NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

A SPEECH ENHANCEMENT ALGORITHM BASED UPON RESONATOR FILTERBANKS

by

Luc Gagnon, B.A.Sc.

A thesis submitted to the

School of Graduate Studies and Research

in partial fulfillment of the requirements for the degree of

Master of Applied Science

Ottawa-Carleton Institute for Electrical Engineering

Department of Electrical Engineering
Faculty of Engineering
University of Ottawa

May, 1991

Luc Gagnon

ISBN 0-315-70497-7

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

Acknowledgements

To William McGee, my thesis advisor, for helping me to focus on this project and for enabling me to express my creativity and try my ideas;

To my employer, The Department of National Defence, for providing an outstanding research environment;

To my friends and colleagues, Griffith Ince and Tom Rogers, for reviewing this thesis and giving me good advice;

To Carl Bond and Gilles Gonthier, for their friendship and nicely designed software program which has become an invaluable tool to design Analysis-Synthesis Resonator Filterbanks;

And most importantly, to my wife Marie Anne, for giving me so much support and understanding.

I owe you all my deepest thanks.

Table of contents

1. Introduction

1.1. Speech enhancement rationale

Speech is the most efficient interactive way to communicate between humans. However, in many instances speech communication is hampered by degradation caused by imperfections in voice communication channels. A simple example often experienced in our lives is the effect of background acoustical noise when we attempt to understand a distant speaker in a noisy room. Electromagnetically transmitted speech can also be corrupted by different types of degradations such as additive noise or interfering speakers. When the speech signal is degraded, a loss in the intelligibility of the message and a reduction in the perceived quality of the speech signal may occur.

For over 20 years, researchers have explored techniques to improve the quality and intelligibility of speech signals received in the presence of interfering noise. Many terms have been used to describe these algorithms: noise removal, noise stripping, noise reduction and speech enhancement [1]. Ten years ago, noise reduction techniques received considerable attention when digital computers rendered this type of speech processing practical. The recent emergence of powerful and inexpensive Very-Large-Scale Integration (VLSI) single-chip digital signal processors makes it possible to implement real time speech enhancement systems of a very small size and at minimum cost. This technology will foster the incorporation of speech enhancement algorithms in many systems such as cellular telephones, VHF/HF radios, speech recognizers and hearing aids.

Noise reduction systems can be used to increase the intelligibility and quality of speech signals transmitted over telephone lines, by radio waves, and by many other communication channels corrupted by noise. These algorithms are also employed in the audio industry for noise reduction of older recordings and for ambient noise diminution of live-recorded speech and music. Speech enhancement systems have also been used as signal pre-processors before speech recognizers to increase their performance in noisy environments. Because there is a broad variety of applications for speech enhancement and many kinds of degradations which may afflict the speech signal, various approaches are exploited in noise reduction algorithms.

1.2. Thesis overview

Our work focused on the development of a computationally efficient speech enhancement algorithm for broadband noise reduction. Initially, we assessed and compared the performance of many speech enhancement algorithms proposed in the literature. Finally, we devised our own algorithm based on an analysis-synthesis

resonator filterbank (ASRF) configuration. This choice was motivated by our observations that the configuration demonstrates excellent synthesis properties when time-varying spectral amplitude modifications are allowed at the output of the analysis stage and because the resonator filterbanks are computationally efficient when implemented on single-chip digital signal processors. Our noise reduction technique is in the family of algorithms which perform spectral decomposition of the noisy signal to estimate the short-time spectral amplitude of the speech signal. We tested the ASRF speech enhancement algorithm on speech degraded by different kinds of broadband noise and compared the enhanced speech with material produced by other speech enhancement algorithms.

The rest of the thesis is organized as follows: In chapter 2, we explore the fundamental elements of the speech enhancement theory. In chapter 3, we present an overview of different speech enhancement algorithms and we particularly concentrate our survey on methods devised for wideband noise reduction. In chapter 4, we introduce the short-time Fourier transform (STFT) which is a generalization of various methods proposed in the literature for spectral analysis, synthesis and modifications of the speech signal. Then we present our analysis-synthesis resonator filterbank (ASRF) configuration. In the following chapter, we describe the overall architecture of our speech enhancement algorithm. In chapter 6, we report on speech enhancement experiments we carried out with our algorithm and we compare the results with enhanced speech produced by other noise reduction algorithms. Finally, in chapter 7, we conclude and discuss possible extensions to this work.

2. Speech enhancement theory

2.1. Speech enhancement applications

2.1.1. Two-way voice communications

Most voice communication channels cause some type of degradation to the speech signal. Speech enhancement can be used in a variety of voice channels such as the telephone network, public exchange systems and HF/VHF radio communications. Speech intelligibility and quality improvements are the two main objectives targeted by speech enhancement algorithms designed for human listeners. Depending on the application, speech enhancement algorithms may attempt to target one or both of these objectives.

For example, speech enhancement can be implemented in ground mobile communication systems such as cellular telephones to reduce background noise in order to increase the perceived speech quality. In the telephone industry, speech quality as expressed by the mean opinion score (*mos*) is of paramount importance. A slight reduction in speech intelligibility might not be as objectionable as a decline in the user's mean opinion score. However, for other applications such as air traffic control and military radio communications, speech quality is certainly a desired feature but speech intelligibility is imperative.

2.1.1.1. Speech intelligibility

Speech intelligibility is the ability to comprehend the speech message. It is an intricate function of the speech signal. Noise reduces speech intelligibility by affecting weak energy portions of the speech signal. Consequently, unvoiced sounds especially fricatives and plosives suffer more from noise interference than high energy voiced segments of the speech signal [2]. Formal tests have been designed to measure intelligibility. Simply stated, an intelligibility test consists of sending a list of words over a voice communication channel and requires listeners to interpret the words. The number of words correct is declared the intelligibility score of the channel.

In the specific case of the Diagnostic Rhyme Test (DRT) [3], pairs of words are selected to differ only by the initial consonants. Each pair of words disagrees phonemically by only one feature. Thus, the DRT verifies that the system under test can distinguish specific acoustic features. This is useful for designers of speech enhancement systems because the DRT test can pinpoint specific flaws in a system performance. The DRT features are: voicing, nasality, sustention, sibilance, graveness, compactness and vowel-likeness. For example if the system under test is checked for sustention (presence or absence of abrupt discontinuity in energy), the

word "POOH" could be transmitted via the channel or system under test and the listeners would have to choose their answer among these two words: "FOO" and "POOH". Until now, few speech enhancement systems have demonstrated a DRT intelligibility score increase when utilized to process speech degraded by broadband noise.

2.1.1.2. Speech quality (listenability)

Speech quality, sometimes called listenability, is a subjective measure of the overall quality of the speech signal. Therefore it is very difficult to quantify. Even though the speech signal to noise ratio (SNR) is linked to the listenability of speech, noise reduction systems which only boost the SNR may introduce a coloration in the remnant noise which is more perceptually annoying than the original noise. Consequently, typical quality rating tests include several parameters such as naturalness of signal, inconspicuousness of background, intelligibility, pleasantness and overall acceptability.

Systems which increase speech quality without increasing the short-term intelligibility, can nevertheless reduce listener fatigue and actually increase long-term intelligibility. Consequently, the CHABA panel on removal of noise from a speech/noise signal [1] concluded that new testing techniques should be developed to assess not only intelligibility as defined by a closed test such as the DRT, but also many other important perceptual aspects such as speech quality, listener fatigue, and long-term intelligibility.

2.1.2. Speech recognition

Many speech recognition systems exhibit excellent performance in noise-free environments but fail to provide adequate performance when the speech signal is corrupted by noise. Speech enhancement systems can be used to pre-process speech signals prior to the application of a recognition algorithm. Kushner *et al.* [4] have shown that the utilization of a spectral subtraction noise reduction algorithm before a speech recognizer may improve the recognition scores. For speech signals degraded with uniform white noise at a 0 dB signal to noise ratio (SNR), the percentage of words correct climbed from 40 % to 75 % in their recognition test done on isolated vowels.

2.1.3. Speech coding

Boll [5] has demonstrated that speech enhancement can improve the intelligibility and quality of speech processed by linear prediction coding (LPC) voice coders. This type of utilization is similar to the speech recognition application since that in both

cases, the algorithm attempts to isolate the speech information from the noisy speech signal prior to the extraction of speech related features by a coding scheme.

In many operational scenarios voice coders are used in noisy environments, such as in airplane and helicopter cockpits or noisy offices. Speech enhancement algorithms are particularly useful when used in tandem with LPC voice coders.

2.1.4. Audio recordings

With the recent progress in VLSI digital signal processing technologies, speech and music noise reduction workstations can now be put together at minimum cost. In the recording industry, such workstations are used to clean the signal from noisy sources such as old recording media or live recorded speech and music [6]. As mentioned in [1], they can also be used to assist the transcription of noisy speech such as public broadcasts, cockpit voice recorder after a crash or forensic material obtained by a law enforcement agency.

2.2. Types of degradations of speech communication channels

Many types of degradations can afflict the speech signal. Room reverberation, limited channel bandwidth distortions, multiplicative noise, additive noise and even speech elocution pathologies are all forms of speech degradations. In this thesis we will focus our study on degradations caused by additive noise.

2.2.1. Impulsive noise

As its name implies, impulsive noise consists of brief disturbances which result from atmospheric storms in radio communications, transmission errors in digital speech samples communication or electrical discharges. Impulsive noise is also encountered as click and pop sounds produced by imperfections on the surface of vinyl records.

Even though they are very short in duration, pulses perturb perception for a certain time after their occurrences in virtue of the temporal masking properties of the human hearing system.

2.2.2. Interfering speakers

The brain is adequately equipped to recover one voice among other voices. This is the well known cocktail party effect; when several people in a room are chatting, one can concentrate on one of the speakers and hear her (his) speech without being impeded by the other speaker's speech. However, to isolate the desired speech from the interferences, the brain needs binaural data.

Monaural co-talker interference is often encountered in telephone and other type of voice communication channels when electromagnetic cross-talk occurs. Because

the interfering speech occupies the same frequency range as the desired speech signal, the blended speech can be very difficult to separate [7].

2.2.3. Periodic noise

Most of the periodic interferences encountered in speech communication are due to automotive noise [8]. For example in helicopter and airplane cockpits, the engine noise can be acoustically coupled to the speech signal. The noise is colored displaying spectral peaks at multiple frequencies of the engine angular speed. Simple filtering techniques do not suffice to clean the signal when the spectral peaks of the interference vary in time. An other common source of periodic noise is electrical interference such as the sixty (60) Hertz hum which can be rich in harmonics [8].

2.2.4. Wideband noise

The most common form of degradation affecting speech signals is wideband noise. Radio noise, telephone channel noise, thermionic random noise in electronic components (audio amplifiers, radio receivers), quantization noise, and wind noise are all different manifestations of broadband noise.

Wideband noise is often modeled as white noise, which implies that the noise samples are uncorrelated in time and that the power spectral density function is flat [7]. Wideband noise is one of the most difficult problems speech enhancement systems have to face because the noise overlaps the speech signal both in time and frequency.

2.3. Single and multi-input speech enhancement

Sometimes, the noisy speech signal can be monitored by more than one sound transducer. Then, if one microphone can pick up a signal correlated with the noise but uncorrelated with the speech signal, adaptive filtering techniques can be exploited to enhance the speech signal [9]. Since there are more applications for single input speech enhancement, most systems have only access to the noisy speech signal. The rest of the thesis will concentrate on monaural speech enhancement.

2.4. Speech production and perception

Even for the limited case of monaural speech enhancement systems for noise reduction of speech degraded by additive noise, various approaches are suggested in the literature. Most of these systems are based upon our knowledge of speech production and perception mechanisms. The rest of the chapter is a concise

discussion of speech communication fundamentals which are consequential for speech enhancement.

