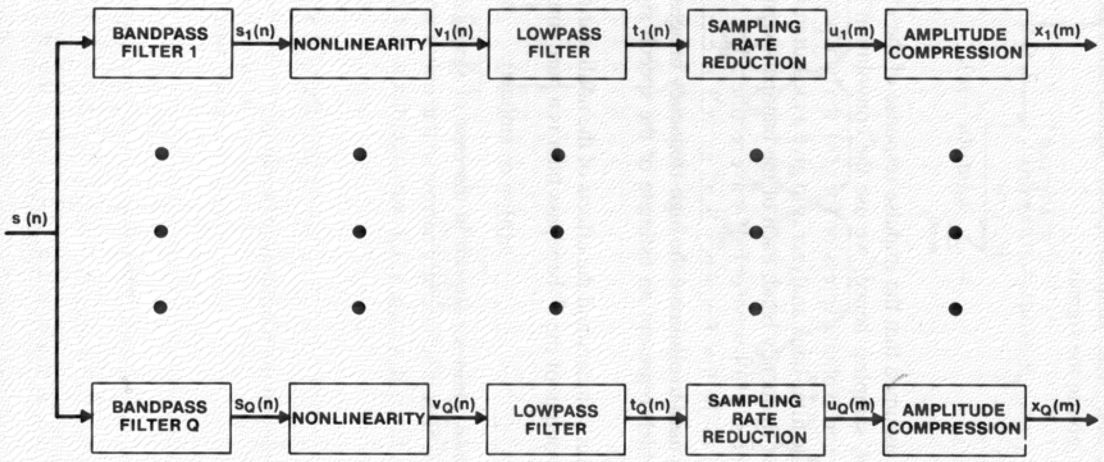


**Figure 3.4** Complete bank-of-filters analysis model.

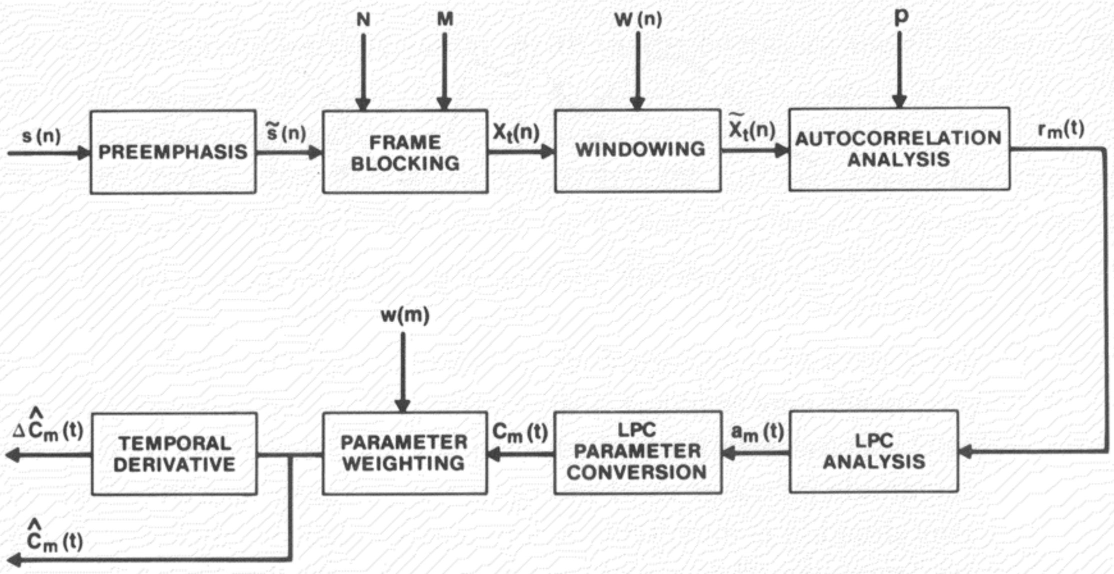


**Figure 3.37** Block diagram of LPC processor for speech recognition.

As mentioned before, the front-end of our system uses MFCCs. The next two figures, also from Fundamentals of Speech Recognition, show the relationship between the Mel frequency scale and the standard frequency scale and a Mel scaling filterbank that can warp a standard frequency spectrum to a Mel frequency spectrum. This would be done in our system's front-end prior to cepstral analysis; after which, the MFCCs are extracted to form feature vectors. These figures can be found in the text on pages 184 and 190 respectively.

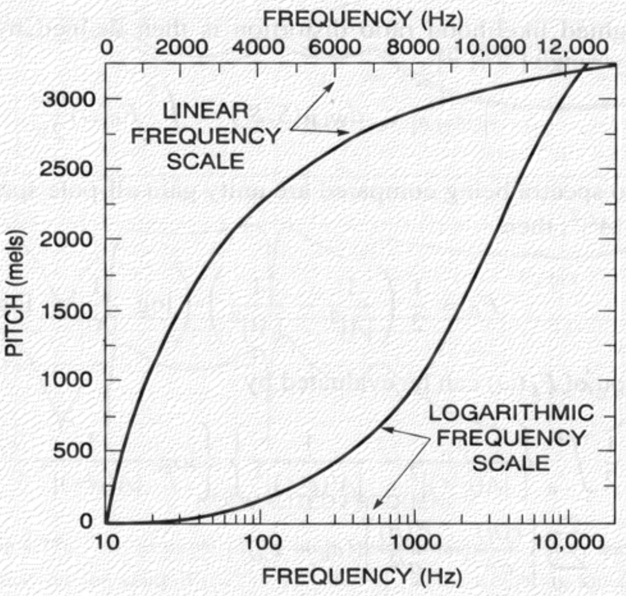


**Figure 4.28** Subjectively perceived pitch, in mels, of a tone as a function of the frequency, in Hz; the upper curve relates the subjective pitch to frequency in a linear scale and the lower curve shows the subjective pitch as a function of the frequency in a logarithmic scale (after Stevens and Volkmann [13]).

