Speaker Verification Based On Speech Encoder Parameters

Francisco Javier Garcia Torcelly

Stockholm, October 26th, 2001
Abstract

Speaker Verification is the task of recognizing people by their voices. Speaker Verification can be used efficiently in applications where a high level of security is needed (telephone banking, access to personal information, etc...). The main problem that appears in Speaker Verification is the loss in performance when conditions in enrolling and testing are not the same. In this thesis, we put focus on the robustness of the system in mobile applications, and different speech parameterizations are studied: Linear Prediction Cepstral Coefficients (LPCC), Line Spectral Pair (LSP) and Adaptive Component Weighting (ACW). The amount of training data and the complexity of the models have been used as parameters. The results show that LPCC outperforms LSP and ACW, but ACW is more robust. LSP is not a good parameterization to perform Speaker Verification.
Preface

This Master of Science Thesis has been done at the Department of Speech, Music and Hearing at the Royal Institute of Technology (KTH), Stockholm. This five months work is the result of cooperation between KTH and Ericsson Mobile Communications AB, Lund.

   Examiner: Prof. Björn Granström
   Advisor at KTH: Tekn. Dr. Mats Blomberg
   Advisor at Ericsson: Tekn. Dr. Fisseha Mekuria

Acknowledgements

This work has been supported by many people to whom I wish to express my gratitude.

I wish to thank Dr. Mats Blomberg for giving me the opportunity to join TMH and whose guidance gave many ideas and greatly helped me "see the big picture". I wish to thank PhD student Håkan Melin for his enthusiasm and help with the programming.

The fruitful conversations with Dr. Fisseha Mekuria of Ericsson Mobile Communications AB, Lund, and his help with all the practical matters should also be acknowledged.

A special thanks goes to the other members of the department of Speech, Music and Hearing with whom I shared really good moments, specially to Pal and Tobias.

I would like to dedicate this Master Thesis to Tamara, for her love, support and understanding during all these months. I will love you forever.

Finally, the immense and never-ending support from my parents made it all possible.
## Contents

1 Introduction .............................................. 1  
   1.1 Speaker Verification in Telecommunication ................. 1  
   1.2 Speaker Verification Systems ................................. 2  
   1.3 Previous Work ........................................... 3  
   1.4 Scope of the Thesis ....................................... 4  
       1.4.1 Problem Background .................................. 4  
       1.4.2 Scope .................................................. 4  

2 Speech Parameterizations ................................. 5  
   2.1 Linear Prediction Analysis ................................. 5  
   2.2 All-pole Based Speech Parameterizations ................. 6  
       2.2.1 Linear Prediction Cepstrum .......................... 6  
       2.2.2 Line Spectral Pair .................................. 6  
   2.3 Pole-zero Based Speech Parameterizations ............... 7  
       2.3.1 Adaptive Component Weighting ....................... 8  
       2.3.2 Adaptive Component Weighting 2 ..................... 9  
       2.3.3 Postfilters ........................................... 10  
   2.4 Further Coefficients .................................... 10  
       2.4.1 Energy ............................................... 10  
       2.4.2 Delta Coefficients .................................. 10  
       2.4.3 Acceleration Coefficients ......................... 11  

3 Experimental Settings .................................. 13  
   3.1 The GIVES Environment .................................. 13  
       3.1.1 Implementation of ACW in GIVES ................... 13  
       3.1.2 Implementation of LSP in GIVES ..................... 15  
   3.2 Experimental Settings .................................. 16  
       3.2.1 The YOHO Database .................................. 16  
       3.2.2 The CAVE Project .................................. 18  
       3.2.3 Experimental Conditions ............................ 18
CONTENTS

4 Results .............................................. 21
  4.1 Reference Experiments ............................... 21
    4.1.1 Linear Prediction Cepstral Coefficients ........... 22
    4.1.2 Line Spectral Pair ................................ 23
    4.1.3 Adaptive Component Weighting Cepstral Coefficients 24
  4.2 Results ........................................... 24
    4.2.1 Linear Prediction Cepstral Coefficients ........... 25
    4.2.2 Line Spectral Pair ................................ 27
    4.2.3 Adaptive Component Weighting ....................... 28
  4.3 Comparison Between the Three Methods ............... 30

5 Discussion ........................................... 31
  5.1 Linear Predictive Cepstral Coefficients ............... 31
  5.2 Line Spectral Pair ................................... 32
  5.3 Adaptive Component Weighting ......................... 32

6 Conclusions .......................................... 35
  6.1 Conclusions ........................................ 35
  6.2 Further Research .................................... 35

7 Memory Requirement ................................... 37
  7.1 Memory Requirements ................................ 37

8 Appendix .............................................. 39
Chapter 1

Introduction

Intrinsic characteristics of a person, such as voice, fingerprints, retinal pattern or genetic structure can be used in many different applications. These include forensic applications, secure access to private information or personal services, and the automated processing of the recognised information.

The use of voice biometrics in applications over the telephone network has increased in recent years, and could make possible new services in the future. For this to be possible, a high reliability on the recognition task is a must.

Growth in mobile applications and necessity of more reliable methods to ensure security on them require from robustness in the speaker verification systems implemented on mobile terminals. In this thesis, we put focus on the use of speech encoder parameters to perform robust speaker verification. A number of speech parameterizations based on all-pole and pole-zero models are studied, and a set of experiments is carried out on them. The results are then compared to those obtained within CAVE\textsuperscript{1}. The following of this chapter briefly recaptures speaker verification, and highlights the most relevant previous work. This chapter concludes with the scope of this thesis.

1.1 Speaker Verification in Telecommunication

Speaker verification (SV) is the task of identifying people by their voices. When using voice as a key to access a service, this task consists on accepting or rejecting the identity claim of the speaker. A SV system is said to be text-dependent if the same utterance is used in both enrolling and testing, while it is said to be text-independent when the key phrase is prompted while testing.

A number of factors contribute to make the verification task more difficult. These can be, for example, the ambient noise, the quality of the

\textsuperscript{1}CAller VErification in Banking and Telecommunications, EU Project DGXIII LE1-1930
microphone used and human changes (e.g., time-of-day voice changes and minor head colds). All of them will lead to an environment quite different from the one during the enrolment of the client of the system, where the conditions are controlled.

1.2 Speaker Verification Systems

A generic SV system can be divided into four blocks [1], each of them performing a different task.

![Speaker Verification System Diagram](image)

Figure 1.1: Speaker Verification System

The **analysis module** converts the speech signal into a sequence of vectors or frames, being these a certain representation of the signal.

The **modelling module** models the speaker’s enrolment utterances, and extracts a number of parameters that characterize the speaker. These can be, for example, linear predictive cepstral coefficients. The modeling is usually done by estimating parameters in Hidden Markov Models\(^2\) (HMM) of the speaker’s utterances from the training data. Each state models a (quasi-stationary) spectral zone characteristic of a particular word. Initial states have to be fed into the iterative reestimation process. Reestimation of transition probabilities, observation means and variances try to maximize the probability that the HMM would emit a given training sequence.

The **scoring module** measures the similarity between a parameterized test utterance and a model. As a measure of likelihood, the likelihood ratio is commonly used. It is defined as:

\[
LLR(Y_1^N) = \log \frac{\mathcal{L}(Y_1^N|\mathcal{X})}{\mathcal{L}(Y_1^N|\Omega)}
\]

\(^2\)Markov chains used to represent a given speech segment in a stochastic manner.
1.3. PREVIOUS WORK

where $\mathcal{L}(Y_1^N|\mathcal{X})$ and $\mathcal{L}(Y_1^N|\Omega)$ are the probability density function for the hypothesis $\mathcal{X}$ (speaker model for the claimed speaker) respectively $\Omega$ (background model) for the observed speech segment $Y_1^N$.

This score will be used by the decision module to decide whether the user is the one claimed to be or not, and accept or reject him/her. This decision is based on a threshold.

Not all of them have to be used in both enrolling and testing. During the enrolling sessions, the user will be prompted to say a few words or phrases. These will be used by the system to build a model of the user, by extracting a certain number of parameters. In this case, only the analysis and modeling modules are used. On the other hand, when testing, the four blocks are used.

1.3 Previous Work

Speech is our most natural way to communicate, and because of it spoken language interfaces to computers is a topic that has fascinated engineers and speech scientists for over five decades. The ability to converse freely with a machine is for many the ultimate challenge to our understanding of the processes involved in human speech communication.

Human language technology aims to enable people to communicate with machines using natural communication skills. Research and development in the area include the coding, recognition, interpretation, translation and generation of language.

Speaker verification and recognition have been the subject of active research for many years, and have many potential applications where propriety of information is a concern. Several sites in Canada, Europe and Japan have been researching spoken language understanding systems. The ESPRIT SUNDIAL project, concluded in August 1993, involved several partners, and developed systems for train timetable queries in German and Italian, and flight queries in English and French. These systems are described in [2]. In the ARPA program, the air travel planning domain has been chosen to support evaluation of spoken language systems. Here speech and language are spontaneous, and the vocabulary for this system is around 2000 words. In the December 1994 benchmarks, the speech recognition error rates was about 12% to 25%. The CAVE project aims to design and test two SV systems oriented to telephone banking operations. A number of different techniques have been investigated to this end. The best result obtained is a gender balanced sex independent equal error rate (GBSI-EER) of 0.028%.
1.4 Scope of the Thesis

1.4.1 Problem Background

Speech, as many other biometrics, can efficiently be used as a way to identify persons. This is obvious, as biometrics cannot be lost, stolen, replaced or forgotten, providing this way with a high grade of security. But extracting features that represent a speaker uniquely is a difficult task. Actually a number of parameterizations are used, most of them based on a Linear Prediction (LP) analysis algorithm by the speech coder. The aim of any of these parameterizations is to extract from the speech signal those features that are characteristic to the speaker, leaving apart those common to all of them.

1.4.2 Scope

This master thesis project aims to test the LP parameters and their derivatives for speaker verification in a mobile phone. All-pole and pole-zero based speech parameterizations are tested, Line Spectral Pair (LSP) and Adaptive Component Weighting (ACW) respectively. The LP analysis part in the speech encoders of modern digital mobile phones is used to generate the basic parameters for robust SV.
Chapter 2

Speech Parameterizations

In this chapter we describe some of the speech parameterizations commonly used in speech encoders. These speech parameterizations can be divided in two groups, all-pole and pole-zero based speech parameterizations.

The first step when processing speech signals is a linear predictive (LP) analysis. The present speech sample is predicted as a linear combination of the past samples. This way the spectral envelope is parameterized.

After the LP analysis, an extraction of features is performed, in order to discriminate between speakers. The selection of the feature to be extracted is the main factor for a speaker verification system to perform successfully. Different features have a different behaviour in different test conditions. Some of them suffer from high sensitivity to channel variations, while others are more sensitive to the additive noise. Because of this, the election of one or another feature in a speaker verification system depends on the final application.

2.1 Linear Prediction Analysis

Linear predictive analysis of speech assumes that the speech signal is stationary within short periods of time (known as frames). The representation of this is an all-pole filter,

$$H(z) = \frac{1}{A(z)} = \frac{1}{1 + a_1 z^{-1} + a_2 z^{-2} + \ldots + a_M z^{-M}}$$ (2.1)

The result of filtering the speech signal using $A(z)$ is the LP residual error, and it is free from near-sample redundancies.

The $a_i$ coefficients, computed by minimizing the mean square prediction error summed over the whole analysis window, are the LP representation of the frame. $M$ is the number of poles in the transfer function, and is known as the order of the LP analysis.
In [3] a detailed description on the way to compute the LP coefficients can be found. The procedure consists basically of the computation of the autocorrelation of the windowed signal and the resolution of the set of equations (2.2) using the Levinson-Durbin algorithm.

\[
\sum_{k=1}^{M} a_k r(|i - k|) = -r(i) \text{ for } i = 1, \ldots, M
\]  

(2.2)

### 2.2 All-pole Based Speech Parameterizations

In this section some of the most common features based on an all-pole transfer function are presented.