2.4.1. Nature of speech signals

Speech is produced by exciting the vocal tract, which can be modeled to a first approximation by a 17 cm long cylindrical acoustic cavity, by either a pseudo-periodic pulse train of air puffs or noise-like air turbulences. A simplified linear-system model for speech production is drawn in figure 2.1. The vocal tract is modeled as a slowly time-varying filter which parameters are constant for the phones (speech sounds) duration. In this model, the filter includes the effects of mouth radiation and the nasal cavity if the velum is open. The source is either a periodic train of pulses for voiced speech or a noise-like random source for unvoiced speech. The vocal tract filter has a resonant nature because it models an acoustic cavity. These resonant frequencies are known as formants. Because many speech enhancement algorithms process the speech signal in the spectral domain, the nature of speech spectra is relevant to our discussion. Figure 2.1 sketches typical time and frequency speech waveforms.

For the purpose of this basic discussion we can divide speech phones in two categories depending on the source of excitation: voiced and unvoiced. Voiced sounds are characterized by a pseudo-periodic source. Thus, voiced sounds spectra are characterized by harmonic peaks located at multiple frequencies of the fundamental (pitch) frequency. The envelope of the speech signal spectra represents the vocal tract transfer function. If the vocal tract is in a vocalic state, this filter transfer response exhibits narrow peaks called formants. This family of sounds comprises all vowel-like phones.

Unvoiced sounds are generated when the vocal tract is excited by a noise source. This category comprises most consonant sounds. Quite often during the occurrence of unvoiced sounds the vocal tract does not exhibit as many and as narrow resonances. For example, in the case of the English fricative /f/ the poles of the vocal tract are cancelled by zeros of the nasal tract up to 1 kHz. Therefore the energy is not concentrated at resonances but rather widely spread over 1 kHz. Unvoiced sounds have typically less energy and shorter durations than voiced sounds making them harder to detect and estimate in the presence of noise.

2.4.2. Hearing

Hearing is the process which converts acoustical sounds into nerve pulses. Hearing takes place in the auditory periphery. The peripheral auditory system is made up of the outer ear, the middle ear and the inner ear (figure 2.2).

The outer ear is the visible portion of the auditory system. It is simply the opening by which sounds enter the hearing system. The middle ear operates as an impedance matching device which optimizes energy transfer from the air medium to the inner

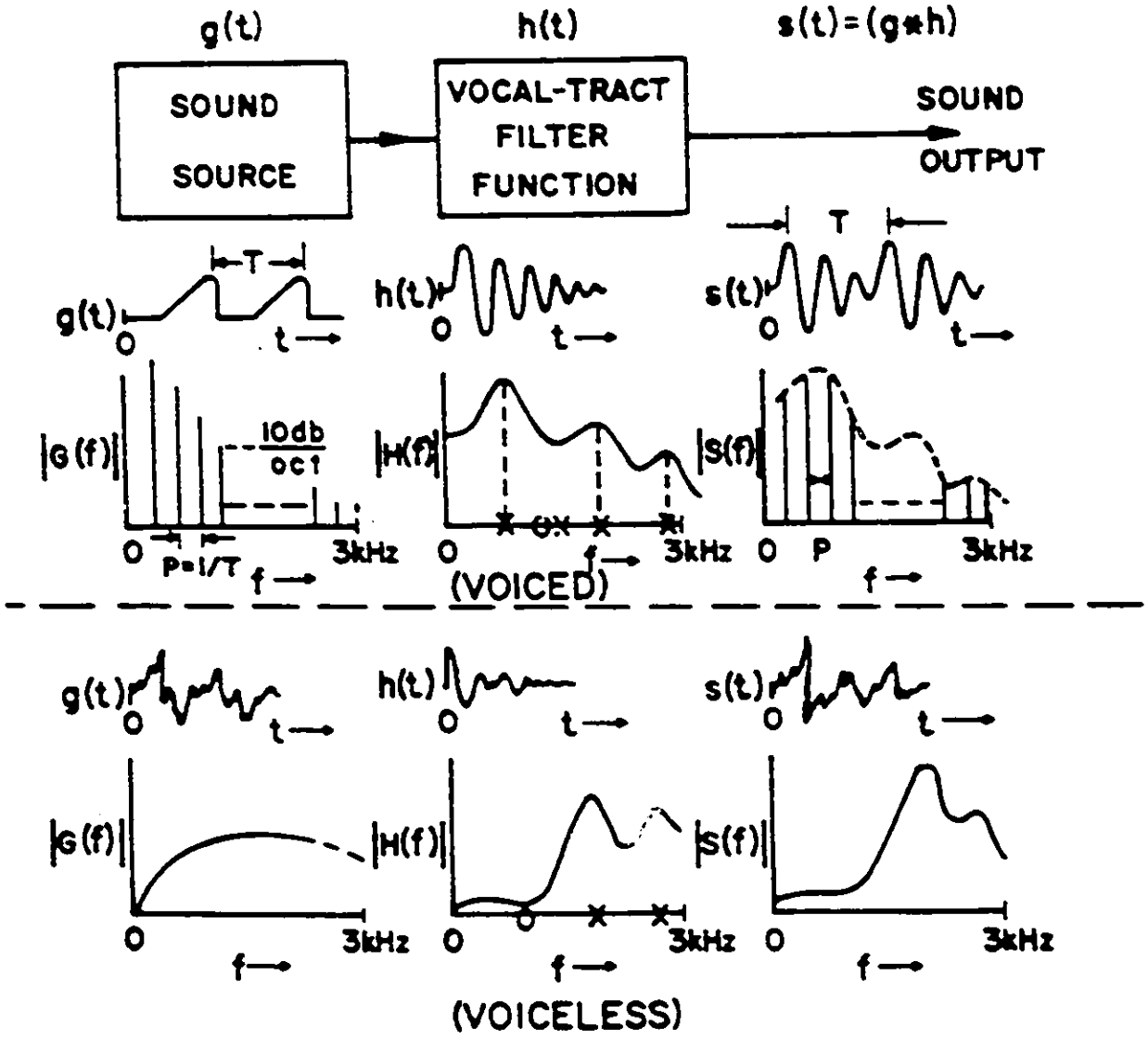

Figure 2.1: Linear system model of speech production [38]

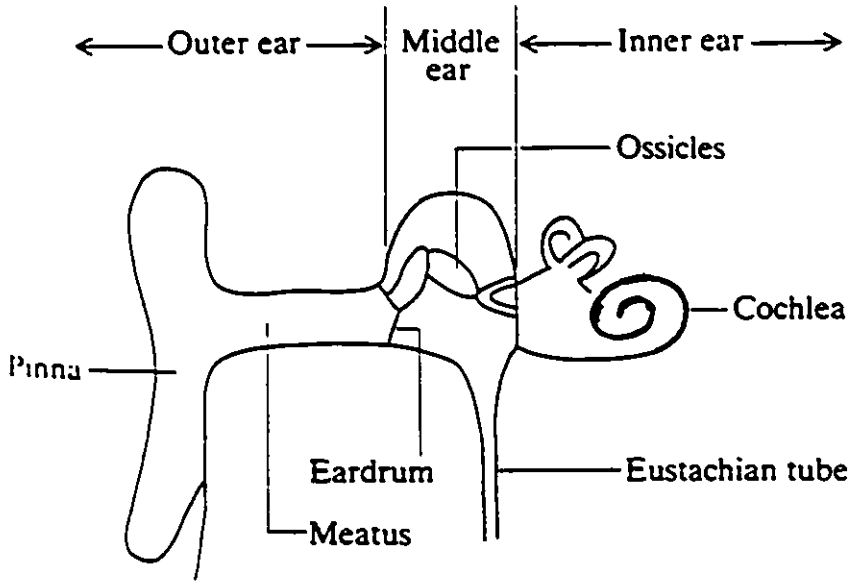

Figure 2.2: The Human ear [8]

ear which is filled with fluid. It also provides amplitude clipping to protect the inner ear from severe sound levels.

The cochlea is the organ located in the inner ear which transforms acoustical vibrations into electrical discharges on its neural fibers [10]. Independent physiological and psychoacoustic experiments have demonstrated that the frequency selectivity of the auditory system is already determined, or at least limited, at the cochlear level [11]. To a first order approximation, the periphery auditory system behaves as a band pass filterbank with a resolution of nearly one-third of an octave above 400 Hertz [12] (figure 2.3 and 2.4). The bandwidths and shapes of the cochlear nerve fibers can explain many characteristics of the interactions between frequency components in the ear.

2.4.3. Speech perception

Once the speech signal has been transformed into electrical discharges by the cochlea and carried to the auditory cortex by the nerves, speech perception is performed by the brain. Science knows less about speech perception than it does about speech production. In the following paragraphs we will discuss some perceptual aspects which are relevant to speech enhancement algorithms.

2.4.3.1. Auditory limits and thresholds

The frequency range of the auditory system is approximately 16 Hz to 16 kHz. The dynamic range of the hearing system is over 100 dB at 1 kHz. Speech intensity is usually measured in dB SPL (sound pressure level). The 0 dB reference point is 10^{-16} watt/cm^2 at 1 kHz. The minimum intensity at which sounds can be heard is known as the auditory threshold. Within 700 Hz and 7 kHz, the auditory threshold is constant within plus or minus 5 dB of 0 dB SPL. Speech covers only a small area within these physical limits, namely the auditory field [10].

2.4.3.2. Critical bands and masking

One of the most important perceptual phenomenon is masking. Masking occurs when one sound interferes with the perception of another. Frequency masking is caused by a simultaneous sound generally of a lower frequency. Frequency masking can be explained in terms of the peripheral auditory system.

Masking experiments show the effect of one sinusoidal tone or a narrow band of noise on another tone. These perceptual experiments provide plots of the additional amount of energy needed to hear a signal in the presence of a masker as a function of signal frequency. These curves agree approximately to the tuning curves of the peripheral auditory nerves. These frequency responses are known as critical bands.

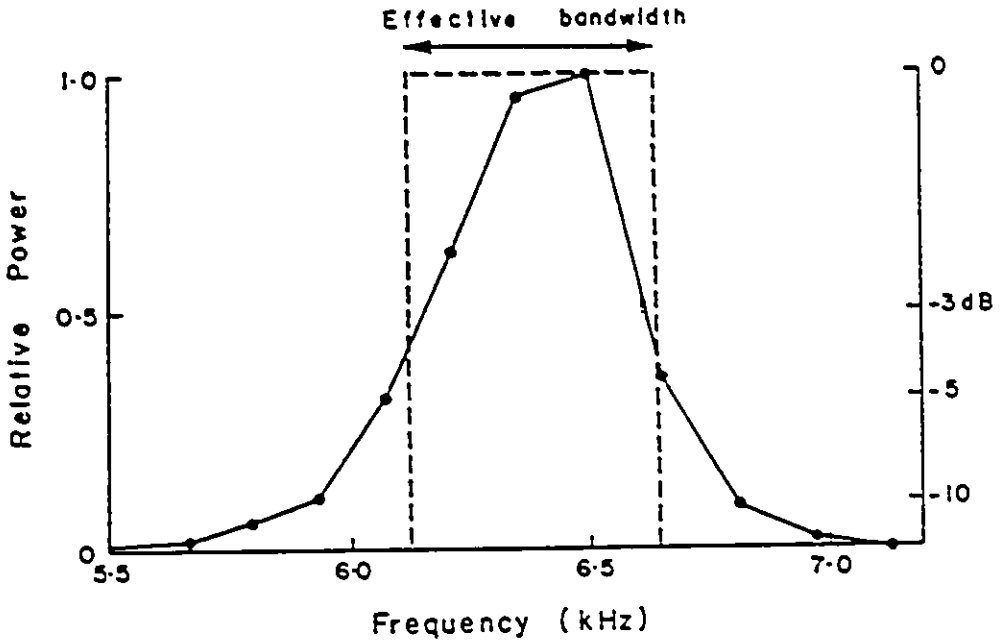

Figure 2.3: Physiological filter curve. This represents a frequency threshold curve of a cochlear nerve fiber expressed as a filter function on linear power and linear frequency coordinates. The equivalent rectangular bandwidth is shown by the dotted line. [11]