Note that there are two different scales used in the above figure, linear and logarithmic. Notice that the logarithmic plot is nearly linear from about a frequency of roughly 1000 Hz up toward 10,000 Hz. This Mel frequency scale is supposed to be a more accurate scale of frequency for representing human hearing. This probably explains the effectiveness of MFCCs in speech technology, and hence, the popularity of MFCCs in practical speech processing systems.

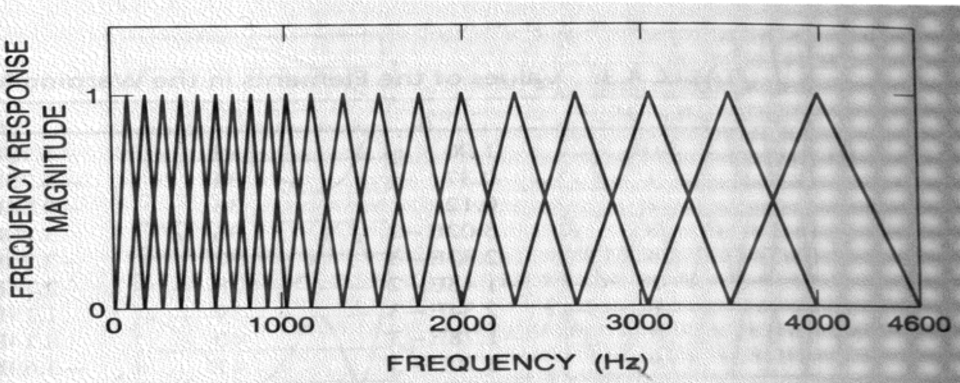


**Figure 4.31** A filter-bank design in which each filter has a triangle bandpass frequency response with bandwidth and spacing determined by a constant mel frequency interval (spacing = 150 mels, bandwidth = 300 mels) (after Davis and Mermelstein [17]).

The individual triangular bandpass filters depicted above correspond to intervals of equal width on the Mel frequency scale. Observe how in this particular Mel scaling filterbank the filters get progressively wider as you proceed from left to right. This supports the notion that humans are less sensitive to higher frequency sounds compared to other animals such as dogs.

## Vector Quantization

The state-of-the-art in feature matching techniques used in speaker recognition includes Dynamic Time Warping (DTW), Hidden Markov Modeling (HMM), and Vector Quantization (VQ). In this project, the VQ approach will be used, due to ease of implementation and high accuracy. Vector quantization is used for language identification in our system. VQ is a process of mapping vectors from a large vector space to a finite number of regions in that space.

A vector quantizer maps k-dimensional vectors in the vector space $R^k$ into a finite set of vectors $Y = \{y_i: i = 1, 2, ..., N\}$. In our project $k = 13$ and $N = 128$. Each vector $y_i$ is called a code vector or a codeword. and the set of all the codewords is called a codebook. Associated with each codeword, $y_i$, is a nearest neighbor region called Voronoi region or cluster, and it is defined by:

$$V_i = \{x \in R^k : \|x - y_i\| \leq \|x - y_j\|, for\,all\; j \neq i\}$$

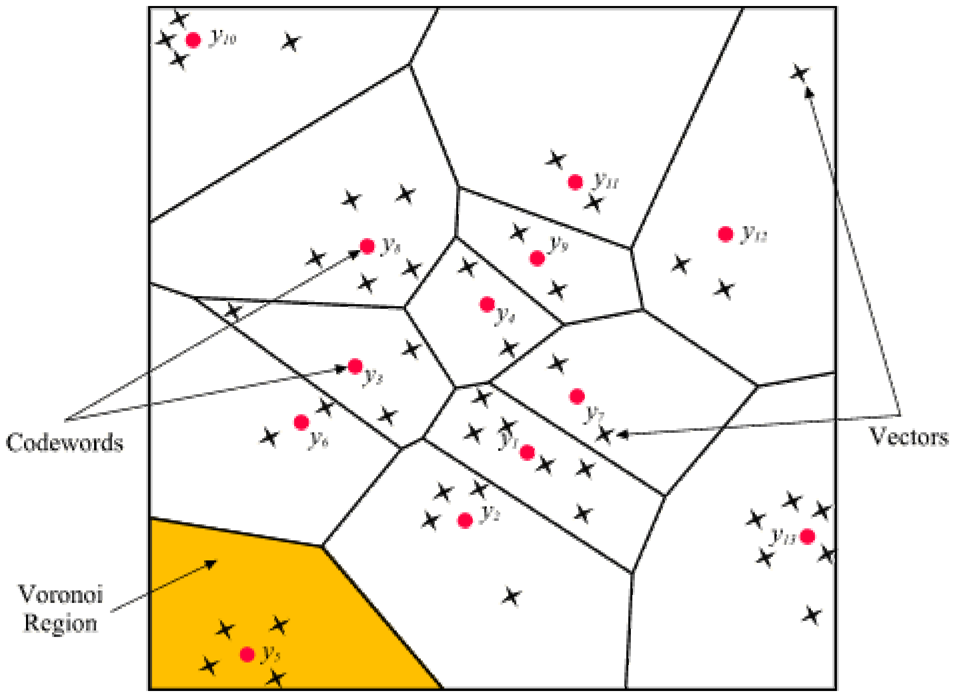
The set of Voronoi regions partition the entire space $R^k$ such that:

$$\bigcup_{i=1}^{N} V_i = R^k$$

$$\bigcap_{i=1}^{N} V_i = \phi \qquad \text{for all } i \neq j$$

The codebook is a kind of library of phonemes, we regroup 128 phonemes in 128 codewords.