#### 2.2.1 Linear Prediction Cepstrum

Linear Prediction Cepstrum\(^1\) (LPC) has been widely used in speech recognition and verification. It has been proved to be the best when training and testing is done on clean speech [4]. It gives a good approach to the spectral envelope of the speech signal. The problem is that differences between the conditions in training and testing sessions result in a lower performance in the recognition task, decreasing this way the robustness\(^2\).

#### 2.2.2 Line Spectral Pair

Line Spectral Pair (LSP) is an alternative way of representing LPC [5]. Assume that the LPC have been computed. Then the inverse filter \(A(z)\) polynomial is decomposed into two polynomials, \(P(z)\) and \(Q(z)\). The roots of these new polynomials are the LSP coefficients.

The polynomials are the following:

\[
P(z) = A(z) + z^{-M-1}A(z^{-1})
\]  

(2.3)

\[
Q(z) = A(z) - z^{-M-1}A(z^{-1})
\]  

(2.4)

and are symmetric and antisymmetric, respectively. Their roots are on the unit circle, and they alternate each other [6]. This property can be used to compute them efficiently (the roots of \(P(z)\) and \(Q(z)\) are the LSP frequencies).

Note that \(P(z)\) has a root \(z = -1\) and \(Q(z)\) has a root \(z = 1\). Two new polynomials are defined by eliminating these roots:

\[
P'(z) = \frac{P(z)}{1 + z^{-1}}
\]  

(2.5)

---

\(^1\) The cepstrum is defined as the inverse Fourier transform of the log-spectum

\(^2\) A system is said to be robust when it performs successfully even if the speech signal is corrupted by noise and/or affected by communication channel effects
2.3. **POLE-ZERO BASED SPEECH PARAMETERIZATIONS**

\[ Q'(z) = \frac{Q(z)}{1 - z^{-1}} \]  \hspace{1cm} (2.6)

These new polynomials have \( M/2 \) roots on the unit circle each, and they alternate, so they can be expressed as:

\[ P'(z) = \prod_{i=1,3,\ldots,M-1} (1 - 2q_i z^{-1} + z^{-2}) \]  \hspace{1cm} (2.7)

\[ Q'(z) = \prod_{i=2,4,\ldots,M} (1 - 2q_i z^{-1} + z^{-2}) \]  \hspace{1cm} (2.8)

where \( q_i = \cos(\omega_i) \) (LSP coefficients in the cosine domain), being \( \omega_i \) the Line Spectral Frequencies (LSF).

Due to the symmetry of the two polynomials, only the first \( M/2 \) coefficients need to be computed, the rest being found by the recursive relations:

\[ p'(i+1) = a_{i+1} + a_{M-i} - p'(i) \]  \hspace{1cm} (2.9)

\[ q'(i+1) = a_{i+1} - a_{M-i} + q'(i) \]  \hspace{1cm} (2.10)

The evaluation of \( P'(z) \) and \( Q'(z) \) is done using the Chebyshev polynomials.

2.3 **Pole-zero Based Speech Parameterizations**

In order to enhance the robustness of the SV system we will try to convert the all-pole transfer function derived from the LP analysis into a pole-zero function [7]. The feature used is the cepstrum of the new transfer function.

As stated in [8], the mean-square difference between two cepstral vectors is directly related to the mean-square difference in the magnitude spectra of the transfer functions from which the cepstral vectors were derived from. Our goal is to build a robust SV system, and this can be done by parameterizing the speech signal such that the difference in the magnitude spectra decreases when it is corrupted by noise and/or affected by channel variations, with respect to the LPCC case.

Several approaches can be made to perform the transformation from the LP transfer function into a pole-zero model. The reason why a pole-zero model is not obtained directly from the speech signal is that the performance decreases [7].
2.3.1 Adaptive Component Weighting

The spectrum of a speech waveform has two main components, narrow-bandwidth and broad-bandwidth components. The former ones correspond to the *formants*, while the latter are generally due to channel and glottal characteristics. Obviously, the narrow-bandwidth components are the ones we are interested in when working on SV, so it would be very useful to be able to emphasize them and, at the same time, attenuate the broad-bandwidth components. This is the goal of the Adaptive Component Weighting (ACW) Cepstral coefficients [9]: to minimize the intraspeaker variance and at the same time maximize the interspeaker variances.

Cepstral weighting techniques account for the sensitivity of the low-order cepstral coefficients to the overall spectral slope and the sensitivity of the high-order cepstral coefficients to noise [9]. Adaptive weighting techniques adapt to the variations of the distortion introduced on the speech signal, deemphasizing the irrelevant variations of the LP cepstral coefficients on a frame-by-frame basis.

For a given frame of speech, the LP model can be expressed as:

\[
H(z) = \frac{1}{A(z)} = \sum_{i=1}^{P} \frac{r_i}{1 - z_i z^{-1}}
\]  

(2.11)

where \( r_i \) are the residues on the poles \( z_i \).

As shown on [9], the sensitivity of a pole to errors in the LP coefficients is proportional to the residues. If we assume that the distortion will affect all the LP coefficients equally, the poles of the components with larger residues will be more strongly affected. A way to avoid this is by normalizing each component by its residue, leading to reduced effects of distortions on the feature set.

The sensitivity of a feature set to channel variations is empirically evaluated in [9]. The conclusion to these experiments is that, under channel variations, the broad-bandwidth components show undesirable variability that results in a mismatch between testing and training patterns.

To enhance the robustness of the LP spectrum, and considering these observations, some modifications can be achieved. These should try to attenuate the contribution of the broad-bandwidth components. This can be done by normalizing the components by the residues \( \{r_i\} \), which results in a modified spectrum which we refer to as the Adaptive Component Weighting spectrum.

The resulting transfer function is no longer an all-pole (autoregressive, AR) transfer function, as a moving average (MA) component appears. The MA filter introduced can be seen as a Finite Impulse Response (FIR) filter, which creates a spectrum whose components’ peak values are inversely proportional to their bandwidths.
2.3. POLE-ZERO BASED SPEECH PARAMETERIZATIONS

Experiments in [9] reveal that the mismatch between the LP spectra before and after processing through a channel is much larger than that between the corresponding ACW spectra.

In the cepstral domain, the introduced MA filter results in a subtractive component to the all-pole cepstrum. Lets denote \( N(z) \) the numerator of the ACW spectrum,

\[
N(z) = P(1 + \sum_{i=1}^{P-1} b_i z^{-i})
\]  

(2.12)

As shown in [12], \( N(z) \) is merely the derivative of the denominator polynomial coefficients. This makes the computation more efficient, and speeds up the parameterization process.

The subtractive cepstral component, \( \{c^h_n(m)\} \), which is associated with \( N(z) \), can be obtained by its recursive relation with \( b_k(m) \):

\[
c^h_0(m) = -b_1(m)
\]  

(2.13)

\[
c^h_n(m) = -b_n(m) + \sum_{k=1}^{n-1} \left( \frac{k}{n} - 1 \right) b_k(m)c^h_{n-k}(m), \text{ for } 1 < n \leq P - 1
\]  

(2.14)

\[
c^h_n(m) = \sum_{k=1}^{n-1} \left( \frac{k}{n} - 1 \right) b_k(m)c^h_{n-k}(m), \text{ for } n > P - 1
\]  

(2.15)

This way, the ACW cepstrum is given by

\[
\hat{c}_n(m) = c_n(m) - c^h_n(m)
\]  

(2.16)

These new cepstral coefficients are then used to perform SV.

2.3.2 Adaptive Component Weighting 2

The LP derived transfer function can be seen as a connection of \( P \) first order filters in cascade. If these sections are connected in parallel, the resulting transfer function is the overall pole-zero one for the ACW method [8].

If we consider \( P/2 \) second-order filters in parallel, then another pole-zero transfer function is obtained. This is known as the Adaptive Component Weighting 2 (ACW2) approach.

The way to pair up the poles is the following: each complex pole will be paired up with its complex conjugate, and any remaining poles are also paired up. The reason to proceed this way is that the impulse response of a second-order section specified by a complex conjugate pole pair is a damped sinusoid. This way, the speech signal is represented as a superposition of amplitude modulated sinusoids. The resulting pole-zero model is more natural for speech. Again, the cepstrum of \( N(z)/A(z) \) is used as feature vector. ACW2 is beyond the scope of this thesis work.
2.3.3 Postfilters

It is a fact that more noise can be perceptually tolerated in the formant regions (spectral peaks) than in the spectral valleys [10]. Another family of pole-zero transfer functions is introduced in [7] relying on the concept of a postfilter. The postfilter is obtained from $A(z)$ and its transfer function is given by

$$H_{pf} = \frac{A(z/\beta)}{A(z/\alpha)} \text{ for } 0 < \beta < \alpha < 1. \quad (2.17)$$

The spectrum of $H_{pf}(z)$ emphasizes the formant peaks. Its cepstrum is equivalent to the weighting of the LP cepstrum by a factor $(\alpha^n - \beta^n)$. This way, the resulting effect is a weighting of the LP cepstrum by a factor $(1 + \alpha^n - \beta^n)$.

Study of the postfilters is beyond the scope of this thesis.

2.4 Further Coefficients

The parameters extracted from the parameterizations described above are said to be static, as they are computed from the information contained on the present sample only. Usually an energy term is added as a part of the feature vector, as it has been shown to improve performance.

By including time derivatives in the feature vector, another improvement in performance can be achieved. Time derivatives (delta and acceleration coefficients) are computed to keep track on the strong time correlation in the speech signal.

These coefficients are presented below.

2.4.1 Energy

The energy term is computed as the log of the signal energy. For example, assuming $s(n)$ to be a sampled signal, the energy term would be:

$$E = \log\left(\sum_{n=1}^{N} s^2(n)\right) \quad (2.18)$$

2.4.2 Delta Coefficients

Delta coefficients (also known as first order regression coefficients) are computed by using the following regression formula:

$$d_t = \frac{\sum_{\theta=-1}^{1} \theta(\alpha_{t+\theta} - \alpha_{t-\theta})}{2 \sum_{\theta=-1}^{1} \theta^2} \quad (2.19)$$
where $d_t$ is the delta coefficient at time $t$ computed in terms of the corresponding static coefficients $a_{t+\theta}$ and $a_{t-\theta}$. The value of $\theta$ is set to 2 ns.

### 2.4.3 Acceleration Coefficients

Acceleration coefficients (or second order regression coefficients) are computed by using a formula similar to the previous one, but involving the delta coefficients instead of the static coefficients, and keeping the same value of $\theta$.

$$a_t = \frac{\sum_{\theta=1}^{\Theta} \theta (d_{t+\theta} - d_{t-\theta})}{2 \sum_{\theta=1}^{\Theta} \theta^2} \quad (2.20)$$
Chapter 3

Experimental Settings

3.1 The GIVES Environment

GIVES\(^1\) is a software package built for research in automatic speaker verification. It is meant to have a general purpose kernel which can be used both in tests on speech databases and in real-time demonstrators.

A model of the speaker has to be defined. This is done by writing a template file, which defines the structure of the speaker model in a hierarchical way. This template is used as an input to a training tool, which sets data parameters to values estimated from training. Once the speaker model has been trained, it can be used in any operation in GIVES. As a result from this operation, a score value is obtained. The score value is a measure of the similarity between two entities, usually a model and a fragment of speech. It will be used to make a decision [11].

One aim of this thesis work was to build LSP and a pole-zero based parameterization in GIVES. This has been done by programming the necessary modules in C++.

ACW has been chosen as the pole-zero parameterization to be implemented, as it has been shown to perform as well as or better than the LP cepstrum for text-independent speaker identification[7]. Here we will test its performance in speaker verification. Another reason to choose ACW is that there exists a fast algorithm for finding it [12].

3.1.1 Implementation of ACW in GIVES

The way to implement ACW in GIVES is by coding the necessary modules to perform the parameterization. Figure 3.1 represents the way ACW is obtained, and on it one can see all the GIVES modules needed.

The first step is to compute the LPC. These are used to compute the LPC cepstrum (LPCC) and ACW cepstrum. ACW was coded first using

\(^1\)General Identity VEnification System [11]
the operator MATLABO, which makes it possible to use MATLAB code, to perform the three first steps in the ACW branch. Only the ACW and SUBTRACT operators were coded in C++. This was done in order to simplify the programming, but was found to be excessively slow. ACW was then reprogrammed, using the results in [12] and a fast algorithm was found. The obtained results are slightly better than the initials, due possibly to the lower amount of rounding errors.