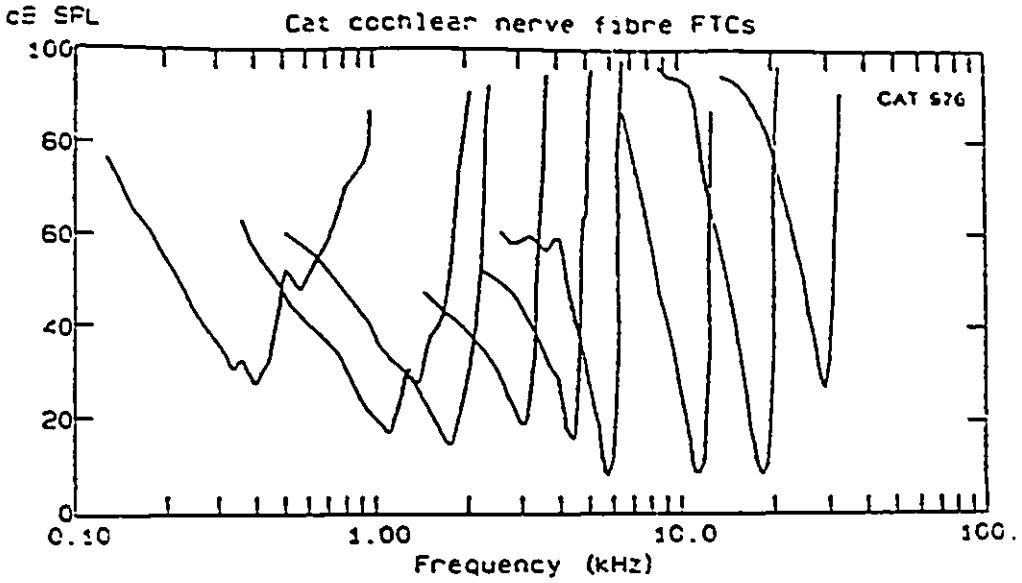

Figure 2.4: Frequency threshold curves for 9 cochlear nerve fibres recorded in the anaesthetized cat [11].

If two competing sinusoidal sounds are within one critical band of each other, the signal with the more energy masks the other one [10].

Frequency masking is not symmetric. Usually a lower frequency sound will tend to mask a high frequency sound. This can be explained by the flattening of the lower frequency skirts of the nerve tuning curves when the sound level is above 45 dB SPL (see figure 2.4) [11].

Successive signals sharing the same spectral emplacements can interfere with each other if their time difference is short enough. This is called temporal masking. Temporal masking is not as well understood as the frequency masking phenomenon.

2.4.3.3. Effects of distortions on intelligibility

Speech can be filtered by a band pass filter with cutoff frequencies of 400 Hz and 6 kHz without loss of intelligibility. As seen in figure 2.5, as the filter bandwidth is made narrower the intelligibility decreases [8]. Surprisingly, infinitely clipped speech, which retains its zero-crossing, low amplitude information, remains intelligible [13]. Actually, it is a perfect example of a signal with poor quality, but excellent intelligibility.

However, center clipping, which keeps only the high amplitude information, can tremendously reduce intelligibility even for small amplitude thresholds. This explains why low amplitude consonants are an important acoustic contribution to the intelligibility of speech even though they represent only a small fraction of the speech signal.

2.4.3.4. Amplitude and phase modification effects on speech intelligibility

It is generally agreed among speech researchers that short-time spectral amplitude is essential for speech perception whereas spectral phase information is less important. When simple perceptual experiments are conducted with sinusoidal signals, the hearing system can detect changes in amplitude and frequency but not changes in phase [14]. This result is often generalized to more complicated signals. Specifically, it is generally accepted that the auditory system is insensitive to the relative phases in the sinusoidal components of a complex signal such as speech or music. This explains why processing speech with an all-pass filter results in almost no audible difference.

However this statement is only partially true because when the relative phase components are changing rapidly, the ear is sensitive to phase effects [14]. It can be easily demonstrated by processing speech by a phase vocoder and replacing the phase angles by fast-varying random components. The synthesized speech signal becomes unnatural and less intelligible.

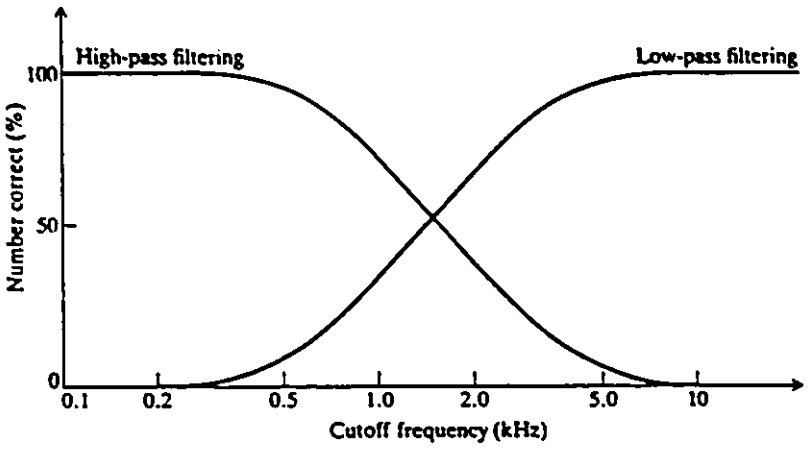

Figure 2.5: Effect of bandpass filtering on intelligibility of speech [8] .

3. Overview of speech enhancement algorithms

3.1. Impulsive noise reduction algorithms

Pulses are characterized by their high amplitudes and short-time durations. If pulses are the result of bit errors in digital transmission of speech samples, they have a typical duration of only few samples. However if they result from atmospheric disturbances in radio communications such as lightning, or scratches on a vinyl record surface, their durations can be as long as 20 ms [15]. Furthermore, their duration can be dilated by the ringing effect of filters used at the front-end of processing equipment such as radio receivers, audio preamplifiers or even the anti-aliasing filters of analog to digital conversion systems. Therefore, impulsive noise reduction schemes exploit detection-interpolation strategies as shown in figure 3.1.

Speech amplitude peaks are approximately equal to three (3) times the long-term signal root mean-square (rms) value [15]. This observation leads to a simple pulse detection scheme used in some pulse removal systems [15]. If the signal amplitude is over a pre-determined threshold, the sample under scrutiny and few neighboring speech samples are declared to be corrupted by a pulse. More sophisticated techniques [16] perform the detection in the so-called LPC residual domain. This method enables a better discrimination of pulses from high amplitude pitch attacks in speech and music signals thus lowering the number of false pulse detections.

Once a segment of the signal has been categorized as contaminated by pulses, it is set to zero and interpolation techniques are then used to estimate the missing samples. The most simple approach [15] cuts and pastes adequately windowed versions of adjacent portions of the signal. Recently, more elaborated techniques have been proposed. Vaseghi and Rayner [16] use a forward-backward linear prediction algorithm which capitalizes on long-term and short-term correlation parameters. This interpolation method was applied to the restoration of speech signals. Segments up to 30 samples long selected at random, were replaced by their interpolated estimates. Up to 20% of the speech signal was discarded and then re-estimated with no audible differences.

3.2. Periodic noise reduction algorithms

Periodic noise components can be removed by filtering techniques. Stationary filters can be used if the frequencies of the interference are fixed. This is the case if a 60 Hz hum and its associated harmonics have to be filtered. Analog or digital banks of notch filters can be easily implemented to attenuate such periodic interferences.

However, in most cases the frequency of the periodic interferences vary in time. A spectrogram of noise recorded at the pilot position in a Buccaneer fighter jet-airplane flying at a speed of 190 knots at an altitude of 1000 feet is shown in figure

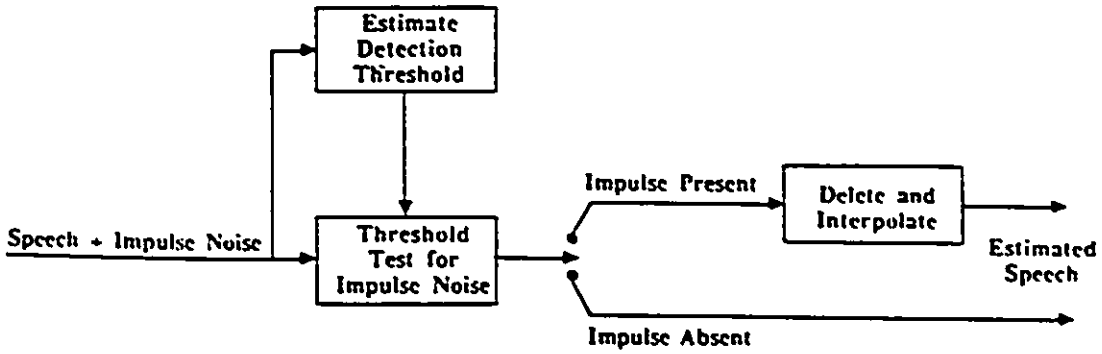

Figure 3.1: A canonical representation for an impulse noise reduction system [1].

3.2 [17]. There is an evident periodic component at 2.8 kHz which has a Doppler shift behavior. Under other flight conditions, this periodic component disappears or is located at different frequencies. Here an adaptive filter can be used to track the periodic components and to remove the interference. Non-linear short-term spectral manipulation can be used to track and filter periodic noise in the frequency domain [15]. Phonemes have a typical duration of about 75 ms, therefore if a spectral component is steady in frequency and amplitude for more than 200 ms, it can be zeroed since it is likely an interference signal. This simple but effective scheme is used in the Speech Enhancement Unit (SEU) developed by the Rome Air Development Center (RADC) [15].

3.3. Wideband noise reduction algorithms

3.3.1. Techniques based on the periodicity of speech

Some speech enhancement methods capitalize on the observation that voiced speech is periodic. If an accurate estimate of the pitch frequency can be obtained at every sample time, a time-variant comb filter can be designed to filter out all the non-harmonic (non-voiced) components. The comb filter is defined by this simple equation [7]

$$y(n) = \sum_{k=-m}^{m} C_k \, x(n-kL). \tag{3.1}$$

Here m is a small constant, the C_ks are the filter coefficients which decay proportionally to k, $x(n)$ is the input signal, $y(n)$ is the output signal and L is the estimated fundamental (pitch) period in samples. The filter output is simply an average of delayed and weighted versions of the input signal. The accurate estimation of the pitch period is crucial for the success of this method. However, pitch estimation in noise can be very difficult. Alternatively, comb filtering can be performed in the frequency domain. The Fourier transform is taken and the spectral components which are harmonics of the fundamental are extracted and the other components are zeroed. Taking the inverse Fourier transform produces the enhanced speech signal.

This approach is inefficient for unvoiced speech since unvoiced phones do not exhibit periodicity. This method is efficient to increase the SNR but does not increase the intelligibility even if perfect knowledge of the fundamental frequency is assumed [18].