As an example we take vectors in the two-dimensional case without loss of generality. Figure 1 shows some vectors in space. Associated with each cluster of vectors is a representative codeword. Each codeword resides in its own Voronoi region. These regions are separated with imaginary lines in figure 1 for illustration. Given an input vector, the codeword that is chosen to represent it is the one in the same Voronoi region.



[ Codewords in 2-dimensional space. Input vectors are marked with an x, codewords are marked with red circles, and the Voronoi regions are separated with boundary lines.]

The representative codeword is determined to be the closest in Euclidean distance from the input vector. The Euclidean distance is defined by:

$$d(x, y_i) = \sqrt{\sum_{j=1}^{k} (x_j - y_{ij})^2}$$

where $x_j$ is the jth component of the input vector, and $y_{ij}$ is the jth is component of the codeword $y_i$.

## Clustering the Training Vectors

As described above, the next important step is to build from the phonemes a VQ codebook, the same for all the three languages. There is a well-know algorithm, namely LBG algorithm [6], for clustering a set of L training vectors into a set of N codebook vectors. The algorithm is formally implemented by the following recursive procedure:

The LBG VQ design algorithm is an iterative algorithm that alternatively solves the above two optimality criteria. The algorithm requires an initial codebook C(0). This initial codebook is obtained by the splitting method. In this method, an initial codevector is set as the average of the entire training sequence. This codevector is then split into two. The iterative algorithm is run with these two vectors as the initial codebook. The final two codevectors are splitted into four and the process is repeated until the desired number of codevectors is obtained.

The program we use in this project implements the Generalized Lloyd Algorithm for designing codebooks. The distortion measure is the mean square error. The first codebook can be obtained starting with N=1 and making splitting (as suggested in the reference above). When there is an empty cell, the most populated cell is splitted. This implementation uses the concept of "partial distortion" to reduce the computational complexity [7]. The savings depend on the vector dimension, but the author of the program got results twice faster. The idea of partial distortion is: the mean square error is a cumulative distortion and one can stop computing a given distortion (skip to the next candidate codeword) if it is already greater than the current smallest distortion.

The algorithm is summarized below.

## LBG Design Algorithm

1. Given $\mathcal{T}$. Fixed $\epsilon > 0$ to be a ``small'' number.
2. Let $N = 1$ and

$$\mathbf{c}_1^* = \frac{1}{M}\sum_{m=1}^{M}\mathbf{x}_m.$$

Calculate

$$D_{ave}^* = \frac{1}{Mk}\sum_{m=1}^{M}||\mathbf{x}_m - \mathbf{c}_1^*||^2.$$

3. Splitting: For $i = 1, 2, \ldots, N$, set

$$\begin{aligned}\mathbf{c}_i^{(0)} &= (1+\epsilon)\mathbf{c}_i^*,\\ \mathbf{c}_{N+i}^{(0)} &= (1-\epsilon)\mathbf{c}_i^*.\end{aligned}$$

Set $N = 2N$.

4. Iteration: Let $D_{ave}^{(0)} = D_{ave}^*$. Set the iteration index $i = 0$.

i. For $m = 1, 2, \dots, M$, find the minimum value of

$$||\mathbf{x}_m - \mathbf{c}_n^{(i)}||^2,$$

over all $n = 1, 2, \dots, N$. Let $n^*$ be the index that achieves the minimum. Set

$$Q(\mathbf{x}_m) = \mathbf{c}_{n^*}^{(i)}.$$

ii. For $n = 1, 2, \dots, N$, update the codevector

$$\mathbf{c}_n^{(i+1)} = \frac{\sum_{Q(\mathbf{x}_m)=\mathbf{c}_n^{(i)}} \mathbf{x}_m}{\sum_{Q(\mathbf{x}_m)=\mathbf{c}_n^{(i)}} 1}$$

iii. Set $i = i + 1$.

iv. Calculate

$$D_{ave}^{(i)} = \frac{1}{Mk} \sum_{m=1}^{M} ||\mathbf{x}_m - Q(\mathbf{x}_m)||^2.$$