The code of all these functions can be found in Appendix 1.

---

Figure 3.1: ACW Parameterization Tree
3.1.2 Implementation of LSP in GIVES

LSP has been programmed into GIVES using the code written in another thesis work [13]. It was originally coded in C, and the code has been reused. The result is a new GIVES module, which performs Line Spectral Pair parameterization. See Appendix 1 for details.

![LSP Parameterization Tree](image)

Figure 3.2: LSP Parameterization Tree
3.2 Experimental Settings

In this section, the conditions under which the experiments were run are presented. YOHO is the database selected to run the set of experiments on, and has been filtered for this purpose (see below). In order to have a reference when comparing the results, our aim has been to set up the conditions as close as possible to those used within the CAVE project. These are presented in this section as well.

3.2.1 The YOHO Database

YOHO is a standard database for testing voice verification systems [14]. It is composed by 138 speakers of both genders (106 males and 32 females), most of them from the New York City area. There are 4 enrollment sessions of 24 utterances each and 10 verification sessions of 4 utterances each for every speaker. These utterances are "combination lock" phrases, as the speaker is prompted to say, for example, "Twenty-six, fifty-seven, eighty-two".

The following is a summary of the characteristics of the YOHO database:

- "Combination lock" phrases
- 138 speakers (106 males, 32 females)
- Recordings collected during 3 months
- Approximately 3-day verification intervals
- Real-world office environment
- 4 enrollment sessions per subject
- 24 utterances per enrollment session
- 10 verification sessions per speaker
- 4 utterances per verification session
- 1380 verification sessions in total
- 8 KHz sampling with 3.8 KHz bandwidth
- High-quality telephone handset (Shure XTH-383) used to collect the speech
- Speech not passed through a telephone channel
3.2. EXPERIMENTAL SETTINGS

All the 118 * 40 = 4720 verification utterances were used both as true claims and as random impostor attempts, which gives a total of 9440 tests.

This last characteristic is undesirable, as we aim to test our SV system in mobile telephone applications. Because of this, the YOHO database had to be filtered.

Not all possible verification phrases are available from enrollment in YOHO, as this would give as a result an excessive enrollment time. Anyway, the enrollment data covers the acoustic space of all possible speech that could be prompted later on during the verification sessions.

Filtering of the YOHO database

To filter the YOHO database, a GSM-Bluetooth frequency mask provided by ERICSSON has been used. Following the constraints imposed by this mask, a digital filter has been implemented. Several approaches were considered to this end, taking into account the fact that a format conversion of the data should also be applied (given that GIVES can only read certain kinds of audio files).

![Figure 3.3: GSM-Bluetooth mask and BP filter applied](image)

The solution adopted was the design of a band pass filter in MATLAB, as it is quite easy to do it and it gives also the possibility to process the data in batch mode. The filter is FIR type, and its transfer function plot is depicted in figure 3.3. The filter attenuates at least 60 dB in both rejection bands. We have to say that we are being more restrictive on the higher rejection band than the mask imposes, and so the results will be slightly
pessimistic. No additive noise has been added.

### 3.2.2 The CAVE Project

The CAVE [15] project (CAller VErification in Banking and Telecommunications) was a 2 year project whose technical objectives was to design, implement and assess 2 telephone-based systems which use SV technology. It was supported by the Language Engineering Sector of the Telematics Applications Programme of the European Union, and by the Swiss partners by the Office Fédéral de l’Education et de la Science.

A number of algorithms have been tested within CAVE, on two different databases, namely YOHO and SESP (whose contents can be considered as very representative of real-world telephone speech). The configuration of the tests we have run in this thesis work are as close as possible to that within the CAVE project. A set of 20 subjects, 10 of each gender, are randomly selected from those in the database, and a world model is built from them. These speakers will not be enrolled into the system. The world model is a speaker-independent model built to be used in the verification algorithm, to compute the log likelihood ratio.

The HMM topology used has \( p \) states per phoneme, and \( q \) Gaussian mixtures per state. This last parameter we will play with in this thesis work, together with the number of sessions on enrolling, when setting up the experimental environment.

### 3.2.3 Experimental Conditions

As stated before, we will try to set up the experimental conditions to be as close as possible to those within the CAVE project. The parameters we will play with are the number of sessions used on enrolling and the number of Gaussian mixtures per state in the models. Performance of the system is studied for different number of enrolling sessions for practical reasons: it is simply more convenient to the final user. On the other hand, the reason to look at the number of gaussian mixtures per state as a parameter is that, in mobile applications, memory is a scarce resource, so it is a must to optimize its use.

This way, the experimental conditions can be represented in a coordinate system as follows:

We will only study the 4 points marked in the figure, as they seem to be representative of the effect that the parameters have on the performance of the system.

**Variance and transition probabilities tying**

In the modeling of the clients, variances of the Gaussian mixtures within a state and transition probabilities between states are not updated. This way
we reduce the number of variance parameters to estimate by a factor equal to the number of Gaussian mixtures per state. It has been shown in [17] that the performance of the system does not degrade significantly, and on the other hand, the storage requirements become lower.
Chapter 4

Results

In this chapter, the results of the experiments carried out are presented and analyzed. The first set of experiments is run using the non-filtered version of the YOHO database, and will be used as a reference when measuring the loss in performance due to the effect of the telephone channel. The rest of the experiments are run on the filtered version of the database, and on them we will study the effect of the amount of training data and the number of Gaussian mixtures per state in the model.

Note that we put focus on text-dependent SV. Impostor trials are carried out by speakers of both sexes.

4.1 Reference Experiments

The aim of this set of experiments is to test the different parameterizations in a favorable environment. That is, the database has not been filtered already, so the quality of the recordings yields good results. This way we will get a feeling of the comparative performance of LPCC, LSP and ACW, an also of the loss in performance when speech is affected by a telephone channel.

A world model is trained by randomly selecting 20 speakers out of the whole database, 10 of each gender. These speakers will not be a part of the system, i.e. they will neither be enrolled on it nor used as possible impostors in verification sessions. The HMM models are Left-to Right type (HMM-LR), and the number of Gaussian mixtures per state is 8. The speakers were enrolled in the system using the 24 recordings available from each of the 4 sessions, so the total number of enrolment utterances per speaker used on enrolling is 96. Testing is done on the 4720 available recorded utterances for verification, and every test is performed on one utterance only. As every verification utterance is used both as true claims and as impostor attempts, the total number of tests is 9440.
4.1.1 Lineal Prediction Cepstral Coefficients

To test LPCC, 12 cepstral coefficients are computed from the 16th order LP model of the speech signal. An energy term is also part of the feature vector. Delta and acceleration coefficients have been shown to improve performance, so they are also computed. This way, the feature vector is composed by \((12 + 1)3 = 39\) elements. Cepstral mean subtraction is used, because it simulates the processing that is commonly used in speech through the telephone network. This might degrade the performance.

Dynamic and static scores have been computed. Dynamic scores yield optimistic results, as the threshold is speaker-dependent and is computed \textit{a posteriori}, once the scores are known. EER assumes that the threshold score value is computed so that the False Rejection Rate is equal to the False Acceptance Rate, taking into account male and female impostors with equal contribution. All the EERs are averaged for each sex, and then averaged themselves. This is done as in the CAVE project, see [15] for further details. The result obtained is \(EER = 0.227\), pretty close to the value obtained in CAVE (0.21).

But we are more interested in the static score. It is computed using speaker-independent threshold, and based only on same-sex impostor attempts. The result is \(EER = 0.94\), higher than the EER computed from the dynamic score.

A Detection Error Tradeoff (DET) curve has been represented. The reason to plot a DET curve instead of the traditional ROC (Receiver Operating Characteristics) curve is that we want to represent the performance of a system where tradeoffs of two error types are involved [16]. See figure 4.1.
4.1. REFERENCE EXPERIMENTS

Figure 4.1: DET curve for LPCC over filtered and unfiltered YOHO

To get a feeling of the loss in performance due to the filtering of the database, the same experiment has been run over the filtered version of YOHO. The results in cross sex tests (males claiming female id or vice versa) show a higher degradation in performance when male speakers act as true clients. The static EER obtained is 1.19. See figure 4.1.

4.1.2 Line Spectral Pair

For LSP, 10 coefficients have been computed from a 16th order LPC model of the signal. Energy, delta and acceleration coefficients are a part of the feature vector as well. That is, the feature vector has 33 elements. The static EER is 6.32.
Figure 4.2: DET curve for LSP over unfiltered YOHO

4.1.3 Adaptive Component Weighting Cepstral Coefficients

Again 12 cepstral coefficients are computed from a 16th order LPC model of the signal. Energy, delta and acceleration coefficients are a part of the feature vector as well. That is, the feature vector has 39 coefficients.

Dynamic scores are computed, obtaining $EER = 1.535$. The static score in this case is 2.67. Figure 4.3 shows the DET plot of the static scores for ACW. Compared to LPC, the results are considerably worse.

The effect of using delta and acceleration coefficients yields significant improvement, as can be seen if we compare the previous results with the ones obtained using only ACW and energy as coefficients: $EER_{dyn} = 2.043$, $EER_{sta} = 2.52$. See 4.3.
4.2. RESULTS

![Effect of delta and acceleration coefficients](image)

**Figure 4.3:** DET curve for ACW+e / ACW+e+d+a

### 4.2 Results

The experiments carried out to test the performance of LPCC, LSP and ACW have the same setup as the reference experiments, except from the fact that the database has been filtered and the number of utterances per enrollment session is 6 instead of 24 (this is more realistic). The experiments on LPCC have also been run using 24 utterances per enrollment session, to get an idea of the effect of the decrease in the amount of training data.

#### 4.2.1 Linear Prediction Cepstral Coefficients

Figure 4.4 shows the results for LPCC in the 4 situations studied.

**Results using 2 enrolment sessions/user 2 Gaussian mixtures/state**

The EER computed from the static score is equal to 2.32 %.

**Results using 2 enrolment sessions/user 8 Gaussian mixtures/state**

In this case, the EER computed from the static score is 1.82 %. An increase on the number of gaussian mixtures per state yields an improved performance of the system, compared to the previous situation. See 4.4.
CHAPTER 4. RESULTS

Results using 4 enrolment sessions/user 2 Gaussian mixtures/state

Here the number of sessions used on enroling the clients in the system is increased from 2 to 4, while the number of gaussian mixtures is kept low and equal to 2. The improvement in performance is higher than in the previous situation.

The EER computed from the static score is 1.67 %.

Results using 4 enrolment sessions/user 8 Gaussian mixtures/state

As expected, the best result is obtained when a lot of training data is used and the number of gaussian mixtures per state is high. The EER computed from the static score is 0.99 %.

![DET plots for LPC parameterization](image)

Figure 4.4: Results for LPCC
4.2. RESULTS

Effect of the amount of training data/session

The same experiments have been repeated, but using the 24 available recordings from each enrolling session. The results are presented in the following table, as well as the values obtained using only 6 utterances/session.

<table>
<thead>
<tr>
<th></th>
<th>6 utt/sess.</th>
<th>24 utt/sess.</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 sess, 2 gauss</td>
<td>2.32 %</td>
<td>1.67 %</td>
</tr>
<tr>
<td>2 sess, 8 gauss</td>
<td>1.82 %</td>
<td>1.02 %</td>
</tr>
<tr>
<td>4 sess, 2 gauss</td>
<td>1.67 %</td>
<td>1.67 %</td>
</tr>
<tr>
<td>4 sess, 8 gauss</td>
<td>0.99 %</td>
<td>0.89 %</td>
</tr>
</tbody>
</table>

Table 4.1: LPCC - Static EER

Looking at the results for 24 utterances/session, we note that there is no improvement when the number of enrolment sessions increases from 2 to 4. The reason could be that using 24 utterances from 2 sessions the system has enough data to train the system properly.