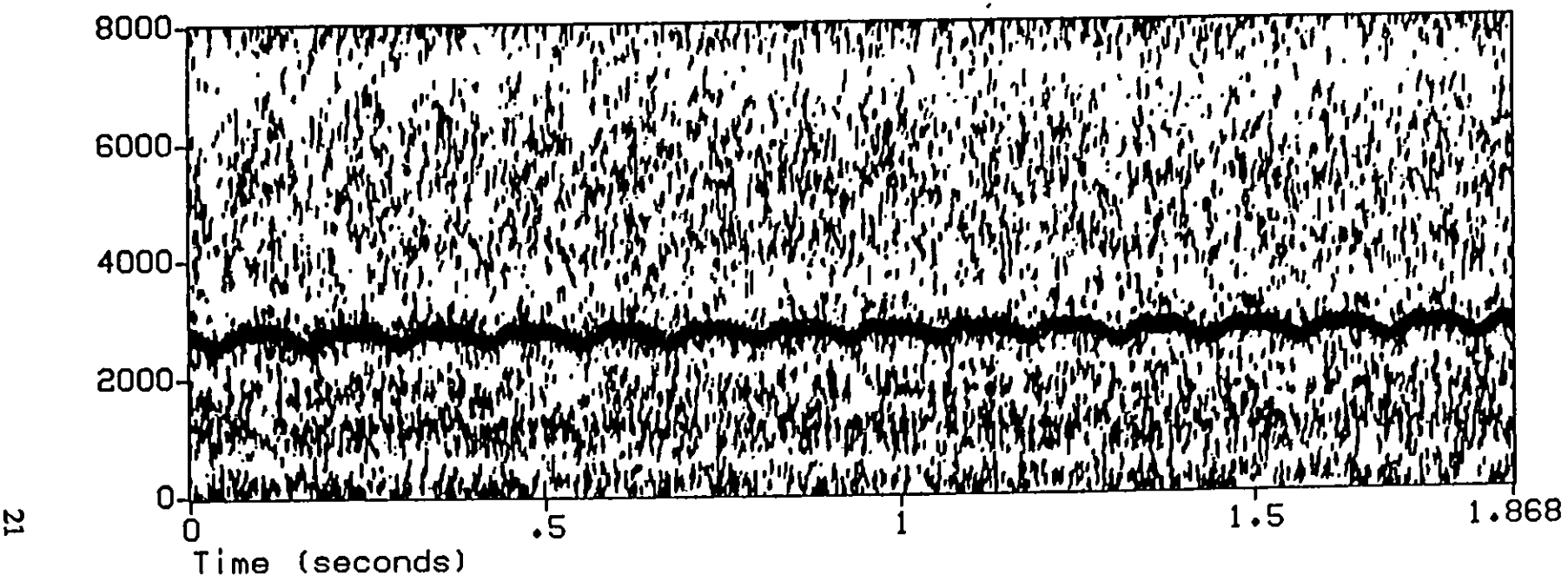

Figure 3.2: Spectrogram of jet-airplane noise recorded in the cockpit.

3.3.2. Techniques based on re-synthesis of speech

Linear prediction coding (LPC) has been used successfully in text-to-speech and voice coding systems [7]. A typical LPC synthesizer is shown in figure 3.3. In voice coding applications, the LPC filter coefficients are estimated from the speech signal and then transmitted via the channel with some information about the residual error signal (the glottal excitation). The signal is re-synthesized at the receiver end with the coefficients and a reconstructed glottal excitation signal. In speech enhancement applications the LPC parameters are estimated from the noisy source. The idea is tempting, if perfect estimation of the parameters is achieved, undegraded re-synthesized speech is produced.

However, LPC analysis which is one of the most powerful tools of the speech researcher's arsenal, is very sensitive to degradations caused by broadband noise. Actually, LPC analysis can be interpreted as a way to decompose the speech signal in its vocal tract filter coefficients (spectral information on formants) and the LPC error signal (glottal excitation signal). When the noisy signal is in the LPC domain, the speech components estimation problem remains complete.

3.3.3. Template-based approaches

Recently template-based approaches have been proposed for speech enhancement. Quatieri and McAulay [19] use a sine-wave analysis-synthesis model to characterize the speech signal. This model is defined by a voicing probability, sine wave amplitudes and the fundamental frequency. Instead of using a suppression filter which operates independently on each sine-wave amplitude, their system attempts to exploit frequency dependance via spectral templates. During the training phase, spectral templates are constructed every 10 ms from the speaker speech. These templates contain the information needed to synthesize a short period of the signal. In the enhancement mode, each noisy frames is analyzed and a spectral estimate is obtained by selecting a template in the training set which minimizes a distortion measure between the clean templates and the current frame spectral information. At a later stage, instead of taking a hard decision for the template selection among the training set, they developed a soft decision method which minimizes "warble" in the reconstructed signal caused by frame-to-frame discontinuities.

Another template approach is being developed at AT&T by Ephraim [20]. Hidden Markov models (HMM) are used to model speech and noise statistics. Then the speech samples are estimated by minimum mean-square error estimation techniques. The parameters of the Markov chains need to be evaluated from training sequences of clean speech and noise samples.

These template methods look promising but so far they have not produced significantly better results than the techniques based upon short-time spectral estimation. Moreover, they are computationally intensive and they require a training phase.

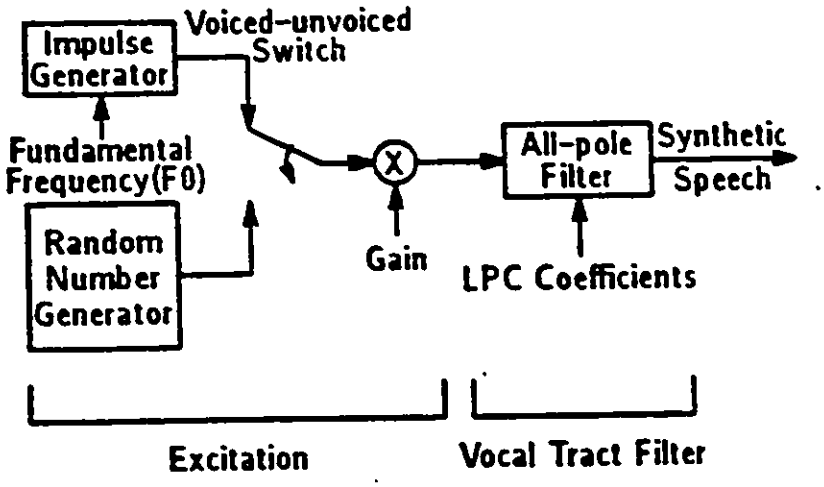

Figure 3.3: Linear predictive coding (LPC) Synthesizer [7].

3.3.4. Techniques based upon short-time spectral analysis and estimation

The algorithms presented in this section focus on the major importance of the short-time spectral amplitude for speech perception. Because the auditory system is particularly sensitive to the spectral amplitude of the speech signal and ignores various aspects of the spectral phase information, many speech enhancement systems estimate the short-time spectral amplitude of the speech signal and recombine the noisy phase of the incoming signal to generate the enhanced speech.

In addition, since speech signals exhibit spectral clustering, short-time spectral decomposition of the noisy signal makes it possible to process the signal in a domain where speech and noise are more easily separated. The main difference between these algorithms is in the approach they utilize to estimate the short-time spectral amplitude. A canonical diagram of this family of algorithms is shown in figure 3.4.

3.3.4.1. Power spectral subtraction

When analyzed in sections of 20 to 30 ms, speech can be modeled as a stationary signal. If it is degraded by an additive random noise signal, we can write the following relation for the composite signal [5]

$$y_w(n) = x_w(n) + n_w(n) \tag{3.2}$$

here $y_w(n)$ is the windowed noisy observation, $x_w(n)$ the windowed speech signal and $n_w(n)$ the windowed noise signal. Under the hypothesis that these processes are uncorrelated, the power spectral density of the measurement becomes

$$S_y(\omega) = S_x(\omega) + S_n(\omega). \tag{3.3}$$

Intuitively, an estimate of the speech power spectral density can be obtained by subtracting an estimate of the noise power from the measured power spectral density [5]

$$S'_x(\omega) = S_y(\omega) - S'_n(\omega). \tag{3.4}$$

Assuming the power spectra are obtained by squaring the Fourier transforms, we can write

$$| X'(e^{j\omega}) |^2 = | Y(e^{j\omega}) |^2 - | N'(e^{j\omega}) |^2 \tag{3.5}$$

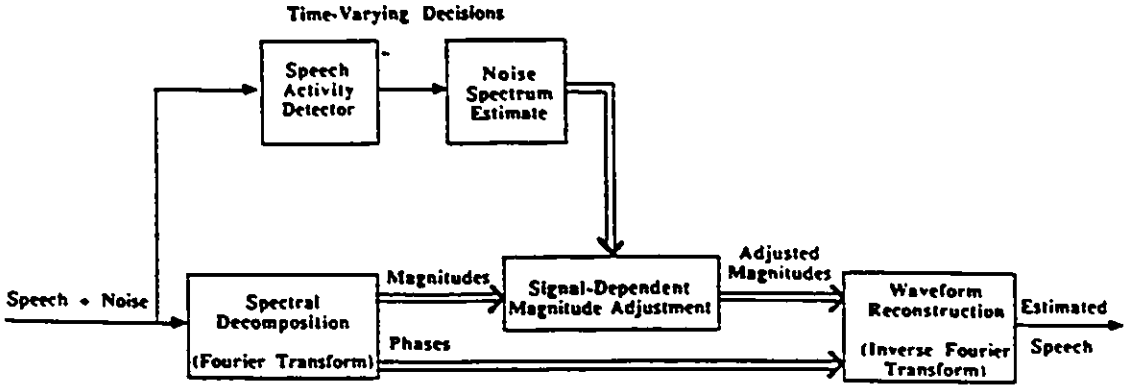

Figure 3.4: Canonical diagram for frequency domain noise reduction techniques. Double lines indicate a set of component values [1].

where $X(e^{j\omega})$ is the Fourier transform of $x_w(n)$ and X' and N' denote estimates. Then, the phase of the noisy observation is appended to the short-time spectral amplitude estimate to produce the estimated Fourier transform of the speech signal

$$X'(e^{j\omega}) = | X'(e^{j\omega}) | e^{j\theta} \qquad \text{where } \theta \text{ is the phase of } Y(e^{j\omega}). \tag{3.6}$$

Then it suffices to take the inverse Fourier transform to produce the enhanced speech. Practically, the noise power is estimated and updated during non-speech activities. If the noise is purely stationary or if its statistics are slowly varying, the estimate is accurate. This method is known as the power spectral subtraction. The power spectral subtraction method can also be formulated as the maximum likelihood (ML) estimator of the speech spectral variance if the noise and speech spectral components are modeled as independent Gaussian random processes [20,22]. Many variants of this method have been devised [5,21]. They differ in the way the noise spectral estimate is obtained and in the way the spectral subtraction is performed. The subtraction can be performed on the root compressed or expanded envelope. Furthermore an overestimate of the noise can be subtracted. The generalized spectral subtraction relation becomes [21]

$$| X'(e^{j\omega}) |^{\alpha} = | Y(e^{j\omega}) |^{\alpha} - \beta [| N'(e^{j\omega}) |^{\alpha}], \tag{3.7}$$

α is the root compression or expansion factor and β is the multiplicative factor which multiplies the noise estimate. A block diagram of a generalized spectral subtraction system is drawn in figure 3.5.

3.3.4.2. Soft-decision noise suppression filter

Assuming the spectral decomposition of a noisy speech signal is performed by a filterbank or equivalently by short-time Fourier transform, the noisy spectral measurement is [22]

$$y_n = s_n + w_n. \tag{3.8}$$

Here y_n is the complex spectral observation for channel n (or equivalently for frequency index n), s_n is the speech spectral component and w_n represents the noise spectral component.