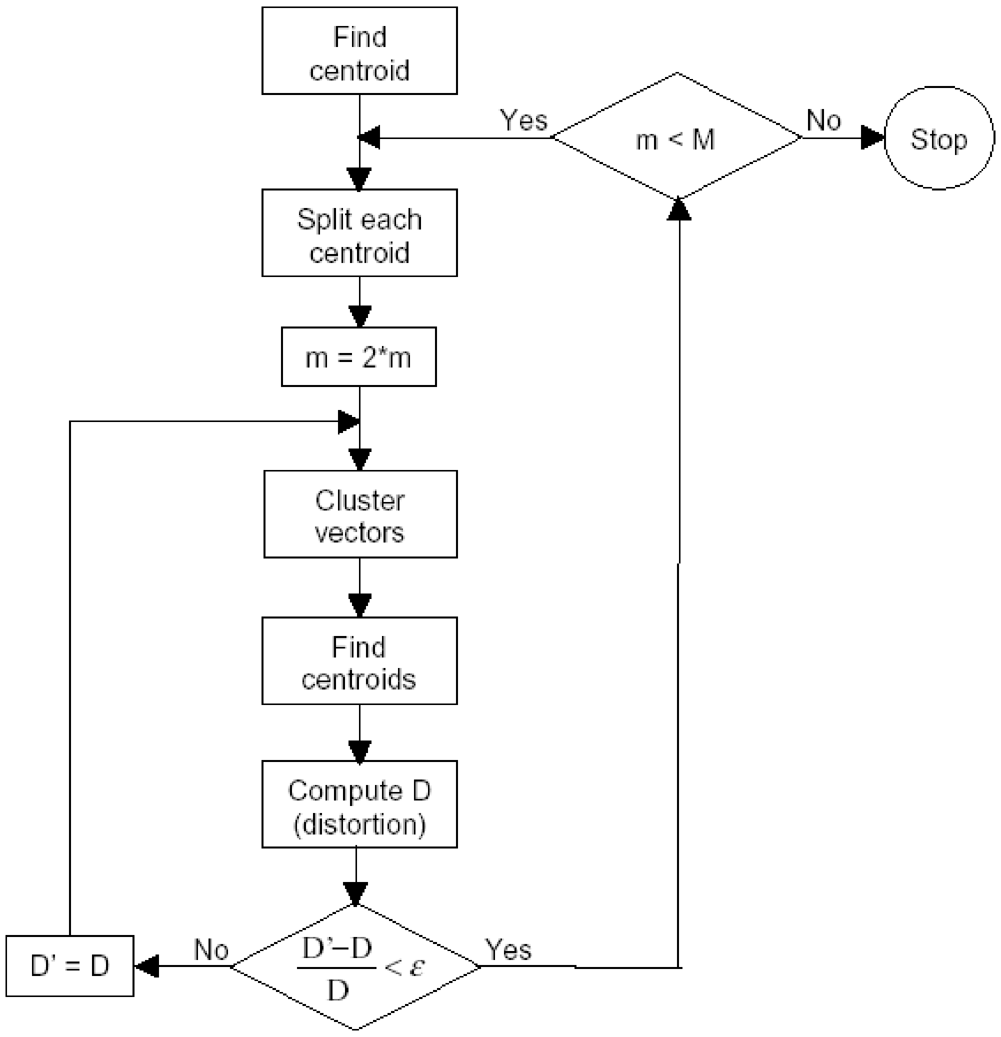
v. If $(D_{ave}^{(i-1)} - D_{ave}^{(i)})/D_{ave}^{(i-1)} > \epsilon$, go back to Step (i).

vi. Set $D_{ave}^* = D_{ave}^{(i)}$. For $n = 1, 2, \dots, N$, set

$$\mathbf{c}_n^* = \mathbf{c}_n^{(i)}$$

as the final codevectors.

5. Repeat Steps 3 and 4 until the desired number of codevectors is obtained.



[Figure. Flow diagram of the LBG algorithm (Adapted from Rabiner and Juang, 1993)]

"Cluster vectors" is the nearest-neighbor search procedure which assigns each training vector to a cluster associated with the closest codeword. "Find centroids" is the centroid update procedure. "Compute D (distortion)" sums the distances of all training vectors in the nearest-neighbor search so as to determine whether the procedure has converged.

## **Training**

To generate the phoneme library for training, we recorded ourselves reading from news sources on the Internet. Abel read aloud in Spanish, William did so in English and Velik in French. Each person read roughly the same amount of time (about 30 minutes) that produced around 200,000 vectors. So, we got more than 600,000 vectors for the phoneme library. A 128-vector codebook was created from these recordings and stored in the system's database.

### Matching

The simplest search method, which is also the slowest, is full search. In full search an input vector is compared with every codeword in the codebook. If there were M input vectors, N codewords, and each vector is in k dimensions, then the number of multiplies becomes kMN, the number of additions and subtractions become MN((k - 1) + k) = MN(2k-1), and the number of comparisons becomes MN(k - 1). This makes full search an expensive method.

The program we used, quantizes the Mel cepstra using a codebook previously designed with the previous program. It implements a full-search. The output is the index of each vectors that we obtain with the front-end program.

## The Probability Matrices and Decision Making Process

The probability matrices and a comparator compose the decision algorithm.

### The Probability Matrices

The basic goal of the matrices is to provide a tool to compare and measure the differences between the testing input data and the training information that was gathered before. There are three matrices corresponding to each of the three languages, English, Spanish, and French. These matrices store the characteristics of each language, the main information stored is the whole set of transitions between two consecutive values in the input data.

### Description of the Matrices

The size of the matrices corresponds to the size of the codebook. Since the codebook has 128 entries, the dimensions of each matrix are 128*128 floats. Each element of the matrix stores a probability value between 0 and 1. The row and columns each designate an entry in the codebook, this entry is called the index. The probability in the row i and column j corresponds to the probability of the index i given the index j. The codebook has entries from 0 to 127. The matrix M has rows and columns from 0 to 127.

Then: $0 \leq i, j < 128$

M[i][j] = P ( entry i | entry j) = probability of the index i given that the index j was present before i.

All of these values are assigned to each matrix during the training phase. In fact, every matrix is trained to store the characteristics of each language.

## Training the Matrices

To train the matrices, the input is first converted to feature vectors, then it is quantized using the codebook entries and a set of indexes is produced. Thus, each speech sample is converted into a set of indexes.

This set of indexes is the input for the matrix. During the training phase, a relatively large speech sample is converted into indexes; these indexes are the training input that is going to fill the matrix.

To compute the probabilities, the following formula is used:

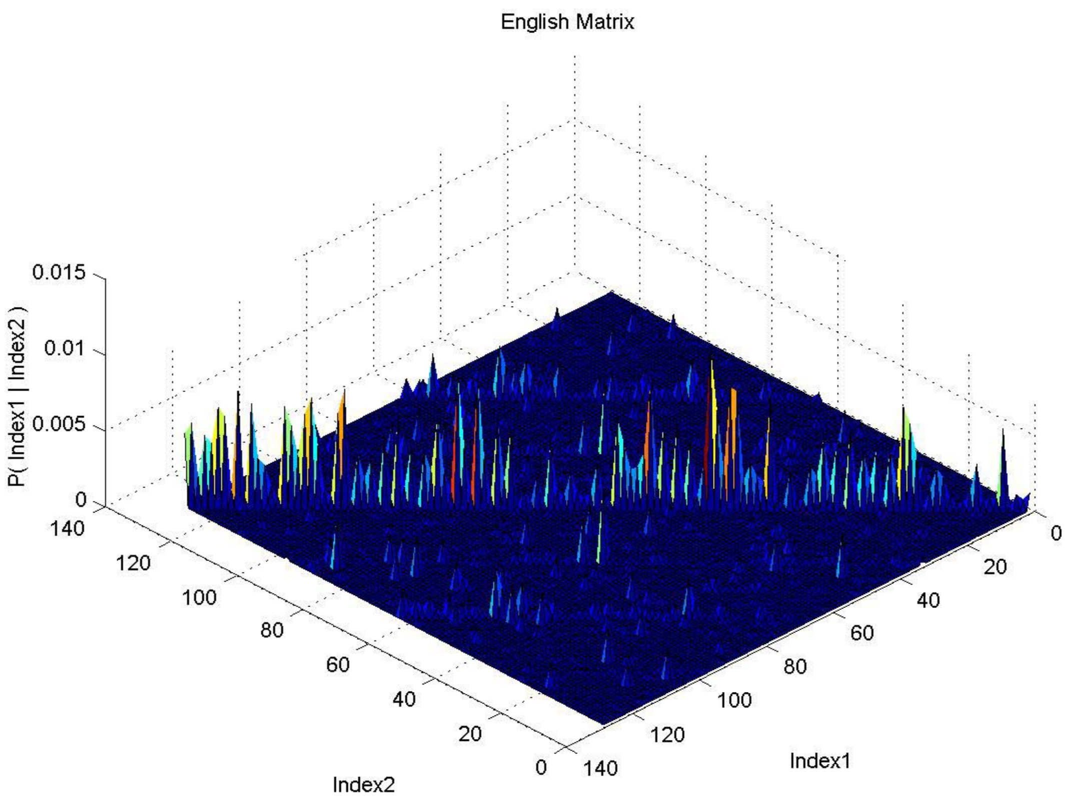
$P(A|B) = P(A \cap B)/P(B) = \#\text{ elements in } (A \cap B) / \#\text{ elements in } B$
Where A: the index is currentindex = i
B: the previous index is lastindex = j
This is done using the following algorithm:

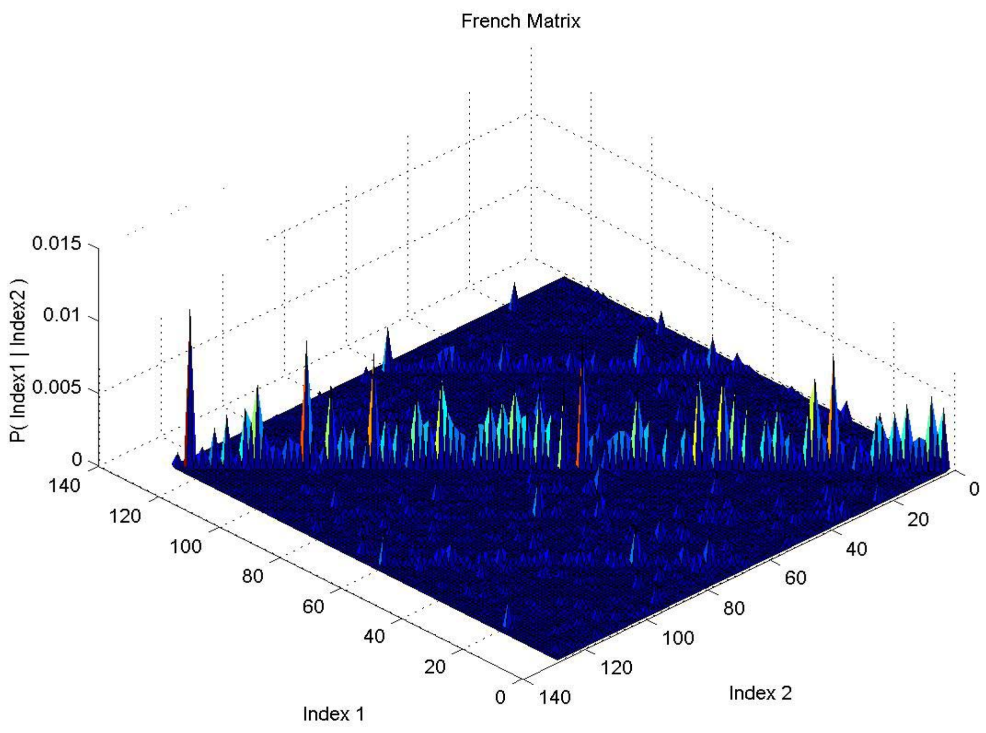
mat is the matrix.