Anyway the result is the same as using only 6 utterances per session and the same number of sessions. The conclusion is that there is no use in increasing the number of utterances per enrolment session if the number of enrolment sessions is at least 4. It would be interesting to study the performance of the system when only 3 and 5 sessions are used on enrolling, but it is beyond the scope of this thesis.

4.2.2 Line Spectral Pair

The results are depicted in figure 4.5

Results using 2 enrolment sessions/user 2 Gaussian mixtures/state

In this situation, \( EER = 12.42 \). The DET plot is depicted in figure 4.5

Results using 2 enrolment sessions/user 8 Gaussian mixtures/state

By increasing the number of Gaussian mixtures per state, the performance of the system is slightly better, but is still bad. The EER in this case is 11.99.

Results using 4 enrolment sessions/user 2 Gaussian mixtures/state

When the number of Gaussian mixtures is kept low but the amount of training data increases, the improvement is higher than in the previous case. The new EER is 9.00.
Results using 4 enrolment sessions/user 8 Gaussian mixtures/state

When both the amount of training data and the number of Gaussian mixtures per state are increased, the EER decreases to 8.02.

4.2.3 Adaptive Component Weighting

The results for ACW are shown in figure 4.6

Results using 2 enrolment sessions/user 2 Gaussian mixtures/state

The EER obtained in this case is 6.42.

Results using 2 enrolment sessions/user 8 Gaussian mixtures/state

By increasing the number of Gaussian mixtures, an improvement in performance is achieved. The EER is 5.96.

Results using 4 enrolment sessions/user 2 Gaussian mixtures/state

When the amount of training data increases, the EER decreases a lot, even though the number of Gaussian mixtures is kept low. The result is very
close to the obtained while testing on the unfiltered version of YOHO, using 24 utterances per enrollment session (instead of 6) and 8 Gaussian mixtures. The new EER is 2.80.

**Results using 4 enrolment sessions/user 8 Gaussian mixtures/state**

Finally, by increasing the number of Gaussian mixtures per state with respect to the previous case, the EER becomes slightly better, and becomes equal to the value obtained while testing on the unfiltered database \(EER = 2.67\).

![DET plots for ACW parameterization](image)

**Figure 4.6: Results for ACW**
4.3 Comparison Between the Three Methods

The following table represents the EER for each of the 4 situations studied before.

<table>
<thead>
<tr>
<th></th>
<th>2 sess/2 G</th>
<th>2 sess/4 G</th>
<th>4 sess/2 G</th>
<th>4 sess/8 G</th>
<th>Clean speech</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACW</td>
<td>6.42</td>
<td>5.96</td>
<td>2.8</td>
<td>2.67</td>
<td>2.67</td>
</tr>
<tr>
<td>LSP</td>
<td>12.42</td>
<td>11.99</td>
<td>9</td>
<td>8.02</td>
<td>6.32</td>
</tr>
<tr>
<td>LPCC</td>
<td>2.32</td>
<td>1.82</td>
<td>1.67</td>
<td>0.99</td>
<td>0.94</td>
</tr>
</tbody>
</table>

Table 4.2: Results
Chapter 5

Discussion

In this chapter, the results obtained are commented. Figure 5.1 depicts the DET plot for the three parameterizations studied, for the best situation studied (4 sessions/user and 8 gaussians/state).

![DET plots for LPC, ACW and LSP (Best case, 4s–8g)](image)

Figure 5.1: Comparative results

5.1 Linear Predictive Cepstral Coefficients

LPCC have been shown to yield the best results in all our experiments.
### 5.2 Line Spectral Pair

LSP has not yielded good results in Speaker Verification. A reason for this might be a bad selection of the parameters while setting up the experiment. An increase in the number of coefficients could give as a result a decrease in the EER, but of course would increase the memory requirements and the computational load. Note that the difference between testing on clean speech and testing on filtered data but in the most favourable situation is high. This does not occur in LPCC or ACW. This might be because LSP requires for more complex models and/or more training data to perform well. As one of the aims of this thesis is to test LSP on mobile applications, we could say that it is not appropriate for this purpose.

<table>
<thead>
<tr>
<th></th>
<th>2 sess/2 G</th>
<th>2 sess/4 G</th>
<th>4 sess/2 G</th>
<th>4 sess/8 G</th>
<th>Clean speech</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSP</td>
<td>12,42</td>
<td>11,99</td>
<td>9</td>
<td>8,02</td>
<td>6,32</td>
</tr>
</tbody>
</table>

Table 5.2: LSP Results

### 5.3 Adaptive Component Weighting

ACW has given the following results:
5.3. ADAPTIVE COMPONENT WEIGHTING

<table>
<thead>
<tr>
<th></th>
<th>2 sess/2 G</th>
<th>2 sess/4 G</th>
<th>4 sess/2 G</th>
<th>4 sess/8 G</th>
<th>Clean speech</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACW</td>
<td>6.42</td>
<td>5.96</td>
<td>2.8</td>
<td>2.67</td>
<td>2.67</td>
</tr>
</tbody>
</table>

Table 5.3: ACW Results

Compared to LPCC, they are considerably worse. In the worst situation, the loss in performance is comparable to that on LPCC.

When the models become more complex (i.e. the number of Gaussian mixtures is higher), the improvement in performance is not very significant. This leads us to think that the number of Gaussians/state is not the main source of degradation in performance in this case.

Looking at the EER when enrolment is done using 4 sessions and 2 Gaussian mixtures/state, the improvement is high, as it gets pretty close to the results obtained when testing on clean speech (even though on clean speech, the number of utterances/session is four times higher and the number of Gaussians as well). Compared to LPCC, ACW seems to be less affected by a decrease in the quality of the recording even if the models used are simpler. ACW is more robust than LPCC.

The combination of the previous cases leads to an EER equal to the one obtained while testing on clean speech, which is also a proof of the robustness of the ACW parameterization.
Chapter 6

Conclusions

6.1 Conclusions

In this thesis work, several speech parameterizations have been tested on Speaker Verification. A first set of tests was run on clean speech, and the aim of this set was to get a feeling on the effect of the quality of the recordings and the amount of training data in performance. LSP has been found to be more sensitive to these.

The next set of experiments was run to test LPCC, LSP and ACW and study the effect of the amount of training data and the complexity of the models in performance. LPCC has been shown to perform the best, even though it is less robust to the restrictions imposed over the amount of training data and the complexity of the models used. The results obtained for LSP make us think that LSP is not a good parameterization for Speaker Verification. ACW has been found to be more robust than the rest, as it performs almost as well as when using clean speech, even if the models used are simpler.

6.2 Further Research

Several directions could be taken to keep on investigating on this subject and decrease the EER:

- One of them could be focused on other pole-zero based speech parameterizations, see [7]. ACW is more robust than LPCC, but the EER is higher in all the situations studied. Maybe an alternative parameterization would keep this robustness and provide with less EER at the same time.

- LSP has been shown to yield good results on Speech Recognition, it might be good to test LSP under different conditions.
• Text-independent SV using pole-zero based speech parameterizations could also be a matter of study.

• Some of the dynamic coefficients could be removed, decreasing this way the computational load and the memory requirements.
Chapter 7

Memory Requirement

In this chapter we aim to give an idea of the memory requirement of our speaker verification system. This is a very important feature, as in mobile applications memory is a scarce resource.

7.1 Memory Requirements

In this section, the amount of memory required is computed. Assume the following scenario:

- ACW parameterization
- 4 sessions on enroling
- 6 utterances per enrolment session
- 2 Gaussian mixtures per state
- 4 seconds long utterances on average
- 6 states/HMM on average
- 6 phonemes/utterance on average

Let $L_{ACW} = 39$ be the length of the ACW feature vector. Each state is characterized by a variance vector and a mean vector of $39 \times 2 = 78$ coefficients each (each state has 2 Gaussian mixtures). Due to the Left-to-right topology of the HMMs, each state has 2 possible transition outputs, with 2 corresponding probabilities. These probabilities will be computed while enroling as well. Then to each state corresponds a set of

\[ L_S 78 + 78 + 2 = 158 \quad (7.1) \]

parameters. Assume each component is coded by 8 bits.
CHAPTER 7. MEMORY REQUIREMENT

The average number of states per second of speech can be computed as follows: each phoneme has 6 states (on average), so the number of states/utterance is 36. As each utterance is 4 seconds long, the number of states per second is \( N_h \frac{36}{4} = 9 \) states/second.

Let each HMM state contain a feature vector with a memory requirement

\[
M_v = 8L_S
\]  

(7.2)

Then the memory estimation per phoneme will approximately be equal to

\[
M_w = M_v N_h = 11376 \text{bits/second} = 1422 \text{bytes/second}
\]  

(7.3)

The memory requirement for our system based on ACW is 1422 bytes/second. But for some parameters the 8-bits word length might not be enough, so some extra memory is needed to store them in a 16-bit word. Assuming \( k \) of these parameters, the memory requirement becomes

\[
M_w = 1422 \text{bytes/second} + k \text{bytes/second}
\]  

(7.4)
Chapter 8

Appendix

This appendix contains the code generated in this Master Thesis. It is coded in C++, and is written to be used as a part the GIVES software package.

oplpcacwfast.cc

///////////////////////////////////////////////////////////////////////////
//
// File: oplpcacwfast.cc
//
// Computes the subtractive cepstral components necessary to perform
// Adaptive Component Weighting (ACW), from a pole-zero model of the
// signal. It is based on the fast algorithm presented in [3]
//
///////////////////////////////////////////////////////////////////////////

#include "PD/plpacwfast.hh"

///////////////////////////////////////////////////////////////////////////
//
// Class: COpLPC2ACWFAST
// Inherits: COpVectorInput
//
// Date: September 2001
// Creator: Francisco Javier García Torcelly
// KTH / Speech Communication and Music Acoustics
//
// # Algorithm:
//        ---------
// Fast recursion formula to compute ACW from LP filter coefficients.
//        ---------
// ### References:
//        ---------
// [1] J. Mammine et al., Speaker Identification Based on Robust
// Cepstral Features obtained from Pole-zero Transfer Functions.
// Algorithm for Finding the Adaptive Component Weighted Cepstrum
// for Speaker Recognition, IEEE Trans. on Speech and Audio Proc.
// ### Description:
//        ---------
// The input is one vector containing the residues and
// poles of a partial fraction expansion of the all-pole
// transfer function that results from LP analysis.
//        ---------
// ### Revision history:
// 010902
// File created.
// /
//.SetActiveWindow

// Define to the name of the class
#define STRP_CLASSNAME    COpLPC2ACWFAST
#define STRP_CLASSNAME_STR "COpLPC2ACWFAST"

// Define to the name of the nearest base class in the
// inheritance chain.
#define STRP_BASECLASS    COpVectorInput

// Default operator parameters
#define DEFAULT_ORDER (12) 
#define DEFAULT_NUMCEPS (12) 
#define DEFAULT_CEPLIFT (0) 

// Extreme operator parameters 
#define MIN_ORDER (2) 
#define MAX_ORDER (64) 
#define MIN_NUMCEPS (2) 
#define MAX_NUMCEPS (64) 
#define MIN_CEPLIFT (0) 
#define MAX_CEPLIFT (30) 

// Construction, setup and destruction: 

STROP_CLASSNAME: :STROP_CLASSNAME(TraceLevel trace) 
     : STROP_BASECLASS(STROP_LPC2ACWFAST, trace) 
{
    SetClassName(STROP_CLASSNAME_STR); 
    ConstructorCall(); 

    myName = "ACW-Cepstrum-Fast";

    // Symbols for loading the operator 
    const int NumSymbols = 3; 
    static char* SymbolTexts[NumSymbols] = {
        // symdoc begin (type of follower, base class, description, in 
        // template, related to, values, default)
        "ORDER",// ;int; ; LPC cepstrum order; ; ; 2-64; 12
        "NUMCEPS",// ;int; ; number of cepstrum coefficients on output; ; ; 2-64; 12
        "CEPLIFT"  // ;int; ; cepstral liftering coefficient; opt; ; >=0, 0=no liftering; 0
    // symdoc end
    
    static int SymbolValues[NumSymbols] = {
        SYM_ORDER,
        SYM_NUMCEPS,
        SYM_CEPLIFT
    
    symbols->AddSet(NumSymbols, SymbolTexts, SymbolValues);
// Default parameter values
order = DEFAULT_ORDER;
numCeps = DEFAULT_NUMCEPS;
cepLift = DEFAULT_CEPLIFT;