Assuming the noise is a Gaussian random process and s_n is a deterministic waveform of unknown amplitude and phase, i.e $s_n = A\, e^{j\theta}$, the probability density function of the channel output can be written as [22]

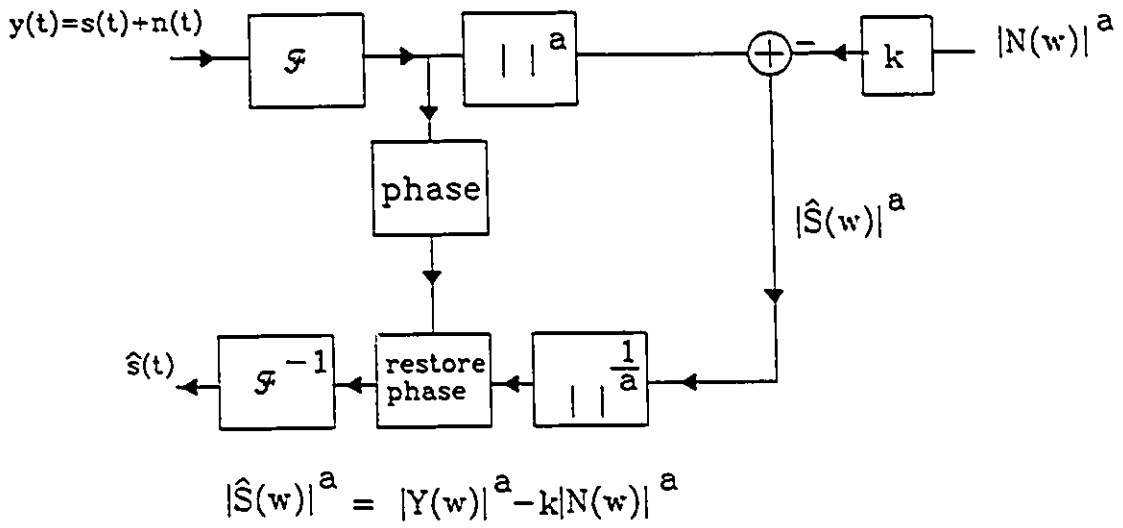

Figure 3.5: Generalized spectral subtraction.

$$p(y_n \mid A, \Theta) = \frac{1}{\pi \lambda_w(n)} \exp\left[- \frac{|y_n|^2 - 2A\,Re\,(e^{-j\Theta}\, y_n) + A^2}{\lambda_w(n)} \right] \qquad (3.9)$$

where $\lambda_w(n)$ is the channel (frequency) dependent noise variance.
An estimate of the speech amplitude parameter is found by maximizing this probability function averaged on the angle parameter Θ . The estimator becomes

$$A' = \frac{1}{2}\left(|y_n| + \sqrt{|y_n|^2 - \lambda_w(n)} \right) \qquad (3.10)$$

where $|y_n|$ is the measured amplitude at the output of the channel and $\lambda_w(n)$ is the noise variance. In order to provide more attenuation when the signal only contains noise, McAulay *et al.* [22] modified this maximum likelihood estimator to introduce a two-state model which takes into account the uncertainty of speech presence. The amplitude estimator now includes a new term, the probability that speech is present in a given channel, leading to this new estimate

$$A' = \frac{1}{2}\left(|y_n| + \sqrt{|y_n|^2 - \lambda_w(n)} \right) P(H1 \mid |y_n|) \qquad (3.11)$$

here *H1* represents the hypothesis that speech is present; the probability term is expressed as

$$P(H1 \mid |y_n|) = \frac{\exp(-\xi)\, Io\left[2\sqrt{\xi \frac{|y_n|^2}{\lambda_w(n)}} \right]}{1 + \exp(-\xi)\, Io\left[2\sqrt{\xi \frac{|y_n|^2}{\lambda_w(n)}} \right]}. \qquad (3.12)$$

This statistically derived spectral estimator generates a family of attenuation curves which depend on the *a priori* SNR : $\xi = \frac{A^2}{\lambda_w(n)}$.

These curves, which are drawn in figure 3.6, have different shapes depending on the *a priori* SNR. McAulay did not estimate the *a priori* SNR in his experiments, ξ simply controls the amount of desired suppression in a given channel and is referred as the suppression factor.

3.3.4.3. Wiener filter

In systems based upon Wiener filtering, the desired signal s(t) and the measured noisy signal x(t) are modeled as wide-sense stationary stochastic processes [23]

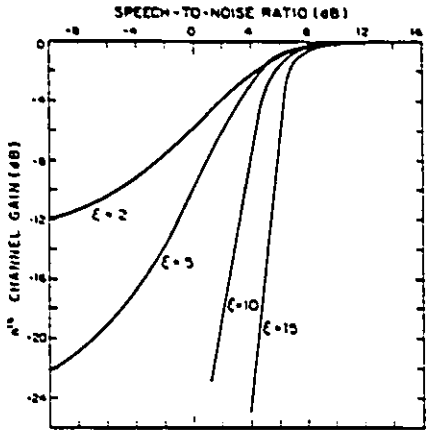

Figure 3.6: Soft suppression curves for maximum likelihood estimation [22].

$$x(t) = s(t) + v(t). \tag{3.13}$$

The linear estimate which minimizes the mean-square error is given by

$$s'(t) = \int_{-\infty}^{\infty} h(\alpha)\, x(t-\alpha)\, d\alpha\ . \tag{3.14}$$

$s'(t)$ is therefore the output of a non-causal time-invariant system with impulse response $h(t)$ and input $x(t)$. In order to find $h(t)$ we invoke the orthogonality principle [23] which states that the mean-square error is minimum if the error is orthogonal to the data, hence

$$E\left[\left[s(t) - \int_{-\infty}^{\infty} h(\alpha)\, x(t-\alpha)\, d\alpha\right] x(t-\tau)\right] = 0 \quad \text{all } \tau, \tag{3.15}$$

rewriting in terms of the autocorrelation functions, yields

$$R_{sx}(\tau) = \int_{-\infty}^{\infty} h(\alpha)\, R_{xx}(\tau-\alpha)\, d\alpha \quad \text{all } \tau. \tag{3.16}$$

This integral is easily solved by taking the Fourier transform on both sides and observing that the right hand side is a time domain convolution, therefore we obtain

$$S_{sx}(\omega) = H(\omega)\, S_{xx}(\omega) \tag{3.17}$$

or

$$H(\omega) = \frac{S_{sx}(\omega)}{S_{xx}(\omega)}. \tag{3.18}$$

This result is called the non-causal Wiener filter. If the noise process and the desired signal are orthogonal (uncorrelated), we can write

$$S_{sx}(\omega) = S_{ss}(\omega) \tag{3.19}$$

and

$$S_{xx}(\omega) = S_{ss}(\omega) + S_{vv}(\omega) \tag{3.20}$$

then

$$H(\omega) = \frac{S_{ss}(\omega)}{S_{ss}(\omega) + S_{ww}(\omega)} \tag{3.21}$$

and the speech power spectral estimate can be written as

$$S'_{ss}(\omega) = S_{xx}(\omega)\, H(\omega). \tag{3.22}$$

This linear estimate can be shown to be equal to the *optimum* non-linear mean-square estimate if the noise and speech processes are zero mean Gaussian random variables [23]. Because the Wiener filter is zero phase, the phase of the noisy observation becomes the phase of the estimated speech power spectrum. Consequently, this method only estimates the spectral amplitude of the speech signal.

In speech enhancement systems, the power spectral densities for the noise and speech processes are evaluated by performing short-time spectral analysis. Because the speech signal and the noise interference can be considered to be stationary only for short-time periods, the power spectral densities are constantly evaluated and updated. Different techniques can be used to estimate the power spectra. The attenuation curve for the Wiener filter is drawn in figure 3.7 with the attenuation curves of the two preceding methods we described.

3.3.4.4. Minimum mean-square error short-time spectral amplitude estimator

Ephraim and Malah [24] made the observation that because the spectral subtraction algorithm is the maximum likelihood estimator of the speech component variance under Gaussian assumptions, and the Wiener filter is the optimum (because it minimizes the mean-square error under Gaussian hypotheses) estimate of the speech spectral components, they are both non optimum *amplitude* spectral estimators under the assumed statistics. Consequently, their study focused on developing a minimum mean-square short-time spectral amplitude (STSA) estimator. The following derivation is taken from their paper [24].

Several efforts have been made in the past to estimate the probability distribution of speech spectral components. However, a true statistical model seems difficult to attain. Because spectral analysis is usually done on segments of about 1000 samples, each Fourier expansion coefficients is after all a weighted sum of random variables. Consequently, the central limit theorem motivates the utilization of Gaussian statistical models for both speech and noise spectral components.

Again, let the observed signal be

$$y(t) = x(t) + d(t) \qquad 0 < t < T \tag{3.23}$$

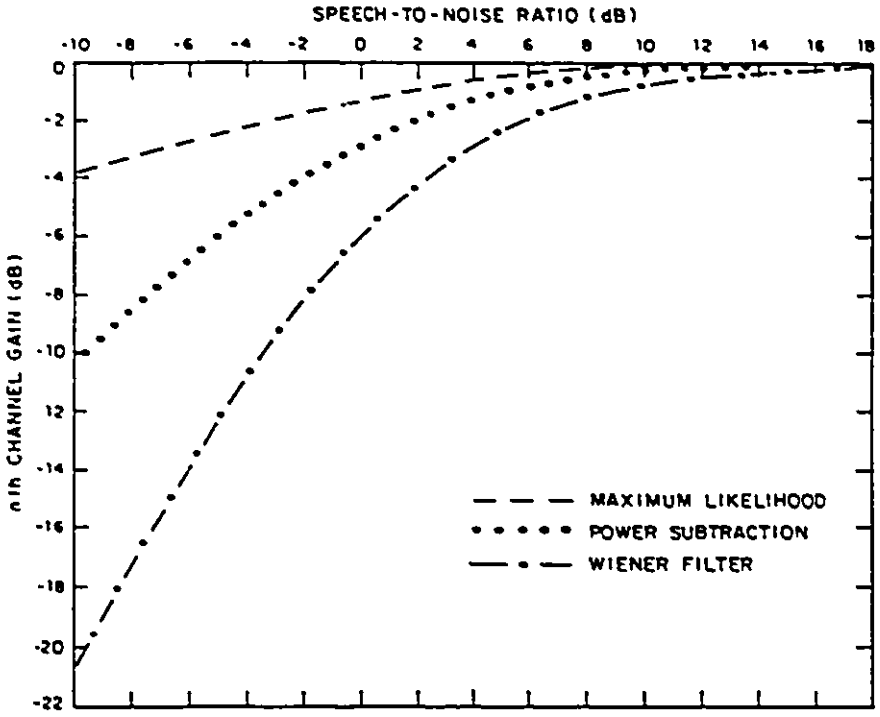

Figure 3.7: Attenuation curves for maximum likelihood estimation, power subtraction and non-causal Wiener filter [22].

where $x(t)$ is the speech waveform and $d(t)$ is the additive noise. This signal is spectrally decomposed by short-time Fourier transform yielding

$$Y_k = X_k + D_k \tag{3.24}$$

where $Y_k = R_k \exp(j\varpi_k)$ and $X_k = A_k \exp(j\alpha_k)$.

The minimum mean-square estimate of the amplitude A_k is sought, it is given by the conditional mean of A_k knowing the measured value Y_k [25]

$$A'_k = E\{A_k \mid Y_k\} = \frac{\int_0^{\infty} \int_0^{2\pi} A_k \; p(Y_k \mid A_k, \alpha_k) \; p(A_k, \alpha_k) \; d\alpha_k \, dA_k}{\int_0^{\infty} \int_0^{2\pi} p(Y_k \mid A_k, \alpha_k) \, p(A_k, \alpha_k) \; d\alpha_k \; dA_k}. \tag{3.25}$$

$p(Y_k \mid A_k, \alpha_k)$ is the probability density of the spectral observation given the speech phase and amplitude components are known. Under Gaussian assumptions D_k is circular Gaussian therefore

$$p(Y_k \mid A_k, \alpha_k) = \frac{1}{\pi \lambda_d(k)} \exp\left[- \frac{|Y_k - A_k e^{j\alpha_k}|^2}{\lambda_d(k)} \right] \tag{3.26}$$

and $p(A_k, \alpha_k)$ is given by

$$p(A_k, \alpha_k) = \frac{A_k}{\pi \lambda_x(k)} \exp\left[- \frac{A_k^2}{\lambda_x(k)} \right] \tag{3.27}$$

where $\lambda_d(k) = E(|D_k|^2)$ and $\lambda_x(k) = E(|X_k|^2)$ are respectively the noise and speech variances of the k^{th} spectral component. The solution to the integral equation gives the amplitude estimator

$$A'_k = \Gamma(1.5) \frac{\sqrt{v_k}}{\gamma_k} M(-0.5 ; 1 ; -v_k) \, R_k \tag{3.28}$$

$$\text{with} \quad v_k = \frac{\xi_k}{1 + \xi_k} \gamma_k .$$

Here $\Gamma(\cdot)$ is the gamma function, M(a ; c ; x) is the confluent hypergeometric function, ξ_k and γ_k are interpreted respectively as the *a priori* and *a posteriori* signal to noise ratios. They are defined by

$$\xi_k = \frac{\lambda_x(k)}{\lambda_d(k)} \tag{3.29}$$

$$\gamma_k = \frac{R_k^2}{\lambda_d(k)}. \tag{3.30}$$

The amplitude estimator can be conveniently expressed as a gain function applied on the measured envelope

$$G_{mmse}(\xi_k, \gamma_k) = \frac{A'_k}{R_k}. \tag{3.31}$$

Figure 3.8 shows the gain curves for different *a priori* and *a posteriori* SNR pairs. For high *a posteriori* signal to noise ratios, $G_{mmse}(\xi_k, \gamma_k)$ has an asymptotic behavior and only depends on the *a priori* SNR ξ_k. For that specific case, It can be shown that $G_{mmse}(\xi_k, \gamma_k)$ is equal to the Wiener gain. Therefore

$$G_{mmse}(\xi_k, \gamma_k) = \frac{\xi_k}{1 + \xi_k} = G_{wiener}(\xi_k) \quad \text{when } \xi_k >> 1. \tag{3.32}$$