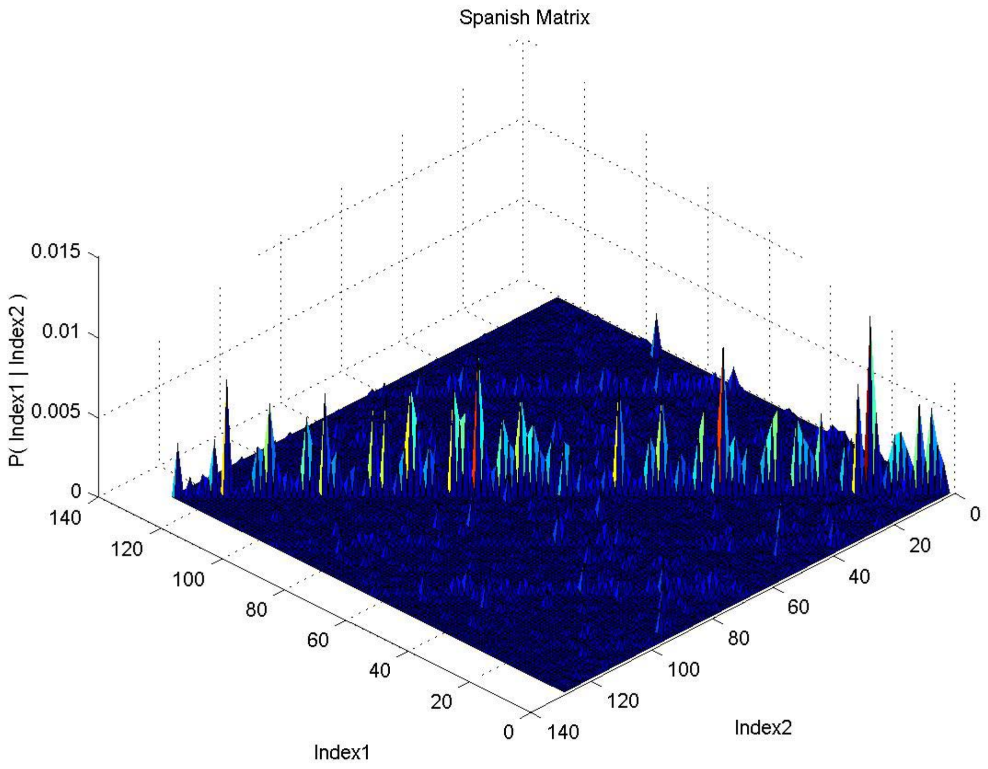
```
#define NCLUSTER 128
int lastindex = indexVector[0];
int currentindex = 0;
for (k=1;k<size;i++) {
        currentindex = indexVector[k];
        mat[currentindex][lastindex] = mat[currentindex][lastindex] + 1;
        lastindex = currentindex;
}

/* Normalizing the matrix */
for (i=0;i<NCLUSTER;i++) {
for (j=0;j<NCLUSTER;j++) {
                mat[i][j] = mat[i][j]/(size-1);
        }
}
```

Finally, this process is repeated for the three matrices and a set of three matrices containing the information for the three languages is created.

Plots of these matrices are shown below.







From an analysis of the similarity between the three matrices, the French and Spanish matrices showed the highest resemblance. On the other hand, the English showed the least resemblance compared with the other two matrices. There could be several reasons for this:

- Differences and similarities in the pronunciation of each language.
- Similarities and differences between the voices of training subjects.
- Timing and noise.

## Testing the Input Data

Once the probability matrices are calculated, the testing could be done using the indexes obtained from the test utterance. In this project, the test utterance lasts 30 seconds. The indexes are used as an input to the matrix and each pair (the current index and the last index) corresponds to a probability. The log of this probability is added to what was obtained before.

Computation of the total probability and decision.

Let's say that we have the following indexes from the test utterance:

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 1 | 4 | 7 | 7 | 8 | 9 | 0 | 6 | 5 | 4 | 2 | 7 | 6 | 5 | 4 | 3 | 2 | 3 | 7 |

The first computation correspond to:
P ( 1|2) = M[1,2) ➔ logP = log(P(1|2))

Then,
P (4|1) = M(4,1) ➔ logP = log(P(4|1))

And so on..

This values are added and a total pseudoprobability is obtained. The logarithms are used to reduce the variations. Indeed, a multiplication could rapidly converge to 0.

This process is repeated for each matrix and the language that has the highest probability is chosen .

Remark.

There is an important issue about the presence of zeros in the matrix. As a rule, a probability matrix should have very few zero values because of the size of the training data. In our case, even though the size of our training data was relatively large, there was still a considerable amount of zero values in the matrices. And since we are working with logarithms, a logarithm of zero poses an important problem. Therefore, we decided to initialize all the values of the matrices with a small EPSILON value. This value was on the order of 10e-21, so the results would not be significantly affected.

The size of the test and training data are shown below:

Size of the training data:
- > 100 Mbytes
- ~ 2 hours of speech equally divided between French, Spanish and English
-

Size of the testing data:
- 480,000 bytes
- 30 sec

## Memory Paging in the EVM

The source code used to determine the feature vectors of the utterance needs to be installed in the EVM. Since the program needs to run considerably fast, the memory management is an important issue. Therefore, we decided to use the following paging:

```
MEMORY
{
 /* Internal Program Memory (IPM) starts at 0x0 */
 ONCHIP_PROG (RX): origin=0x0, length=0x10000 /* 64K */
 /* Internal Data Memory (IDM) starts at 0x80000000 */
 ONCHIP_DATA (RW): origin=0x80000000, length=0x10000  /* 64K */

 /* 256K of onboard RAM starts at 0x00400000 */
   /* SBSRAM_PROG has 100 Kbytes; SBSRAM_DATA has 156 Kbytes */

 SBSRAM_PROG (RX): origin=0x00400000, length=0x19000
 SBSRAM_DATA (RW): origin=0x00419000, length=0x27000

 /* 8M of synchronous dynamic RAM (expansion memory) starts at 0x02000000 */
 SDRAM0 (RW): origin=0x02000000, length=0x400000 /* bank 0 4M */
 SDRAM1 (RW): origin=0x03000000, length=0x400000 /* bank 1 4M */
}

SECTIONS
{
 .vec:      > 0x0             /* Interrupt vector table */
 .text:     > SBSRAM_PROG     /* Executable code and constants */
 .const:    > ONCHIP_DATA     /* Global and static const variables that are
                               * explicitly initialized and that are string literals */
 .bss:      > ONCHIP_DATA     /* Global and static variables */
 .data:     > ONCHIP_DATA     /* Compiler does not use this section */
 .cinit:    > ONCHIP_DATA     /* Tables for explicitly initialized global and
                                 static variables */
 .stack:    > ONCHIP_DATA     /* Stack (for local variables) */
 .far:      > ONCHIP_DATA     /* Variables defined with far */
 .sysmem:   > SDRAM0          /* Memory for malloc functions */
 .cio:      > ONCHIP_DATA     /* System area for printf */
 .ipmtext:  > ONCHIP_PROG     /* section defined in rts6701.lib */
```