// Default null values for arrays

STROP_CLASSNAME::STROP_CLASSNAME()
{
    DestructorCall();
}

// ================================================================================
// Retrieve, store and print info:
// ================================================================================

boolean STROP_CLASSNAME::InterpretSymbol(pCInputParser parser, boolean* wasEnd, pCSymbolAndValue sym)
{
    // First, see if the symbol is defined in the inherited class
    if (STROP_BASECLASS::InterpretSymbol(parser, wasEnd, sym)) {
        return true;
    }

    // It wasn’t, try my own symbols
    *wasEnd = false; // Default
    if (sym) {
        switch(sym->value) {
        case SYM_ORDER:
            order = parser->GetInt(MIN_ORDER, MAX_ORDER);
            break;
        case SYM_NUMCEPS:
            numCeps = parser->GetInt(MIN_NUMCEPS, MAX_NUMCEPS);
            break;
        case SYM_CEPLIFT:
            cepLift = parser->GetInt(MIN_CEPLIFT, MAX_CEPLIFT);
            break;
        default:
            // The symbol was not found on this level
            return false;
        }
    }
return true;
}

// There was not even a symbol to interpret...
return false;

}  

tSLStatus STROP_CLASSNAME::LocalSave(pCOutputParser parser)
{
  // First, store elements in the inherited class
  tSLStatus status = STROP_BASECLASS::LocalSave(parser);
  if (status != SLOK) return status;

  // Store my own elements
  SaveParameter(parser, SYM_ORDER, order, DEFAULT_ORDER);
  SaveParameter(parser, SYM_NUMCEPS, numCeps, DEFAULT_NUMCEPS);
  SaveParameter(parser, SYM_CEPLIFT, cepLift, DEFAULT_CEPLIFT);

  return Parse2SLStatus(parser);
}

void STROP_CLASSNAME::PrintParam() {
  printf(" LPC order : %d\n", order);
  printf(" Numceps : %d\n", numCeps);
  printf(" Lifting coeff. : %d\n", cepLift);
}

// ========================================================================
// ComputeSamples:
// ========================================================================

pCStreamSample STROP_CLASSNAME::ComputeSample(tUttId utt, tSampleIndex n, pCStreamSource src, tSampleStatus& st, pCStreamNode)
{
  int i, k = 0;
  // float secondterm = 0;
  float secondterm2 = 0;

  PrintTrace(TRACEDETAIL1, "getting input vector with index %d", n);
  Vector a = InputVector(utt, n, src, st);
  if (!a)
    return 0;
PrintTrace(TRACE_DETAIL1, "vector has %d elements", fvector_size(a));

    // PrintTrace(TRACE_DETAIL1, "commencing first loop to calculate b array");
    float b[order];
    // Compute b[i]
    // Index i in b[i] is OK
    b[0] = 1;
    for (i=1; i < order; i++) {
        b[i] = (order-i)*a[i]/order;
    }

    // What we put into d must be an object of type Vector. Copy c to a Vector cv
    Vector cv = create_fvector(order-1);
    for (i=1;i<order; i++){
        cv[i] = b[i]; // Note: Vector cv indexed from 1
    }

    // Pack the vector into a stream sample class
    PrintTrace(TRACE_DETAIL1, "packing output sample vector");
    pCStreamSample d = new CStreamSample(SAVector);
    d->PutVector(cv);
    return d;

    //======================================================================================
    // Diagnostics:
    //======================================================================================
int STROP_CLASSNAME::CheckInterface(pCStreamSource s){
    int err = 0;
    // Count the number of errors on this inheritance level

    // Check the number of source nodes
    if (s->NumSources() != 1) {
        PrintWarning(GoOn, 17, "Only one source node allowed (%d are assigned)", s->NumSources());
        err++;
    }

    // Return the number of errors on this level plus the number of
    // errors in higher levels.
    return(err + STROP_BASECLASS::CheckInterface(s));
}

int STROP_CLASSNAME::CheckInternal()
{
    int err = 0;

    // Count the number of errors on this inheritance level
    if (numCeps > vecSize) {
        PrintWarning(GoOn, 17, "numCeps(%d) > vecSize(d)", numCeps, vecSize);
        err++;
    }

    // Return the number of errors on this level plus the number of
    // errors in higher levels.
    return(err + STROP_BASECLASS::CheckInternal());
}

int STROP_CLASSNAME::SameParameters(pCStreamOperator p)
{
    // Check on inherited classes first
    if (!STROP_BASECLASS::SameParameters(p)) {
        return false;
    }

    // The class pointed to must be of my own class
    // Need pointer q to access operator parameters.
    STROP_CLASSNAME* q = (STROP_CLASSNAME*)p;

    // Compare the parameters on this level
```cpp
int b;
  b = ((order == q->order) &&
       (numCeps == q->numCeps) &&
       (cepLift == q->cepLift));

  return b;
}

// Operation implementation:
//

int STROP_CLASSNAME::InitOperator()
{
  // Do initialisation in ancestors first
  int err = STROP_BASECLASS::InitOperator();

  // initialisation on this level and add errors to err.

  // Print info
  if (DoTrace(TRACE_DETAIL1)) {
    PrintParam();
  }

  PrintTrace(TRACE_DETAIL1, "Operator %s initialized.", mName);

  return err;
}

// ------------------ end of file ----------------------

oplpcacwfast.hh

//
// File: oplpcacwfast.hh
//
// Description:
```
// Computes Adaptive Component Weighting from LPC, using a fast alg.
//
#include "ASV/opvecin.hh"

class COpLPC2ACWFAST : public COpVectorInput {
private:
    enum SymConsts {
        SYM_NUMCEPS = 0x0010,
        SYM_CEPLIFT = 0x0020,
        SYM_ORDER = 0x0040
    };

protected:

    // Operator parameters, the _variables hold the default values
    int order; // LPC order
    int numCeps; // number of cepstral coefficients
    int cepLift; // cepstral lifting coef

public:

    // Construction and destruction
    COpLPC2ACWFAST(TraceLevel trace=TRACE_NONE);
    virtual ~COpLPC2ACWFAST();

    // Retrieve and print info
    int InitOperator();
    void PrintParam();

    // Data operations
    pCStreamSample ComputeSample(tUttId, tSampleIndex, pCStreamSource,
        tSampleStatus&, pCStreamNode=0);

    // Miscellaneous
    int CheckInterface(pCStreamSource);
    int CheckInternal();
protected:

// Overloading...
bool InterpretSymbol(pCInputParser, bool* wasEnd, pCSymbolAndValue sym);
tSLStatus LocalSave(pCOutputParser);
int SameParameters(pCStreamOperator);

#endif // _OPLPCACWFAST_HH_

// ---------------- end of file -------------------------------

oplsp.cc

///////////////////////////////////////////////////////////////////////
//
// File: oplsp.cc
//
// Description:
// Line Spectral Pair (LSP) from Linear Prediction Coefficients (LPC)
//
///////////////////////////////////////////////////////////////////////

#include "PD/oplsp.hh"

///////////////////////////////////////////////////////////////////////
//
// Class: CLSP
// Inherits: COpVectorInput
//
// Date: August 2001
// Creator: Francisco Javier García Torcelly
// KTH / Speech, Music and Hearing (TMH)
//
///////////////////////////////////////////////////////////////////////

// ### Algorithm:
//      ------
//
// ### References:
//      ------
//
// ### Description:
//  
// Line Spectral Pair (LSP) from Linear Prediction Coefficients (LPC)
//
//  
// ### Revision history:
//
//  
// Define to the name of the class
#define STROP_CLASSNAME CLSP
#define STROP_CLASSNAME_STR "CLSP"

// Define to the name of the nearest base class in the
// inheritance chain.
#define STROP_BASECLASS COpVectorInput

#define MIN_ORDER (2)
#define MAX_ORDER (64)

#define DEFAULT_ORDER (10)

// Construction, setup and destruction:

STROP_CLASSNAME: :STROP_CLASSNAME(TraceLevel trace):
 : STROP_BASECLASS(STROP_LSP, trace)
{
  SetClassName(STROP_CLASSNAME_STR);
  ConstructorCall();

  myName = "LSP";

  // Symbols for loading the operator
  const int NumSymbols = 1;
  static char* SymbolTexts[NumSymbols] = {
    "ORDER" // ;int; ; LPC cepstrum order; ; ; 2-64; 10
  } // symdoc end
};
static int SymbolValues[NumSymbols] = {
    SYM_ORDER
};
symbols->AddSet(NumSymbols, SymbolTexts, SymbolValues);

// Default parameter values
order = DEFAULT_ORDER;
// nc = order/2;
}

STROP_CLASSNAME::~STROP_CLASSNAME()
{
    DestructorCall();
}

// ===========================================================================
//Retrieve, store and print info:
// ===========================================================================

boolean STROP_CLASSNAME::InterpretSymbol(pCInputParser parser, boolean* wasEnd, pCSymbolAndValue sym)
{
    // First, see if the symbol is defined in the inherited class
    if (STROP_BASECLASS::InterpretSymbol(parser, wasEnd, sym)) {
        return true;
    }

    // It wasn’t, try my own symbols
    *wasEnd = false; // Default
    if (sym) {
        switch(sym->value) {
            case SYM_ORDER:
                order = parser->GetInt(MIN_ORDER,MAX_ORDER);
                break;

            default:
                // The symbol was not found on this level
                return false;
        }
        return true;
    }

    // There was not even a symbol to interpret...
return false;
}

tSLStatus STROP_CLASSNAME::LocalSave(pCOutputParser parser)
{
  // First, store elements in the inherited class
  tSLStatus status = STROP_BASECLASS::LocalSave(parser);
  if (status != SLOK) return status;

  // Store my own elements
  SaveParameter(parser, SYM_ORDER, order, DEFAULT_ORDER);
  return Parse2SLStatus(parser);
}

void STROP_CLASSNAME::PrintParam()
{
  printf("LPC order : %d\n", order);
}

// ==================================================================================================
//     Function:
//     //
//     // FUNCTION: Chebys
//     //
//     // PURPOSE: Evaluates the Chebyshev polynomial series
//     //
//     // DESCRIPTION:
//     // - The polynomial order is n = m/2 = 5
//     // - The polynomial F(z) (F1(z) or F2(z)) is given by
//     //   F(w) = 2 \exp(-j5w) C(x)
//     // where
//     //   C(x) = T_n(x) + f(1)T_{n-1}(x) + ... +f(n-1)T_1(x) + f(n)/2
//     // and T_m(x) = \cos(mw) is the mth order Chebyshev polynomial
//     // (x=\cos(w))
//     // - The function returns the value of C(x) for the input x.
//     //
//     // ==================================================================================================

float STROP_CLASSNAME::Chebys (float x, float f[], int n)
{
int i;
float cheb;
float b0, b1, b2;

b2 = 1.0;
b1 = 2 * x + f[1];
for (i = 2; i < n; i++)
{
    b0 = 2.0 * x * b1 - b2 + f[i];
    b2 = b1;
    b1 = b0;
}

cheb = x * b1 - b2 + f[i] / 2;
return (cheb);
}

// Computes the difference filters to get the LSPs from the LP coefficients
int ComputeSample(pCStreamSample STROP_CLASSNAME::ComputeSample(tUttId utt, tSampleIndex n, pCStreamSource src, tSampleStatus& st, pCStreamNode)
//assert(1pc);

// Vector outv = create_fvector(order);