Like McAulay *et al.* [22], Ephraim and Malah introduced in their model the signal presence uncertainty. With this two-state model, the amplitude estimator becomes

$$A'_k = \frac{\Lambda(Y_k, q_k)}{1 + \Lambda(Y_k, q_k)} E\{ A_k | Y_k, H_k^1 \}. \tag{3.33}$$

$\Lambda(Y_k, q_k)$ is the generalized likelihood ratio, q_k is the probability of signal absence in the k^{th} spectral component, and H_k^1 denotes the hypothesis of signal presence in the k^{th} spectral component. Under the assumed statistical models, we can write the following expressions

$$\Lambda(\xi_k, \gamma_k, q_k) = \frac{(1 - q_k)}{q_k} \frac{\exp(v_k)}{1 + \xi_k} \tag{3.34}$$

$$n_k = \frac{\lambda_x(k)}{\lambda_d(k)} \tag{3.35}$$

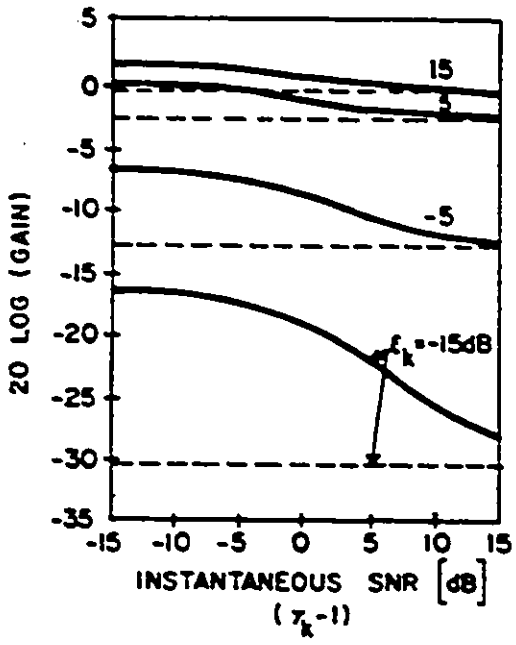

Figure 3.8: Curves describing the MMSE STSA gain Gmmse(ξ_k,γ_k). The dashed line represents the Wiener gain [24].

$$\xi_k = \frac{n_k}{(1 - q_k)}. \tag{3.36}$$

The spectral amplitude estimator can be conveniently expressed in terms of the previously derived estimator $G_{mmse}(\xi_k, \gamma_k)$, yielding

$$A'_k = \frac{\Lambda(\xi_k, \gamma_k, q_k)}{1 + \Lambda(\xi_k, \gamma_k, q_k)} G_{mmse}(\xi_k, \gamma_k) R_k \tag{3.37}$$

We see that the *a priori* SNR is modified to take into account the speech signal uncertainty. The resulting new set of curves is plotted in figure 3.9.

The evaluation of the frequency dependant *a priori* SNR proved to be a critical factor for the performance of this algorithm. The *a priori* SNR should be re-estimated at every analysis frame since speech is not stationary. The method which gave the best results in their study is the decision-directed estimation approach. This *a priori* SNR estimate is based upon the definition of ξ_k. Since ξ_k can be expressed by either

$$\xi_k(n) = \frac{E\{A_k^2(n)\}}{\lambda_d(k,n)} \tag{3.38}$$

or

$$\xi_k(n) = E\{\gamma_k(n) - 1\}, \tag{3.39}$$

then a cogent estimator can be written as

$$\xi_k(n) = \alpha \frac{A_k^2(n\text{-}1)}{\lambda_d(k, n\text{-}1)} + (1\text{-}\alpha)(\gamma_k(n) - 1) \quad 0 <= \alpha < 1. \tag{3.40}$$

This *a priori* SNR estimate depends on the current measured SNR $\gamma_k(n)$ -1 and the last frame estimated spectral amplitude $A_k(n\text{-}1)$; α is determined by listening tests and is used to compromise between noise attenuation and speech distortion. In their experiments computer-generated stationary noise was added to the speech signal, consequently $\lambda_d(k,n)$ was simply estimated once from a 500 milliseconds section of the noise process.

Ephraim and Malah tested their algorithm on speech material degraded by white noise at different signal to noise ratio (SNR) levels. An informal test was done to compare the performance of their MMSE STSA with the power spectral subtraction [5], the soft-decision noise suppression filter of McAulay *et al.* [22] and the Wiener filter. They claim their algorithm produced the best results. Specifically, a significant reduction of the noise is achieved and the residual noise is colorless.

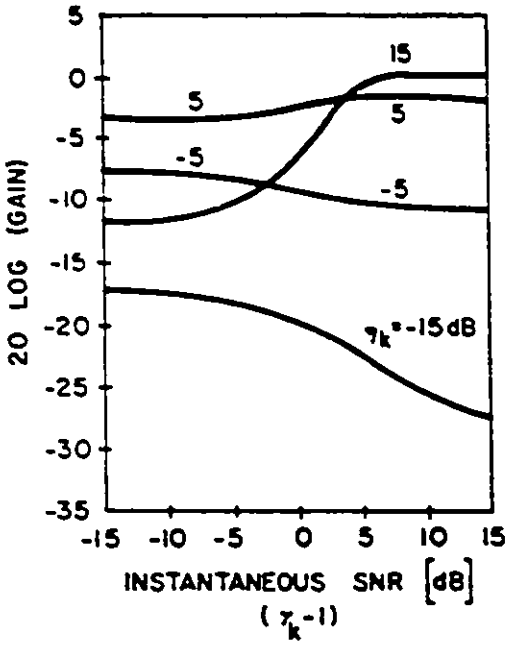

Figure 3.9: Curves depicting the MMSE STSA gain Gmmse taking into account speech presence uncertainty. The probability of speech absence is set to qk =0.2 [24].

3.3.4.5. Parametric spectral amplitude estimation

Many spectral amplitude estimation strategies which are proposed in the literature for speech enhancement can be expressed in closed form as a gain function applied on the measured spectral envelope. If $|Y(\omega)|$ is the measured spectral envelope, and $|X(\omega)|$ is the estimated envelope, $G(\omega)=\frac{|X(\omega)|}{|Y(\omega)|}$ is the frequency dependent gain function. When expressed as gain functions, many speech enhancement spectral estimation strategies can be interpreted as a specific case of the following parametric Wiener filter [26]

$$G(\omega)=\left[\frac{\xi(\omega)}{\xi(\omega)+T}\right]^{r}. \tag{3.41}$$

Here ξ is the *a priori* signal to noise ratio, T is a threshold where the gain starts to decrease and r is the exponent which imposes the sharpness of the attenuation curve shape. Evidently, if $r=1$ and $T=1$, the classical Wiener filter is realized. Table 3.1 shows the values which can be attributed to r an T, and the way the *a priori* SNR is calculated to get specific amplitude estimators proposed in the literature.

Method	r	T	$\xi(\omega)$ ***a priori*** **SNR**	**references**
Power spectral subtraction	0.5	1	$\frac{\|Y(\omega)\|^2}{\|N(\omega)\|^2}-1$	[26] [21] [5]
Wiener filter	1	1	decision directed	[24]
Generalized spectral subtraction $\|X(\omega)\|^{\alpha}=\|Y(\omega)\|^{\alpha}-\beta\|N(\omega)\|^{\alpha}$	$\frac{1}{\alpha}$	1	$\frac{\|Y(\omega)\|^{\alpha}}{\beta\|N(\omega)\|^{\alpha}}-1$	[21] [26]
Non-linear spectrum processor.	1	1	$\left[\frac{\|Y(\omega)\|}{\alpha}\right]^{N}$	[27]
Expander function	$2r$	λ	$\|Y(\omega)\|$	[6]
Soft suppression filter $\|X(\omega)\|=0.5\|Y(\omega)\|+.5\sqrt{Y\|(\omega)\|^2-\|N(\omega)\|^2}$	1	1/4	$\frac{\|X(\omega)\|^2}{\|N(\omega)\|^2}$	[22]

Table 3.1: Parametric Wiener filter realizations.

3.3.5. The INTEL method

The INTEL method was developed in the mid-1970s by Nicolet Scientific Corporation [14] for the Rome Air Development Center. This algorithm is used in the Speech Enhancement Unit (SEU) manufactured by Martin Marietta Aerospace as the wideband noise reduction sub-system constituent.

The INTEL process was empirically developed as an attempt to better separate the speech signal from the wideband interference. The INTEL process analyses the signal in the pseudo-cepstral domain. A block diagram of the method is shown in figure 3.10.

Here is the description of the process [14] :

1. The incoming signal is divided into overlapping (50%) segments of about 50 ms long;

2. Triangular weighting is applied to the speech segments to reduce spectrum sidelobes during the process and smooth the transition between processed frames;

3. A discrete Fourier transform (DFT) is taken on the array of data;

4. The absolute value of the transform is taken and the phase is saved for future utilization;

5. The upper half of the DFT transformed array is set to zero. The n^{th} root of the remaining part is taken. This is called root compression;

6. A sign reversal operation is done on the odd numbered DFT bins. This is of no consequence for the algorithm since the operation simply has the effect of shifting the origin of the second transform to a more convenient location.

7. The array is DFT transformed a second time. The result is called the pseudo-cepstrum. It differs from the cepstrum because root compression is used instead of logarithmic compression;

8. Then, either some pseudo-cepstral samples adjacent to the origin are attenuated or an estimate of the noise pseudo-cepstrum is subtracted from the processed speech;

9. Then the inverse operations are done to come back to the time domain;

10. The enhanced speech is reconstructed by overlapping and adding appropriately delayed processed signal samples. In the absence of modifications, the system reduces to an identity in virtue of the 50 % overlapping analysis-synthesis method.

The pseudo-cepstral transformation moves most of the noise down to within a few samples of the origin, whereas vocalic speech will be spread over the entire pseudo-cepstrum band. Intuitively this can be explained by considering the effect of the DFT transforms on speech and noise data. If wideband white noise is DFT transformed once, the resulting spectral amplitudes will tend to have a constant value. Root compression will minimize variations among adjacent components. Assuming the spectral amplitudes are truly constant in a given frame after the root compression process, the second DFT transform will create a single pulse at the

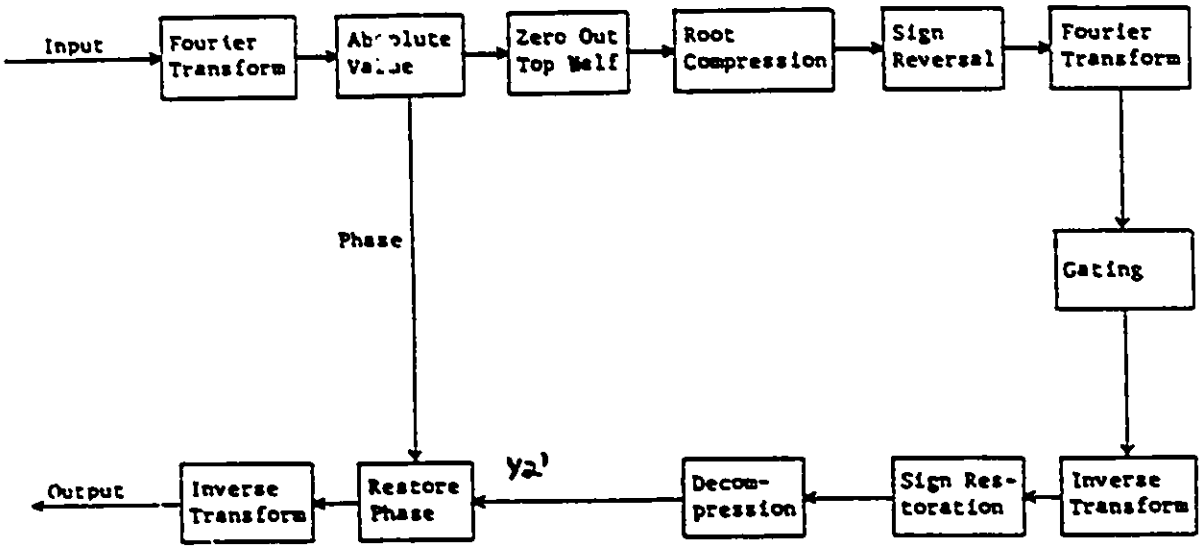

Figure 3.10: Block diagram of INTEL method [10].

pseudo-cepstral origin. On the other hand, vocalic speech DFT spectra will be rich in harmonics but will not exhibit constant amplitude, therefore the speech energy will be spread in the entire pseudo-cepstrum domain.