The program uses 64 Kbytes of Internal Program Memory and another 64 Kbytes for the Internal Data Memory. The detail for the allocation of the Internal Data Memory is as follows:

ON_CHIP Data Memory needed for 1 second of speech at 8 KHz, 16 bits.

| | |
|---|---|
| 2 buffers for the speech raw data (short): | 32000 bytes |

Memory needed for the calculation includes:

| | |
|---|---|
| MFCC_features_buffer[100 *13](float) | 5200 bytes |
| frame_buffer[240](float) | 960 bytes |

| | |
|---|---|
| hammingWindow[240] (float) | 960 bytes |
| feature_vector[13](float) | 52 bytes |

ON_CHIP Data memory needed exclusively by the program: 38 Kbytes

Therefore, our program can process only a second of speech in the ON_CHIP Internal Data Memory. This restricts the speed of our program because the program needs to communicate with the EVM Memory and the PC for each iteration, it means for each second. If the size of the ON_CHIP Internal Data Memory was larger or the memory access was optimized, the program would run much faster. This could be an idea for a future improvement.

**Speed Issues**

As it was said before, the program needs to run considerably fast so the system can work in real time. Since we needed to read the speech data and process it at the same time, hardware interrupts were used to deal with the incoming data.

Hardware interrupts handle critical processing that the application must perform in response to external asynchronous events. In this case, these asynchronous events are the input speech data at a frequency of 8 KHz. The DSP/BIOS HWI module is used to manage hardware interrupts. The interrupt causes the processor to vector to the ISR address. The address to which a DSP/BIOS HWI object causes an interrupt to vector can be a user routine or the common system HWI dispatcher. In our case, it was a user routine.

The instructions in the routine must be short and fast otherwise data can be lost. Finally, the interrupt takes a second of data and save the samples in a buffer of 8162 short integers. Once the buffer is full, the main program processes it while at the same time the interrupts are saving data in another buffer. That's the reason why there are two buffers to receive the speech data in the program

The program then takes all the speech samples and finds their feature vectors sending them to the PC.

Experimental results:

| | |
|---|---|
| Number of speech samples processed each second: | 8161 |
| Number of feature vectors generated per second: | 100 |
| Number of elements per feature vector: | 13 |
| Number of cycles needed per second of speech | 42,965,473 cycles |
| Time needed to process a second of speech data considering each cycle lasts 7ns: | 0.3 sec |
| Time needed to calculate the feature vectors for the test data (30 sec): | ~ 15 sec |

Improvements.

- All the stuff in the EVM
- Faster, play with the memory
- Threads, perhaps better than interruptions.

## Results Discussion

| | | | | |
|---|---|---|---|---|
| William | English | -9103 | -11425 | -11852 |
| William | English | -8973 | -10297 | -10528 |
| William | English | -9282 | -10014 | -10573 |
| Abel | Spanish | -11280 | -10768 | -10279 |
| Abel | Spanish | -11530 | -9967 | -9308 |
| Abel | Spanish | -10428 | -8798 | -8385 |
| Velik | French | -11453 | -9796 | -10253 |
| Velik | French | -11096 | -9635 | -9916 |
| Velik | French | -11254 | -9696 | -10280 |
| Abel | English | -8229 | -8040 | -8098 |
| Abel | English | -9267 | -8406 | -8202 |
| Abel | English | -7580 | -7199 | -7016 |
| Abel | French | -10149 | -9235 | -9044 |
| Abel | French | -8906 | -8412 | -8402 |
| Abel | French | -11936 | -10189 | -9910 |
| William | Spanish | -8113 | -10262 | -10100 |
| William | Spanish | -8034 | -10293 | -9850 |
| William | Spanish | -7425 | -9676 | -9543 |
| Velik | English | -11183 | -9552 | -10202 |
| Velik | English | -11174 | -9959 | -10896 |
| Velik | English | -11101 | -9571 | -10136 |
| Diane | English | -11228 | -14331 | -14557 |
| Adeep | English | -11060 | -13816 | -15183 |

[ Table of some results, column one is the name of the person, second one is the language spoken and the last column is result for each matrices (English, French and Spanish). ]

We compute three probability-based scores for a given input speech sample, using the data from the three probability matrices (more on this later). The smallest numbers determine which language is spoken. As you can see, when Velik and Abel tried to speak English (not their native languages), the program tended to indicate that they were speaking their native languages. This may be well explained by the fact that Velik and Abel generated French and Spanish training data, respectively. We also tested other French speakers speaking English, but the program always said that they were speaking French. The program seems to perform some kind of accent identification. This is not so bad because for example, you might use this type of program for tourist information. If the program was trained in the native language of the speaker (with authentic accent), he can really be surprised at the results. But it also can be a problem if, for example, for a language, you have a foreign accent. You might be speaking perfectly sensible speech in that language, but the system would pick up on your accent and identify you with the language that is best correlated with the accent. However, this problem can be solved by better training of the codebook and during the generation of the matrices. Better, in this context, means using a very large collection of training data with people of different accents and perhaps other acoustic qualities such as pitch. We also tested with other English speakers and the results were also very good (100%) even though one English speaker is not a native English speaker and does have a perceptible accent.

During the demo, the results were very good (100%) but we had some problems. We got some problems just before the demo; the problem was just the difference in accents for Spanish between our test subject from Chile and Abel, our group member and the voice behind the Spanish training, who is from Bolivia. But the result was not so bad. We also experienced problems with the EVM and the gain of the microphone. The noise was too strong, and the result was completely wrong. This was fixed by adjusting the gain parameters of the codec microphone input.

In summary, the bigger our training data, the better the results will be. We are very surprised about such positive results, as we did not expect 100% success.