PrintTrace(TRACE_DETAIL1, "Getting input vector with index %d", n);
Vector a = InputVector(utt, n, src, st);
if(!a)
    return 0;
PrintTrace(TRACE_DETAIL1, "Vector has %d elements",
             fvector_size(a));

int nc = order/2;

float lsp[order];

float grid[61] = {1.000000, 0.998630, 0.994522, 0.987688, 0.978148,
                  0.965926, 0.951057, 0.933580, 0.913545, 0.891007, 0.866025, 0.838671,
                  0.809017, 0.777146, 0.743145, 0.707107, 0.669131, 0.629320, 0.587785,
                  0.546639, 0.500000, 0.453990, 0.406737, 0.358368, 0.309017, 0.258819,
                  0.207912, 0.156434, 0.104528, 0.052336, 0.000000, -0.052336, -0.104529,
                  -0.156435, -0.207912, -0.258819, -0.309017, -0.358368, -0.406737,
                  -0.453991, -0.500000, -0.546639, -0.587785, -0.629320, 0.669131,
                  -0.707107, -0.743145, -0.777146, -0.809017, -0.838671, -0.866025,
                  -0.891007, -0.913545, -0.933580, -0.951057, -0.965926, -0.978148,
                  -0.987688, -0.994522, -0.998630, -1.000000};

int m=order;
int i,j,nf,ip;
float xlow,xhigh,ylow,yhigh,prod,xmid,ymid,xint;
float x,y,sign;
float *coef;
float f1[nc+1], f2[nc+1];

//------------------------------*
// find the sum and diff. pol. F1(z) and F2(z) *
//------------------------------*

f1[0] = 1.0;
f2[0] = 1.0;

for (i = 0; i < nc; i++)
{
    f1[i+1]=a[i+1]+a[m-i]-f1[i];
f2[i+1]=a[i+1]-a[m-i]+f2[i];
}

// find the LSPs using the Chebychev pol. evaluation

nf = 0;
ip = 0;

coef = f1;

xlow = grid[0];
ylow = Chebps (xlow, coef, nc);

j = 0;
while ( (nf < m) && (j < 60) )
{
j++;
    xhigh = xlow;
    yhigh = ylow;
    xlow = grid[j];
    ylow = Chebps (xlow, coef, nc);

    prod=ylow*yhigh;
    if (prod <= 0)
    {
        // divide 4 times the interval
        for (i = 0; i < 4; i++)
        {
            xmid = (xlow + xhigh)/2;
            ymid = Chebps (xmid, coef, nc);

            if (ylow*ymid <= 0)
            {
                yhigh = ymid;
                xhigh = xmid;
            }
            else
            {
                ylow = ymid;
                xlow = xmid;
            }
        }
    }
}
x = xhigh-xlow;
y = yhigh-ylow;

if (y == 0)
{
    xint = xlow;
}
else
{
    sign = y;
y = (xhigh-xlow)/(yhigh-ylow);

    if (sign < 0)
    {
        y = -y;
    }

    xint = xlow - ylow*y;
}

lsp[nf] = xint;
xlow = xint;
nf++;

if (ip == 0)
{
    ip = 1;
    coef = f2;
}
else
{
    ip = 0;
    coef = f1;
}
ylow = Chebys (xlow, coef, nc);
}
)
}

// Save results in outv
Vector outv = create_fvector(order);

for (i=1;i<=order; i++){
   outv[i] = lsp[i-1]; // Note: Vector outv indexed from 1
}

// Pack the vector into a stream sample class
PrintTrace(TRACE_DETAIL1, "Packing output sample vector");
pCStreamSample d = new CStreamSample(SAVector);
d->PutVector(outv);
return d;

}

// ==============
// Diagnostics:
// ==============

int STROP_CLASSNAME::CheckInterface(pCStreamSource s)
{
   int err = 0;
   // Count the number of errors on this inheritance level

   if (s->NumSources() != 1) {
      PrintError(GoOn, 17, "Only one source node allowed (%d are
assigned).", s->NumSources());
      err++;
   }

   // Return the number of errors on this level plus the number of
   // errors in higher levels.
   return(err + STROP_BASECLASS::CheckInterface(s));
}
int STROP_CLASSNAME::CheckInternal()
{
    int err = 0;
    // Count the number of errors on this inheritance level

    if (order > vecSize) {
        PrintError(GoOn, 17, "numCeps(%d) > vecSize(q)", order, vecSize);
        err++;
    }

    // Return the number of errors on this level plus the number of
    // errors in higher levels.
    return(err + STROP_BASECLASS::CheckInternal());
}

int STROP_CLASSNAME::SameParameters(pCStreamOperator p)
{
    // Check on inherited classes first
    if (!STROP_BASECLASS::SameParameters(p)) {
        return false;
    }

    // The class pointed to must be of my own class
    // Need pointer q to access operator parameters.
    STROP_CLASSNAME* q = (STROP_CLASSNAME*)p;

    // Compare the parameters on this level
    int b;
    b = (order == q->order);

    return b;
}

// =======================================================================
// Operation Implementation
// =======================================================================

int STROP_CLASSNAME::InitOperator()
{
    // Do initialisation in ancestors first
    int err = STROP_BASECLASS::InitOperator();
// initialisation on this level and add errors to err.

// Print info
if (DoTrace(TRACE_DETAIL1)) {
    PrintParam();
}

PrintTrace(TRACE_DETAIL1, "Operator %s initialized.", myName);
return err;
}

//---------------End of file---------------------------

oplsp.hh

#include "ASV/opvecin.hh"

class CLSP : public C0pVectorInput {

private:

    enum SymConsts {
        SYM_ORDER = 0x0040
    };

public:
// Construction and destruction
CLSP(TraceLevel trace=TRACE_NONE);
virtual ~CLSP();

// Retrieve and print info
int InitOperator();
void PrintParam();

// Miscellaneous
int CheckInterface(pCStreamSource);
int CheckInternal();

pCStreamSample ComputeSample(tUttId, tSampleIndex, pCStreamSource,
tSampleStatus&, pStreamNode=0);
float Chebys(float x, float f[], int n);

protected:

int order;

// Overloading...
boolean InterpretSymbol(pCInputParser, boolean* wasEnd,
pSymbolAndValue sym);
tSLLstatus LocalSave(pCOutputParser);
int SameParameters(pCStreamOperator);

};
#endif // _OPLSP_HH_

//-------------------------------End of file----------------------------------

oplpcacw.cc

ừaรูปการนี้/ฟังก์ชันการนำกลับไปยังโตร์


#include "PD/oplpacw.hh"

/****************************************************************************
// Class: COpLPC2ACW
// Inherits:  COpVectorInput
// Date: June 2001
// Creator: Francisco Javier García Torcelly
// KTH / Speech Communication and Music Acoustics
/****************************************************************************

### Algorithm:
--------

Recursion formula to compute ACW from LP filter coefficients.

### References:
--------

[1] J. Mammane et.al., Speaker Identification Based on Robust
   Cepstral Features obtained from Pole-zero Transfer Functions.


### Description:
--------

The input is one vector containing the residues and
poles of a partial fraction expansion of the all-pole
transfer function that results from LP analysis.

### Revision history:
--------

010626
File created.
//
// Define to the name of the class
#define STROP_CLASSNAME COpLPC2ACW
#define STROP_CLASSNAME_STR "COpLPC2ACW"

// Define to the name of the nearest base class in the
// inheritance chain.
#define STROP_BASECLASS COpVectorInput

// Default operator parameters
#define DEFAULT_ORDER (12)
#define DEFAULT_NUMCEPS (12)
#define DEFAULT_CEPLIFT (0)

// Extreme operator parameters
#define MIN_ORDER (2)
#define MAX_ORDER (64)
#define MIN_NUMCEPS (2)
#define MAX_NUMCEPS (64)
#define MIN_CEPLIFT (0)
#define MAX_CEPLIFT (30)

// Construction, setup and destruction:

STROP_CLASSNAME::STROP_CLASSNAME(TraceLevel trace)
 : STROP_BASECLASS(STROP_LPC2ACW, trace)
{
    SetClassName(STROP_CLASSNAME_STR);
    ConstructorCall();

    mName = "ACW-Cepstrum";

    // Symbols for loading the operator
    const int NumSymbols = 3;
    static char* SymbolTexts[NumSymbols] = {
        "ORDER", // ;int;     ; LPC cepstrum order; ; 2-64; 12
"NUMCEPS", // ;int;  ; number of cepstrum coefficients
on output; ; ; 2-64; 12
"CEPLIFT" // ;int;  ; cepstral liftering coefficient;
opt; ; >=0, 0=no liftering; 0
// symdoc end
};
static int SymbolValues[NumSymbols] = {
   SYM_ORDER,
   SYM_NUMCEPS,
   SYM_CEPLIFT
};
symbols->AddSet(NumSymbols, SymbolTexts, SymbolValues);

// Default parameter values
order   = DEFAULT_ORDER;
numCeps = DEFAULT_NUMCEPS;
cepLift = DEFAULT_CEPLIFT;

// Default null values for arrays

}

STROP_CLASSNAME::~STROP_CLASSNAME()
{
   DestructorCall();
}

// ============
// Retrieve, store and print info:
// ============

boolean STROP_CLASSNAME::InterpretSymbol(pCInputParser parser, boolean*
   wasEnd, pCSymbolAndValue sym)
{
   // First, see if the symbol is defined in the inherited class
   if (STROP_BASECLASS::InterpretSymbol(parser, wasEnd, sym)) {
      return true;
   }

   // It wasn’t, try my own symbols
   *wasEnd = false; // Default
   if (sym) {
      switch(sym->value) {
      case SYM_ORDER:
order = parser->GetInt(MIN_ORDER, MAX_ORDER);
break;
case SYM_NUMCEPS:
    numCeps = parser->GetInt(MIN_NUMCEPS, MAX_NUMCEPS);
    break;
case SYM_CEPLIFT:
    cepLift = parser->GetInt(MIN_CEPLIFT, MAX_CEPLIFT);
    break;

default:
    // The symbol was not found on this level
    return false;
}
return true;

// There was not even a symbol to interpret...
return false;

// First, store elements in the inherited class
tSLStatus status = STROP_BASECLASS::LocalSave(pCOutputParser parser);
if (status != SOK) return status;

// Store my own elements
SaveParameter(parser, SYM_ORDER, order, DEFAULT_ORDER);
SaveParameter(parser, SYM_NUMCEPS, numCeps, DEFAULT_NUMCEPS);
SaveParameter(parser, SYM_CEPLIFT, cepLift, DEFAULT_CEPLIFT);

return Parse2BSLStatus(parser);

void STROP_CLASSNAME::PrintParam() {
    printf(" LPC order : \%d\n", order);
    printf(" Numceps     : \%d\n", numCeps);
    printf(" Lifting coeff. : \%d\n", cepLift);
}

// ComputeSamples:
// =======================================================================
pcStreamSample STROP_CLASSNAME::ComputeSample(tUttId utt, tSampleIndex n,
    pcStreamSource src,
    tSampleStatus& st,
    pcStreamNode)
{
    int i, k = 0;

    PrintTrace(TRACE_DETAIL1, "getting input vector with index %d", n);
    Vector a = InputVector(utt, n, src, st);
    if (!a)
        return 0;

    PrintTrace(TRACE_DETAIL1, "vector has %d elements", fvector_size(a));

    // Extract and normalize the numerator
    float P = a[1];

    PrintTrace(TRACE_DETAIL1, "commencing first loop to calculate b array");
    float b[order];
    for (i = 0; i<order-1; i++) {
        b[i] = a[i+2]/P;
    }
    PrintTrace(TRACE_DETAIL1, "a es igual a %f %f %f %f %f", a[1], a[2],
              a[3], a[4], a[5]);
    PrintTrace(TRACE_DETAIL1, "b es igual a %f %f %f %f %f", b[0], b[1],
              b[2], b[3], b[4]);
    // printf("a es igual a %f %f %f %f %f", a[1], a[2], a[3], a[4], a[5]);
    // printf("b es igual a %f %f %f %f %f", b[0], b[1], b[2], b[3], b[4]);

    // The index variable, i, corresponds to the index of the
    // coefficients in the numerator.