The enhancement process takes place at step 8. Because there is a noise built-up near the origin, Weiss *et al.* [14] initially decided to attenuate the samples near the origin. Obviously, the pseudo-cepstrum origin cannot be zeroed, otherwise on the return-trip the value of y_2' would become negative which is absurd since it corresponds to an absolute value. Later, other modification strategies were tried such as comb filtering of non pitch-harmonic components in the pseudo-cepstral domain [14] and subtraction of a pseudo-cepstrum noise estimate [4].

4. Analysis-synthesis structures for spectral modifications of speech signals

In light of the preceding chapter, we observe that many speech enhancement systems perform spectral decomposition of the noisy speech signal prior to the application of a noise reduction algorithm.

Various schemes have been proposed in the past to implement short-term spectral analysis, synthesis and modification of speech signals. In this chapter we briefly introduce the short-time Fourier transform which is the theoretical basis of many of these spectral analysis-synthesis structures. Then we present our analysis-synthesis resonator filterbank (ASRF) configuration.

4.1. Short-time Fourier analysis

4.1.1. Short-time Fourier transform definition

The Fourier transform plays a fundamental role in the analysis of signals. The success of the Fourier transform is due to the fact that it provides a unique representation of signals in terms of the eigen functions of linear time-invariant systems, namely the complex exponentials [28]. The Fourier transform of a discrete time signal $x(n)$ is given by

$$X(\omega) = \sum_{n=-\infty}^{\infty} x(n) \exp[-j\omega n] \tag{4.1}$$

and the inverse Fourier transform is defined by

$$x(n) = \frac{1}{2\pi} \int_{-\pi}^{\pi} X(\omega) \exp[j\omega n]\, d\omega. \tag{4.2}$$

These two equations provide a unique correspondence between $X(\omega)$ and $x(n)$. They both represent valid and complete representations of the signal.

For time-varying signals such as speech and music, this *static* standard Fourier representation has limitations. The entire signal $x(n)$ must be known to obtain its transform. Moreover, the total time waveform information is mapped into the frequency domain. Consequently, the Fourier transform does not have the appropriate mechanisms to handle time-varying systems and signals. Speech and music signals are slowly time-varying, their temporal and spectral properties can be assumed fixed only for short-time periods of 10 to 30 ms. Clearly, a more convenient

representation is obtained if we take into account both the time and frequency characteristics of time-varying signals. This transform is called the short-time Fourier transform (STFT). The STFT is defined by

$$X(n,\omega) = \sum_{m=-\infty}^{\infty} w(n-m)\, x(m) \exp[-j\omega m] \tag{4.3}$$

and the inverse STFT is

$$x(n) = \frac{1}{2\pi} \int_{-\pi}^{\pi} X(n,\omega) \exp[j\omega n]\, d\omega. \tag{4.4}$$

$w(n)$ is known as the analysis window and it usually has a finite duration. For speech signals, the window length is usually selected to have a duration of 10 to 30 ms. The STFT is a function of both a discrete time variable n and a continuous frequency variable ω.

There are two important interpretations to this transform. First, if n is assumed fixed, $X(n,\omega)$ can be seen as the standard Fourier transform of the windowed sequence $w(n-m)\, x(m)$. If ω is fixed, $X(n,\omega)$ is the convolution of the window $w(n)$ with the sequence $x(n) \exp[-j\omega n]$ or in other words, $X(n,\omega)$ is the output of a filter with impulse response $w(n)$ excited by a demodulated version of $x(n)$.

4.1.2. Sampling the transform in time and frequency

In order to use the STFT with digital computers, $X(n,\omega)$ is sampled in both time and frequency domains. If we let $\omega = \frac{2\pi m}{T}$ and $n=n'D$, m and n' now represent the discrete frequency and time indexes. Portnoff [28] has shown that under-sampled representations of a signal in terms of the discrete STFT in either the frequency or the time domain can nevertheless be perfectly recovered. These under-sampled representations are useful for coding applications, where the overall bit rate has to be lowered, but they are very sensitive to spectral modifications. Therefore they are not well suited to speech enhancement algorithms which perform non-linear filtering operations in the spectral domain.

In order to obtain a non-aliased sampled representation, we must apply the Nyquist theorem in both the time and frequency domain [29]. Since $w(n)$ has a lowpass characteristic and is of finite duration, it has two characteristic lengths in the Nyquist sense, one in the time domain and one in the frequency domain.

In the time domain $w(n)$ has a length of T samples. In the frequency domain the window is not totally bandlimited, therefore an attenuation criteria is used to define the bandwidth. In the case of the Hamming window a 42 dB attenuation criteria gives a length of $F=4/T$. The Nyquist theorem states that $X(n,m)$ must be sampled at a rate greater than or equal to twice its highest frequency. An equivalent

statement is that the density of time samples must be greater than or equal to the characteristic frequency length [29]. Thus, the time sampling period D becomes $D=1/F = T/4$. We see that it is not necessary to calculate the short Fourier transform for every time sample n. $1/D$ is called the frame advance rate.

We now apply the Nyquist theorem in the frequency domain. The continuous spectra can be replaced by a set of discrete frequencies given the density of frequency samples is greater or equal to the characteristic time length. Because the window has a duration and a characteristic length of T, we can sample at $fm = m/T$ for $m : 0...T$.

The discrete short-time Fourier transform can now be written as

$$X(n,m) = \sum_{k=0}^{T-1} w(nD-k)\, x(k) \exp[-j\, 2\pi km/T] \tag{4.5}$$

or

$$X(n,m) = \mathrm{DFT}\{\, w(nD-k)\, x(k)\}. \tag{4.6}$$

Here n is the time index, m the frequency index, D is the frame period and T is the window length. Equation 4.6 shows that the discrete STFT can be conveniently evaluated using discrete Fourier transforms applied on windowed signal sequences.

4.1.3. Overlap and add synthesis with the discrete STFT

Allen [29], proposed a method to recover $x(n)$ from its discrete STFT. If the window is band limited to $\frac{1}{2D}$, we can write this following identity

$$\sum_{n=-\infty}^{\infty} w(nD-k) = 1 \quad \text{all } k. \tag{4.7}$$

Observing that

$$x(k)\, w(nD-k) = \mathrm{DFT}^{-1}\{\, X(n,m)\,\} \tag{4.8}$$

and multiplying 4.7 by $x(k)$ we obtain the following synthesis rule

$$x(k) = \sum_{n=-\infty}^{\infty} \mathrm{DFT}^{-1}\{\, X(n,m)\,\}\ . \tag{4.9}$$

Therefore the signal $x(n)$ can be re-synthesized by summing overlapping sections of the inversely transformed STFT signal representation.

Many speech enhancement systems use versions of this method to spectrally analyze, modify and re-synthesize speech signals. For example in the spectral subtraction algorithm proposed by Boll [5], a Hanning window is applied to the incoming sequence. A DFT is performed on the windowed sequence to obtain the discrete STFT. The frame advance rate is taken to be half the window length. An inverse DFT is then applied on the STFT representation and overlapping sections of the output buffer are summed. In the absence of spectral modifications, the system reduces to an identity.

4.1.4. Synthesis with spectrum modifications

The analysis-synthesis system described above can be generalized to permit modifications of the short-term spectrum [29]. Let P_m be a fixed multiplicative spectral modification, then the synthesized signal can be written as

$$y(k) = \sum_{n=-\infty}^{\infty} \mathrm{DFT}^{-1} \{ P_m X(n,m) \} \tag{4.10}$$

Since P_m is a fixed multiplicative spectral modification, in the time domain we can write the equivalent convolution

$$y(k) = p(k) * x(k). \tag{4.11}$$

In order to obtain a non-aliased time version of $y(k)$, zeros are appended to the incoming sequence to satisfy the enlargement in the characteristic time-length caused by filtering.

The advantages of this method are that spectral modifications are permitted without using interpolation windows and the method is also computationally efficient since fast Fourier transforms can be used to obtain the discrete STFT representation.

4.2. Analysis-synthesis resonator filterbanks (ASRF)

4.2.1. Introduction

The STFT theory we just discussed is a generalization of many analysis-synthesis systems used in speech enhancement algorithms. As we have seen, Allen [29] proposed a block processing overlap and add technique using the discrete STFT which results in a symmetric uniformly spaced filter bank analysis structure. Others [30,31], have proposed filter bank realizations of the STFT which are comprised of contiguous filters in the analysis stage. These systems are particularly useful for

voice coding applications. But, because they result in a under-sampled spectral representation, they are sensitive to phase and amplitude modifications.

We propose an all-pass analysis-synthesis configuration which is designed to be robust to modifications of the short-time spectrum so that the reconstructed signal does not suffer from distortions introduced by the processing system. Like Allen [29], we use overlapping filters. This way, the signal reconstruction is less susceptible to modifications of the spectral envelope and phase. Unlike Allen, we do not force the filters to be linearly spaced, but allow them to be arbitrarily spaced along the frequency axis.

Our filterbanks exhibit the following properties: Given the N center frequencies, the frequency response of the m^{th} filter is unity at f_m and zero at all the other center frequencies f_j, and the sum of the squares of the frequency responses corresponds to an all-pass network (see fig. 4.1). Moreover, because we use a resonator filterbank for synthesis, we do not need to pad the incoming signal with zeros to avoid aliasing. The action of the second filter, apart from complementing the frequency response of the network to obtain an all pass network, is to filter all the out of bands frequencies which would create aliasing distortions in the reconstructed signal.

Unlike many speech processing systems, the signal is analyzed and synthesized sample by sample. This results in an over-sampled spectral representation which contributes to the robustness of the network to fast time-varying modifications of the spectral components. The filters we consider are of the frequency interpolation type [32], sometimes called resonator filterbanks. These banks which have the structure of figure 4.2 consist of first order complex ideal resonators for the central frequencies of the filters. An overall feedback is applied to stabilize the frequency response.