## <u>The Role of the EVM</u>

In our system, the role of the C6701 is to handle all of the front-end processing of speech data. As described before, this front-end processing is the conversion from the original digital speech data to feature vectors of Mel Frequency Cepstral Coefficients. Since the C6701 is a specialized digital signal processor, we thought that it was natural to incorporate the EVM into our project as the device for performing all of the front-end processing.

Despite the fact that we were told that the C6701 may be overkill for speech processing applications, we still faced several problems with using it. Issues concerning the availability of memory were common during our efforts toward the end of this project in integrating all the separate modules together. These problems arose when we worked on getting the EVM to be able to accept input speech data from a microphone through its codec, in order to prepare for a “real-time” demonstration.

While testing during this phase of the project, we encountered problems about running out of memory on the EVM.
At one point, we found that we were capable of getting the EVM to receive recorded speech directly from its codec’s microphone input for perhaps 9-10 seconds. Then, the

code on the EVM would halt, and we would get messages within Code Composer Studio telling us that no more memory could be allocated on the EVM.

We resolved this issue when we came to the realization that we were repeatedly allocating memory for some data buffers on the EVM side when it was necessary to allocate memory for them only once. These buffers were for such things as a Hamming window and a Mel-scaling filterbank, both of which require memory allocation once and need to be set up with data only once. The appropriate changes to the code were made, and then this issue became an issue no longer. But having an additional daughter board connected to the EVM would have been helpful.

Memory issues were the primary ones that we experienced with the C6701, but speed was also a concern at times. Mostly, we were concerned with its speed in doing the front-end processing for our large amounts of training data. At first, we would run the front-end processing codes that we received from the Sphinx group in a Unix platform because they were designed to be executed in a Unix platform. We found that these original codes ran quickly and could generate feature vector data from original raw speech data in a reasonable amount of time.

Using these codes, we adapted them to work on PC and EVM together since we had made the decision that the C6701 was the designated front-end processor. We had to make some pretty significant changes to the codes, eliminating much of the irrelevant stuff that we did not need to save in both memory and speed. We expected the resulting codes to be able to produce feature data from raw speech data at least as fast as, if not faster than, the original codes running in Unix. This was not the case at all. The adapted codes utilizing both PC and EVM were actually slower and this left us somewhat puzzled.

There are some ideas that we have which may explain why this happened. One idea is that the overhead in communications between the PC and the EVM may be the bottleneck slowing the processing down. This seems reasonable as the data must be divided into chunks and each chunk must be passed from the PC to the EVM for processing.
Then, the PC must wait for the EVM to finish, and then, it receives the feature data for the chunk just delivered to the C67. This procedure repeats, and every time data is transferred between the two sides, there is the overhead of synchronization messages and data transfers via the host port interface (HPI) of the EVM.

Another theory is our lack of effort into optimization of the EVM side codes. It may be that the front-end processing codes we developed can run as fast as the codes that they were derived from. However, making this happen would require careful optimizations of the codes, particularly on the EVM side. Given that we had limited time and the fact that we devoted much time to simply getting working codes on the C6701, we did not spend much, if any, time on optimizations that would lead to higher performance in speed and perhaps lower memory requirements.

A third possible explanation for the slower performance of our front-end processing codes compared to the originals is perhaps the most simple one of all. It is conceivable that the TI C6701 is simply technologically inferior to other computing hardware we have available, like the lab PCs, Unix workstations, new laptops, etc. We do live in a day and age where advances in computing technologies are rapid and make what was once a breakthrough totally obsolete.  This may be the reason or at least a contributing reason to the disparity in performance.

## References

[1] Fundamentals of Speech Recognition by Lawrence Rabiner and Biing-Hwang Juang. Englewood Cliffs, New Jersey: Prentice Hall, 1993.

[2] "Comparison of Four Approaches to Automatic Language Identification of Telephone Speech" by Marc A. Zissman. IEEE Transactions on Speech and Audio Processing, Volume 4, Number 1, January 1996.

[3] "Language Identification with Language-Independent Acoustic Models" by C. Corredor-Ardoy, J.L. Gauvain, M. Adda-Decker, and L. Lamel. Spoken Language Processing Group, LIMSI-CNRS, BP 133, Orsay cedex, France.

[4] "Selective Training for Hidden Markov Models with Applications to Speech Classification" by Levent M. Arslan.
IEEE Transactions on Speech and Audio Processing, Volume 7, Number 1, January 1999.

[5] "Automatic Language Identification with Recurrent Neural Networks" by Jerome Braun and Haim Levkowitz. IEEE, 1998.

[6] "An Algorithm for Vector Quantiser Design" by Linde, Y., A. Buzo, and R. M. Gray, IEEE Trans Communications, Vol. 28, pp.84-95, JaA. Gersho and R. M. Gray, Vector Quantization and Signal Compression.

[7] "An Improvement of the Minimum Distortion Encoding Algorithm for Vector Quantization" by Bei, C. & Gray, R. IEEE Transactions on Communications, 33(10):1132-3, Oct 1985.

[8] ``Vector Quantization'' by R. M. Gray. IEEE ASSP Magazine, pp. 4--29, April 1984.

## Web References

The Center for Spoken Language Understanding (CSLU) at the Oregon Graduate Institute (OGI): http://cslu.cse.ogi.edu/. This is a fairly well known research center on speech technologies.

PhD Student Aldebaro Barreto da Rocha Klautau Jr. at the University of California San Diego (UCSD): http://speech.ucsd.edu/aldebaro/. This is where we found source codes for all the Vector Quantization algorithms.

The Speech Group here at Carnegie Mellon University: http://www.speech.cs.cmu.edu/. Right here, much research has been done on speech processing and speech technologies. Also, there are numerous links to other speech resources on the Internet