    // Compute the subtractive cepstral coefficients

    float c[numCeps];

    // BUGFIX: the index in vector b is the index in bi coefficients less
    // 1 (as in vector c) -> subtracting 1 from every b index!
    float factor = 0;
    c[0] = -b[0];
// Modif: recursive formula splitted in several steps.
PrintTrace(TRACE_DETAIL1, "commencing nested loop to compute c
array");
for (i=1; i<=order-2; i++) {
    factor = 0;
    for (k=1; k<=i-1; k++){
        factor = factor+(k/(i+1)-1)*b[k-1]*c[i-k];
    }
    c[i] = -b[i]+factor;
}

PrintTrace(TRACE_DETAIL1, "commencing final loop");
for (i=order-1; i<=numCeps-1; i++) {
    factor = 0;
    for (k=1; k<=i-1; k++){
        factor = factor+(k/(i+1)-1)*b[k-1]*c[i-k];
    }
    c[i] = factor;
}

// What we put into d must be an object of type Vector. Copy c to a
// Vector cv
Vector cv = create_fvector(numCeps);
for (i=0; i<numCeps; i++) {
    cv[i+1] = c[i]; // Note: Vector cv indexed from 1
}

// Pack the vector into a stream sample class
PrintTrace(TRACE_DETAIL1, "packing output sample vector");
pCStreamSample d = new CStreamSample(SAVector);
d->PutVector(cv);
return d;
}

//==================================================================
// Diagnostics:
//==================================================================

int STROP_CLASSNAME::CheckInterface(pCStreamSource s)
{
    int err = 0;
}
// Count the number of errors on this inheritance level

// Check the number of source nodes
if (s->NumSources() != 1) {
    PrintError(GoOn, 17, "Only one source node allowed (%d are assigned) ", s->NumSources());
    err++;
}

// Return the number of errors on this level plus the number of
// errors in higher levels.
return(err + STROP_BASECLASS::CheckInterface(s));

int STROP_CLASSNAME::CheckInternal()
{
    int err = 0;

    // Count the number of errors on this inheritance level
    if (numCeps > vecSize) {
        PrintError(GoOn, 17, "numCeps(%d) > vecSize(d) ", numCeps, vecSize);
        err++;
    }

    // Return the number of errors on this level plus the number of
    // errors in higher levels.
    return(err + STROP_BASECLASS::CheckInternal());
}

int STROP_CLASSNAME::SameParameters(pCStreamOperator p)
{
    // Check on inherited classes first
    if (!STROP_BASECLASS::SameParameters(p)) {
        return false;
    }

    // The class pointed to must be of my own class
    // Need pointer q to access operator parameters.
    STROP_CLASSNAME* q = (STROP_CLASSNAME*)p;

    // Compare the parameters on this level
    int b;
    b = ((order == q->order) &&
         (numCeps == q->numCeps) &&
         (order == q->order) &&
         (numCeps == q->numCeps));
(cepLift == q->cepLift )
);

return b;
}

// .nextSibling = child;

// .nextSibling = parent.lastChild;

// .nextSibling = child.nextSibling;

// .nextSibling = parent.firstChild;

// .nextSibling = child.nextSibling;

// .nextSibling = parent.firstChild;

// .nextSibling = child.nextSibling;

// .nextSibling = parent.firstChild;

// .nextSibling = child.nextSibling;

// .nextSibling = parent.firstChild;

// .nextSibling = child.nextSibling;

// .nextSibling = parent.firstChild;

// .nextSibling = child.nextSibling;

// .nextSibling = parent.firstChild;

// .nextSibling = child.nextSibling;

// .nextSibling = parent.firstChild;

// .nextSibling = child.nextSibling;

// .nextSibling = parent.firstChild;

// .nextSibling = child.nextSibling;

// .nextSibling = parent.firstChild;

// .nextSibling = child.nextSibling;

// .nextSibling = parent.firstChild;

// .nextSibling = child.nextSibling;

// .nextSibling = parent.firstChild;

// .nextSibling = child.nextSibling;

// .nextSibling = parent.firstChild;

// .nextSibling = child.nextSibling;

// .nextSibling = parent.firstChild;

// .nextSibling = child.nextSibling;

// .nextSibling = parent.firstChild;

// .nextSibling = child.nextSibling;

// .nextSibling = parent.firstChild;

// .nextSibling = child.nextSibling;

// .nextSibling = parent.firstChild;

// .nextSibling = child.nextSibling;

// .nextSibling = parent.firstChild;

// .nextSibling = child.nextSibling;

// .nextSibling = parent.firstChild;

// .nextSibling = child.nextSibling;

// .nextSibling = parent.firstChild;

// .nextSibling = child.nextSibling;

// .nextSibling = parent.firstChild;

// .nextSibling = child.nextSibling;

// .nextSibling = parent.firstChild;

// .nextSibling = child.nextSibling;

// .nextSibling = parent.firstChild;

// .nextSibling = child.nextSibling;

// .nextSibling = parent.firstChild;

// .nextSibling = child.nextSibling;

// .nextSibling = parent.firstChild;

// .nextSibling = child.nextSibling;

// .nextSibling = parent.firstChild;

// .nextSibling = child.nextSibling;

// .nextSibling = parent.firstChild;

// .nextSibling = child.nextSibling;

// .nextSibling = parent.firstChild;

// .nextSibling = child.nextSibling;

// .nextSibling = parent.firstChild;

// .nextSibling = child.nextSibling;

// .nextSibling = parent.firstChild;

// .nextSibling = child.nextSibling;

// .nextSibling = parent.firstChild;

// .nextSibling = child.nextSibling;

// .nextSibling = parent.firstChild;

// .nextSibling = child.nextSibling;

// .nextSibling = parent.firstChild;

// .nextSibling = child.nextSibling;

// .nextSibling = parent.firstChild;

// .nextSibling = child.nextSibling;

// .nextSibling = parent.firstChild;

// .nextSibling = child.nextSibling;

// .nextSibling = parent.firstChild;

// .nextSibling = child.nextSibling;

// .nextSibling = parent.firstChild;

// .nextSibling = child.nextSibling;

// .nextSibling = parent.firstChild;

// .nextSibling = child.nextSibling;

// .nextSibling = parent.firstChild;

// .nextSibling = child.nextSibling;

// .nextSibling = parent.firstChild;

// .nextSibling = child.nextSibling;

// .nextSibling = parent.firstChild;

// .nextSibling = child.nextSibling;

// .nextSibling = parent.firstChild;

// .nextSibling = child.nextSibling;
#ifndef _OPLPCACW_HH_
define _OPLPCACW_HH_

#include "ASV/opvecin.hh"

class COpLPC2ACW : public COpVectorInput {
private:

    enum SymConsts {
        SYM_NUMCEPS = 0x0010,
        SYM_CEPLIFT = 0x0020,
        SYM_ORDER = 0x0040
    };

protected:

    // Operator parameters, the _variables hold the default values
    int order; // LPC order
    int numCeps; // number of cepstral coefficients
    int cepLift; // cepstral liftering coef

public:

    // Construction and destruction
    COpLPC2ACW(TraceLevel trace=TRACE_NONE);
    virtual ~COpLPC2ACW();

    // Retrieve and print info
    int InitOperator();
    void PrintParam();

    // Data operations
    pCStreamSample ComputeSample(tUtId, tSampleIndex, pCStreamSource,
        tSampleStatus&, pCStreamNode=0);

    // Miscellaneous
    int CheckInterface(pCStreamSource);
    int CheckInternal();

protected:
// Overloading...
bool InterpretSymbol(pCInputParser, bool wasEnd, pCSymbolAndValue sym);
t好莱LocalSave(pCOutputParser);
intSameParameters(pCStreamOperator);
}

#endif // _OPLPCACW_HH_

// -------------- end of file -------------------------------------

opssubtract.cc

///////////
// File: opsub...
//
// Class: CSubtractOperator
// Inherits: CStreamOperator
//
// Date: July 2001
// Creator: Francisco Javier García Torcelly
// KTH / TMH
//
// # Algorithm:
//
// # References:
//
// # Description:
//
// Subtracts second vector from the first one, element by element.
//
// # Revision history:
//
// Define to the name of the class
#define STROP_CLASSNAME CSubtractOperator
#define STROP_CLASSNAME_STR "CSubtractOperator"

// Define to the name of the nearest base class in the
// inheritance chain.
#define STROP_BASECLASS CStreamOperator

// Construction, setup and destruction:
//
STROP_CLASSNAME::STROP_CLASSNAME(TraceLevel trace)
: STROP_BASECLASS(STROP_SUBTRACT, trace)
{
    SetClassName(STROP_CLASSNAME_STR);
ConstructorCall();

myName = "Subtract";

// STROP_SYMBOLS3
// Symbols for loading the operator
const int NumSymbols = 1;
static char* SymbolTexts[NumSymbols] = {
    // symdoc begin (type of follower, base class, description, in
    template, related to, values, default)
    "PARAM" // ;object; CParam; a data stream - definition of or
    reference to a speech parameterization; yes; ; ; none
    // symdoc end
};
static int SymbolValues[NumSymbols] = {
    SYM_PARAM
};
symbols->AddSet(NumSymbols, SymbolTexts, SymbolValues);

plist = new ParamList;
assert(plist);
plist->DeleteObjects(true);
}

STROP_CLASSNAME::STROP_CLASSNAME()
{
    DestructorCall();

delete plist;
}

// =============================================================================
// Retrieve, store and print info:
// =============================================================================

boolean STROP_CLASSNAME::LastLoad(pCInputParser)
{
    PrintTrace(TRACE_DETAIL1, "Subtracting operator has %d branches",
    plist->GetSize());

    return true;
}

boolean STROP_CLASSNAME::InterpretSymbol(pCInputParser parser, boolean*
wasEnd, pCSymbolAndValue sym) {
    // First, see if the symbol is defined in the inherited class
    if (STROP_BASECLASS::InterpretSymbol(parser, wasEnd, sym)) {
        return true;
    }

    // It wasn’t, try my own symbols
    *wasEnd = false; // Default
    if (sym) {
        switch(sym->value) {
        // STROP_SYMBOLS4

            case SYM_PARAM:
                pCParam pp;
                pp = new CParam(GetTraceLevel());
                pp->Load(parser);
                pp->SetSymId(sym->symbol);
                plist->PutLast(pp);
                break;

            default:
                // The symbol was not found on this level
                return false;
                }
                return true;
    }

    // There was not even a symbol to interpret...
    return false;
}

tSLStatus STROP_CLASSNAME::LocalSave(pCOutputParser parser) {
    // First, store elements in the inherited class
tSLStatus status = STROP_BASECLASS::LocalSave(parser);
    if (status != SLOK) return status;

    // Save param definitions
    parser->NewLine();
    ListIterator<pCParam> iter(plist);
    for(; iter.AtElem(); ++iter) {

        pCParam d = iter;

if (!d)
  continue;

  // Save param
  d->Save(parser);
}

return Parse2SLStatus(parser);

// ---------------------------------------------------------------------
// Operation:
// ---------------------------------------------------------------------

// Operator that takes two vectors as input and subtracts the 2nd from the 1st.
// Both vectors should have the same length.
//
pCStreamSample STROP_CLASSNAME::ComputeSample(tUttId utt, tSampleIndex n, pCStreamSource s, tSampleStatus& st, pCStreamNode)
{
  assert(s);
  int N = s->NumSources();
  pCStreamSample* ssin = new pCStreamSample[N];
  assert(ssin);
  int j;

  // Check the number of input vectors.
  if (N != 2) {
    PrintError(ExitNow, 17, "The number of input nodes must be 2");
  }

  // Extract first input vector
  pCStreamNode pn0 = s->GetNode(0);
  assert(pn0);
  ssin[0] = pn0->GetSample(utt, n, st);
  switch(st) {
    case SST_OK:
      break;
    case SST_Wait:

case SST_NotAvailable:
delete ssin;
return 0;

default:
    PrintError(GoOn, 17, "Unknown sample status (%d) in Compute Sample", st);
};

pCStreamSample ps0 = ssin[0];
Vector minuendo = ps0->GetVector();
Vector vmin = create_fvector(fvector_size(minuendo));
for(j=1; j<fvector_size(minuendo); j++) {
    vmin[j] = minuendo[j];
};

// Extract second input vector
pCStreamNode pn1 = s->GetNode(1);
assert(pn1);
ssin[1] = pn1->GetSample(utt, n, st);
switch(st) {
    case SST_OK:
        break;
    case SST_Wait:
    case SST_NotAvailable:
        delete ssin;
        return 0;
    default:
        PrintError(GoOn, 17, "Unknown sample status (%d) in ComputeSample", st);
};

pCStreamSample ps1 = ssin[1];
Vector sustraendo = ps1->GetVector();
Vector vsus = create_fvector(fvector_size(sustraendo));
for(j=1; j<fvector_size(sustraendo); j++) {
    vsus[j] = sustraendo[j];
};
// PrintError(GoOn, 17, "Minuendo tiene %d elementos", fvector_size (vmin));
// PrintError(GoOn, 17, "Sustraendo tiene %d elementos", fvector_size (vsus));

// Check if both vectors have the same length
int length_pn0 = 0, length_pn1 = 0;
length_pn0 = fvector_size(minuendo);
length_pn1 = fvector_size(sustraendo);
if (length_pn0 != length_pn1) {
    PrintError(ExitNow, 17, "Both input vectors must be of the same
length");
}

// Create the output vector and fill it
Vector outv = create_fvector(length_pn0);
int i=0;

for (i=1; i <= length_pn0; i++) {
    //outv[i] = ps0[i] - ps1[i];
    outv[i]= minuendo[i] - sustraendo[i];
}

// Cleanup
// delete ssin;

//myShowVector("comb", outv);

// Create a new stream sample and return it
pCStreamSample newss = new CStreamSample(SAVector);
assert(newss);
newss->PutVector(outv);
st = SST_OK;
return newss;
}

long STROP_CLASSNAME::SampPeriod(tUttId u, pCStreamSource s) {
    // Pass on the sample period of the source
    pCStreamNode node = s->GetNode(0);
    return node->SampPeriod(u);
}
//                          
// Diagnostics:          
//                          
int STROP_CLASSNAME::CheckInterface(pCStreamSource s) 
{ 
    int err = 0; 
    // Count the number of errors on this inheritance level 
    // Return the number of errors on this level plus the number of 
    // errors in higher levels. 
    return(err + STROP_BASECLASS::CheckInterface(s)); 
} 

int STROP_CLASSNAME::CheckInternal() 
{ 
    int err = 0; 
    // Count the number of errors on this inheritance level 
    // ... 
    // Return the number of errors on this level plus the number of 
    // errors in higher levels. 
    return(err + STROP_BASECLASS::CheckInternal()); 
} 

int STROP_CLASSNAME::SameParameters(pCStreamOperator p) 
{ 
    // Check on inherited classes first 
    if (!STROP_BASECLASS::SameParameters(p)) { 
        return false; 
    } 
    // The class pointed to must be of my own class 
    // Need pointer q to access operator parameters. 
    //STROP_CLASSNAME* q = (STROP_CLASSNAME*)p; 
    // Compare the parameters on this level 
    int b; 
    // ... (no parameters in this class) 
    b = true; 
    return b; 
}
int STROP_CLASSNAME::InitOperator()
{
    // Do initialisation in ancestors first
    int err = STROP_BASECLASS::InitOperator();

    // Now, do initialisation on this level and add errors to err.
    // ...

    return err;
}

bool STROP_CLASSNAME::IsCombining()
{
    return true;
}

pParamList STROP_CLASSNAME::GetBranches()
{
    return plist;
}

// -------------- end of file -------------------------------------------

opssubtract.hh

.isDefined OPSUBTRACT_HH_
#define _OPSUBTRACT_HH_

#include "ASV/trstream.hh"

// Forward declaration
class CSubtractOperator;

// Type definitions
typedef CSubtractOperator* pCSubtractOperator;

class CSubtractOperator : public CStreamOperator {
private:

    // Constants for loading an operator from a file
    //
    // STROP_SYMBOLS1
    enum SymConsts {
        SYM_PARAM = 0x0001
    };

protected:

    pParamList plist;

public:

    // Construction and destruction
    CSubtractOperator(TraceLevel trace=TRACE_NONE);
    virtual ~CSubtractOperator();

    // Retrieve and print info
    int InitOperator();

    // Data operations
    pCStreamSample ComputeSample(tUttId, tSampleIndex, pCStreamSource,
        tSampleStatus&, pCStreamNode=0);

    // Miscellaneous
    long SampPeriod(tUttId, pCStreamSource);
    int CheckInterface(pCStreamSource);
    int CheckInternal();

    boolean IsCombining();
pParamList GetBranches();

protected:

    // Overloading...
    boolean LastLoad(pCInputParser);
    boolean InterpretSymbol(pCInputParser, boolean* wasEnd,
        pCSymbolAndValue sym);
    tSLStatus LocalSave(pCOutputParser);
    int SameParameters(pCStreamOperator);
};

#endif /* _OPSUBTRACT_HH_ */

// ----------------- end of file -----------------------------

residuezmodif.m

function RPK = residuez_modif(A)

%RESIDUEZ Z-transform partial-fraction expansion, assuming numerator= 1
%  [RPK] = residuez_modif(A) finds the residues,poles and direct terms
%  of the partial-fraction expansion of B(z)/A(z),
%  B(z)       r(1)       r(n)
%  ----- = --------- +... --------- + k(1) + k(2)z^(-1) ...
%  A(z)  1-p(1)z^(-1)   1-p(n)z^(-1)
%  B and A are the numerator and denominator polynomial coefficients,
%  respectively, in ascending powers of z^(-1). B is assumed to be B=1
%  The result is stored in ONE vector, RPK. The structure of RPK is as
%  follows:
%  [RPK]=[residues poles direct_terms]
%  The number of poles is
%      n = length(A)-1 = length(R) = length(P)
%  The direct term coefficient vector is empty if length(B)<length(A);
%  otherwise,
%      length(K) = length(B)-length(A)+1
%  If P(j) = ... = P(j+m-1) is a pole of multiplicity m, then the
%  expansion includes terms of the form
%      R(j)     R(j+1)     R(j+m-1)
\[
1 - P(j)z^{-1} \quad (1 - P(j)z^{-1})^2 \quad (1 - P(j)z^{-1})^m
\]

See also RESIDUE, RESIDUEZ, PRONY, POLY, ROOTS, SS2TF, TF2SS, TF2Z

Author(s): J. McEllan, 10-24-90

Modified: F. Javier García Torcelly, 07-03-01

References:

mpoles_tol = 0.001; \quad \%--- 0.1 percent
avg_roots = 0; \quad \%--- FALSE, turns off averaging

\%----------------- convert B/A to partial fractions ------------
\%\%\%
\%
\%
\%
\%--------Operate over each column of the input matrix--------
\%
RPK=[];
[nrow,ncol]=size(A);
for k=1:ncol

a=A(:,k);

if( a(1) == 0 )
    error('First coefficient in A vector must be non-zero.')
end
b = [1];
b = b(:).'/a(1); \quad \%--- b() & a() are now rows
a = a(:).'/a(1);
LB = length(b); \quad LA = length(a);
if( LA == 1 )
    r = []; p = []; k = b; \quad \%--- no poles!
    return
end
Nres = LA-1;
p = zeros(Nres,1); r = p;
LK = max([0;LB-Nres]);
k = [];
if( LK > 0 )
    [k,b] = deconv(fliplr(b),fliplr(a));
k = fliplr(k);
b(1:LK) = [];  
b = fliplr(b);  
LB = Nres;
end

p = roots(a);  
N = Nres;  
%--- number of terms in r()

xtra = 1;  
%--- extra pts invoke least-squares approx

imp = zeros(N+xtra,1);  
%--- zero padding to make impulse

imp(1) = 1;

h = filter( b, a, imp );

% polmax = max(abs(p));

polmax = 1.0;

h = h.*filter(1,[1 -1/polmax],imp);  
% if polmax == 1, this is just h = h;

[mults, idx] = mpoles( p, mpoles_tol );

p = p(idx);

S = zeros(N+xtra,N);

for j = 1:Nres

    if( mults(j) > 1 )  
        %--- repeated pole
        S(:,j) = filter( 1, [1 -p(j)/polmax], S(:,j-1) );
    else
        S(:,j) = filter( 1, [1 -p(j)/polmax], imp );
    end
end

jdone = zeros(Nres,1);

bflip = fliplr(b);

for i=1:Nres

    ii = 0;

    if( i==Nres )
        ii = Nres;
    elseif( mults(i+1)==1 )
        ii = i;
    end

    if( ii > 0 )

        if( avg_roots )  
            %--- average repeated roots ??
            jkl = (i-mults(i)+1):i;
            p(jkl) = ones(mults(i),1)*mean( p(jkl) );
        end

        temp = fliplr(poly( p([1:(i-mults(i)), (i+1):Nres)] ));
        r(i) = polyval(bflip,1/p(i)) ./ polyval(temp,1/p(i));
        jdone(i) = 1;
    end

end

%------- now do the repeated poles and the direct terms ------
jjj = find(jdone==1);
jkl = find(jdone~=1);
if( any(jdone) )
    h = h - S(:,jjj)*r(jjj);
end
Nmultipoles = Nres - length(jjj);
if( Nmultipoles > 0 )
    t = S(:,jkl)
h;
    r(jkl) = t(1:Nmultipoles);
end

rpk=[r',p',k];            %--- write the result in ONE vector
RPK=[RPK,rpk'];
end;

%------------------------------------------------end of file--------------------

residuezmodif2.m

function [BA] = residuez_modif2( RPK )

% RESIDUE Z-transform -> partial-fraction expansion
%
% [BA] = RESIDUE(RPK, order) converts the partial-fraction expansion
% back to B/A form.

% BA are is a vector containing the numerator and denominator
% polynomial
% coefficients, as follows: [numerator denominator],in ascending
% powers of
% z^(-1).  RPK is a row vector containing the residues, poles, and
% the direct terms. The number of poles is specified by "order"
% See also RESIDUE, PRONY, POLY, Roots, SS2TF, TF2SS, TF2ZP AND ZP2SS
% Author(s): J. McClellan, 10-24-90
% Modified: F. Javier Garcia Torcelly, 07-03-01
%
% References:

mpoles_tol = 0.001;    %-- 0.1 percent
avg_roots = 0;  %--- FALSE, turns off averaging

%------------- partial fractions ---> B(z)/A(z) -------------
%------ NOW, B <--> R
%------ A <--> P
%------ T <--> K

% % %
%--------Operate over each column of the input matrix--------
BA=[];
order=length(RPK(:,1))/2;
[nrow,ncol]=size(RPK);
for k=1:ncol

  rpk=RPK(:,k);
  res = rpk(1:order)';  %--- res is a column containing the residues
  pol = rpk(order+1:nrow)';  %--- p is column containing the poles
  t=0;  %--- using this function on LPC2ACW (GIVES) will have
  % order(B)<=order(A)
  LR = length(res);  LP = length(pol);  LK = length(t);
  if( LR ~= LP )
    error('Length of R and P vectors must be the same.')
  end
  if( LP == 0 )
    p = []; r = t(:)';  %--- no poles!
    return
  end
  N = LP+LK;  %--- number of terms in b() and a()

  [mults, idx] = mpoles( pol, mpoles_tol, 0 );  % Maintain relative
  % pole ordering
  pol = pol(idx);  %--- re-arrange poles & residues
  res = res(idx);
  p = poly(pol);  % p = p(:);  %--- p is really A(z)

  if( LK > 0 )
    r = conv( p, t );  %--- A(z)K(z)
  else
    r = zeros(1,N);  %--- r is B(z), returned as ROW
  end
  for i=1:LP
    temp = poly( pol([1:(i-mults(i)), (i+1):LP)] )
    r = r + [res(i)*temp zeros(1,N-length(temp))]
  end
r=r(:.');  
p=p(:');  \% polynomials are ROWs  

ba=[r p];  
BA=[BA,ba'];  

end

\%-----------------------------------------end of file---------------------

resnorm.m

function [RPK_MODIF]=resnorm(RPK);

\% function [rpk_modif]=resnorm(rpk);
\%
\% Input = vector containing the residues, poles and
\% direct terms of a partial fraction decomposition,
\% and the number of residues.
\%
\% Output = the same vector, but with the residues set to one.
\%
\%-----------------------------------------

\%----------Operate over each column of the input matrix!
order=length(RPK(:,1))/2;

RPK_MODIF=RPK;
RPK_MODIF(1:order,:)=1;

\%-----------------------------------------end of file---------------------
Bibliography


[3] Proakis, Digital Signal Processing,