The transfer function from the input of the analysis filter to the m^{th} filter output is

$$H_m(z) = \frac{O_m(z)}{I(z)} = \frac{\dfrac{K_m p_m}{z - p_m}}{D(z)} \tag{4.12}$$

with

$$D(z) = 1 + \sum_{i=0}^{N-1} \frac{K_i p_i}{z - p_i} \tag{4.13}$$

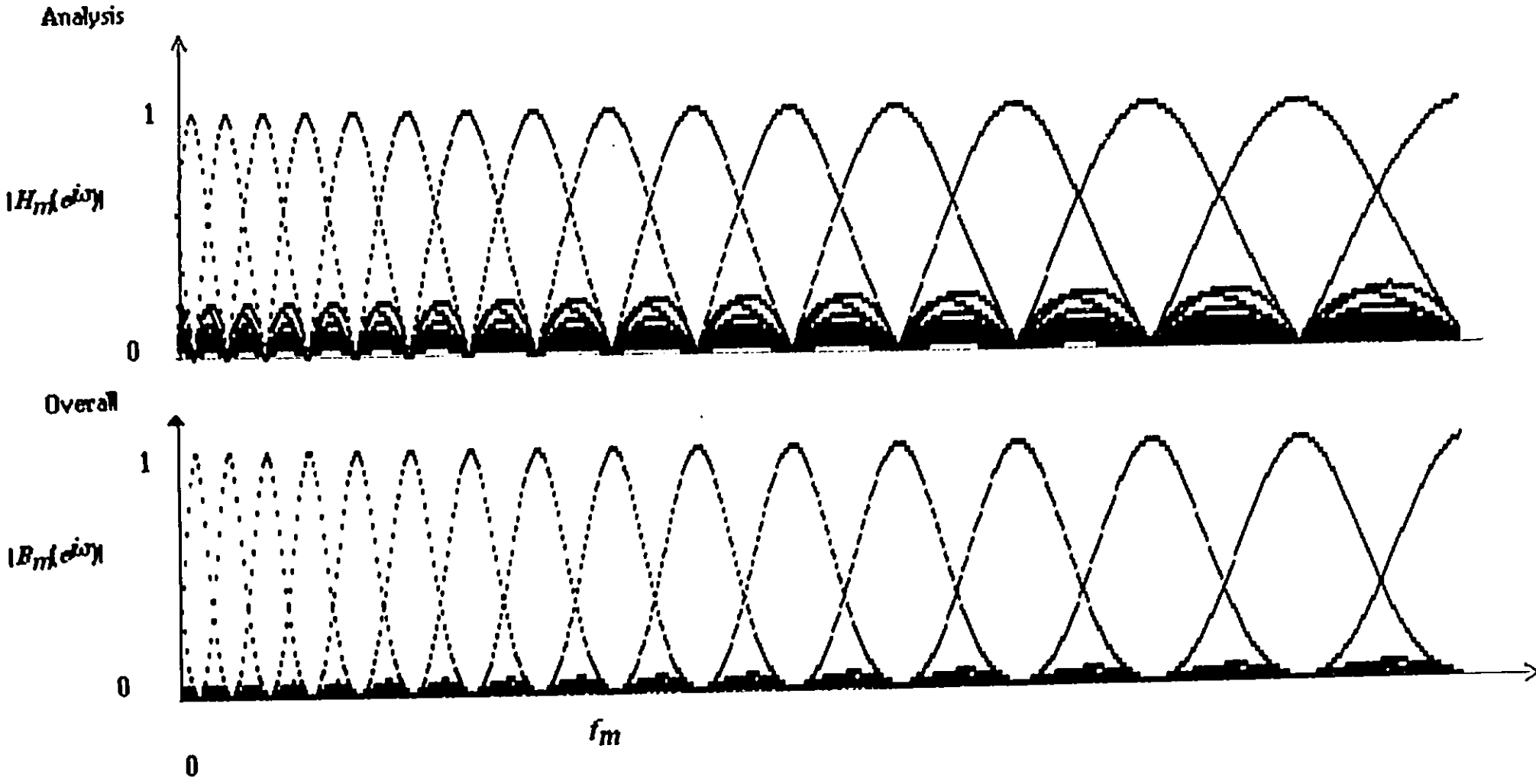

Figure 4.1: Transfer functions for ASRF configuration.
A filterbank with logarithmic spacing is depicted.

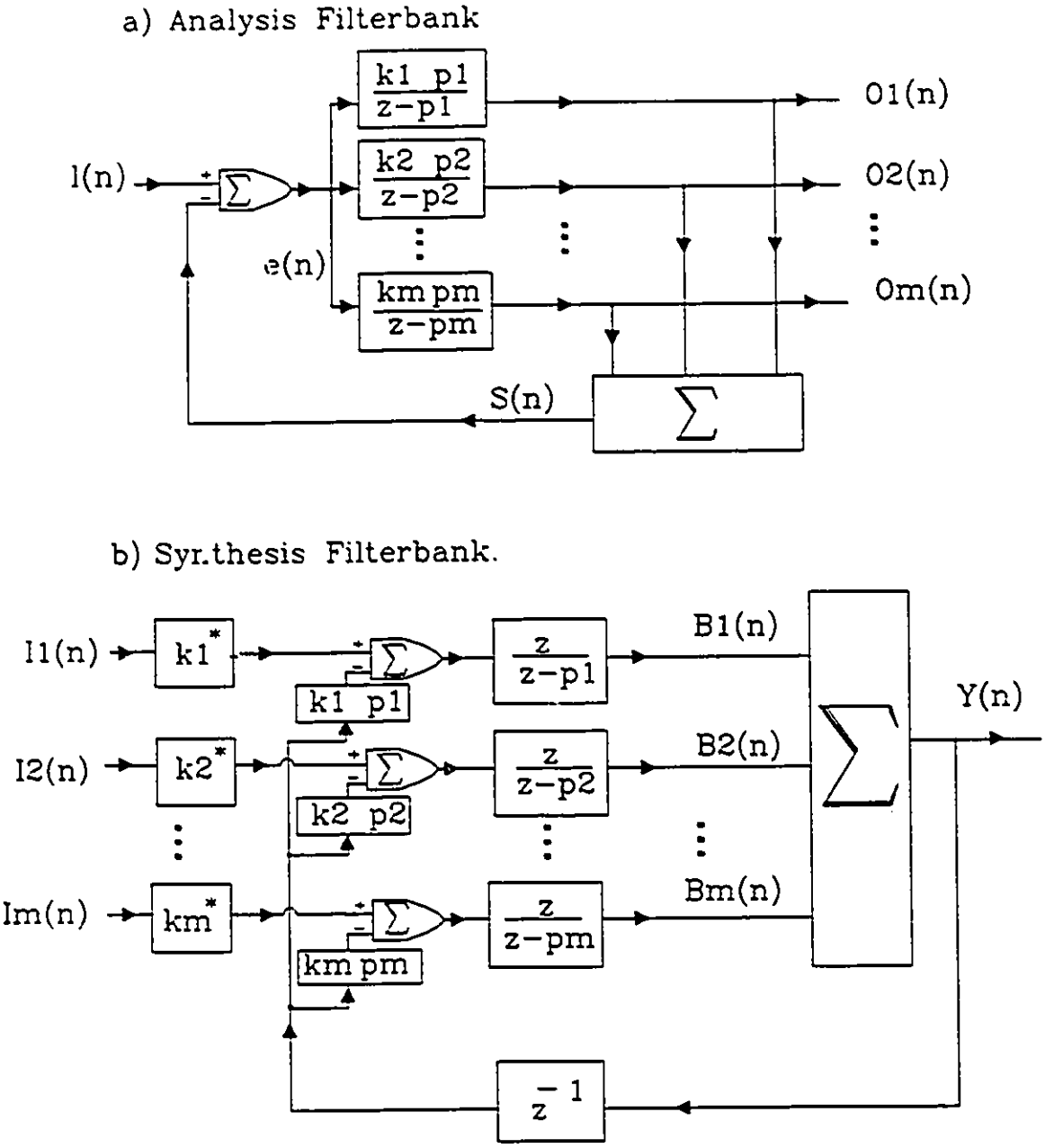

Figure 4.2: ASRF structures.

The synthesis filters have the following transfer function:

$$G_m(z) = \frac{B_m(z)}{I_m(z)} = \frac{\dfrac{K_m^* z}{z - p_m}}{D(z)}. \tag{4.14}$$

The overall transfer function becomes

$$T(z) = \sum_{m=0}^{M-1} G_m(z) H_m(z) = \frac{N(z)}{D(z)^2} \tag{4.15}$$

where

$$N(z) = \sum_{i=0}^{M-1} \frac{|K_i|^2 p_i z}{(z - p_i)^2}. \tag{4.16}$$

The numerator $N(z)$ has the property that if a zero is inside the unit circle at $z=z_m$, then there is a matching zero at the reciprocal conjugate position outside the unit circle, namely at $1/z_m^*$. Consequently, if we factor the numerator $N(z)$ into a product of two factors, one containing the zeros inside the unit circle and one containing the zeros outside, we can identify $D(z)$ and $D_*(z)$. If $N(z) = D(z)D_*(z)$, T(z) is all pass by definition. We call this structure, an analysis-synthesis resonator filterbank (ASRF).

4.2.2. Parameters selection and determination

To design an ASRF, we first select the M center frequencies ω_m. Because we are dealing with pure resonators, the poles are trivially found to be

$$p_m = e^{j\omega_m} \tag{4.17}$$

Then the absolute value of the weights $|k_m|$ are selected given that they must sum to unity for stability reasons and that they are proportional to the desired filter bandwidths. In the case of a linear-spacing realization $|k_m| = 1/M$. For constant Q logarithmically-spaced filterbanks, we set the $|k_m|$ proportional to the center frequencies.

To complete the filterbank realization, we must calculate the phase angle of K_m [32]. The steps needed to find the phase angle of the coefficients are the following:

1. Given $|k_m|$ and p_m evaluate $N(z)$;

2. Find the factors of $N(z)$ corresponding to the zeros inside the unit circle, this is $D(z)$;

3. Use equation (4.13) to identify the coefficients K_m by partial fraction expansion.

The computationally expensive and ticklish portion is step 2 since it necessitates the utilization of root finding routines for high order polynomials. For the case of linear spacing, the K_ms are simply equal to $1/M$.

4.2.3. ASRF implementation and complexity

Because they are digital filters, the ASRF structures are trivially implemented with digital signal processing chips or computers. Even though arbitrary spacing of the center frequencies is permitted, the resonator filterbanks exhibit some symmetry properties which make them computationally efficient.

4.2.3.1. Analysis filterbank recursive equations

We recall that the transfer function of the analysis filterbank is

$$H_m(z) = \frac{\frac{K_m p_m}{z - p_m}}{D(z)} \tag{4.18}$$

From figure 4.2 a),

$$H_m(z) = \frac{O_m(z)}{I(z)}, \tag{4.19}$$

defining an error signal $e(z)$,

$$e(z) = I(z) - \sum_m O_m(z) = I(z) - \sum_{i=0}^{N-1} \frac{K_i p_i}{z - p_i} e(z) \tag{4.20}$$

we find

$$e(z) = \frac{I(z)}{D(z)}. \tag{4.21}$$

From figure 4.2 we can write

$$O_m(z) = e(z) \frac{K_m p_m}{z - p_m} = e(z) \frac{z^{-1} K_m p_m}{1 - p_m z^{-1}}. \tag{4.22}$$

Taking the inverse Z transform, we get the time domain equation

$$O_m(n) = k_m p_m e(n\text{-}1) + p_m O_m(n\text{-}1), \tag{4.23}$$

defining a temporary variable

$$S(n) = \sum_{i=0}^{M-1} O_m(n) \tag{4.24}$$

the error is conveniently expressed as

$$e(n) = I(n) - S(n). \tag{4.25}$$

Equations 4.23 to 4.25 are the sought recursive equations. They are evaluated at every sample time n. The state variables which must be saved at each iteration are $O_m(n\text{-}1)$ and $e(n\text{-}1)$. Even though the poles p_m and the weights k_m exist for both positive and negative frequencies, the output of the negative frequency filters are the complex conjugates of their positive counterparts. Therefore, we only need to compute the equations for half the filters. This symmetry property forces $S(n)$ and $e(n)$ to be real numbers which greatly simplifies the calculations. When implemented on a DSP56000 VLSI processor, this structure only necessitates 7 cpu cycles per filter per sample.

4.2.3.2. Synthesis filterbank recursive equations

The transfer function from the *spectral* inputs $I_m(z)$ to the overall output of the structure $Y(z)$ is given by

$$Y(z) = \sum_m \frac{\frac{K_m \cdot z}{z - p_m}}{D(z)} I_m(z) \tag{4.26}$$

or

$$Y(z) = \sum_m \frac{K_m * z}{z - p_m} I_m(z) - \sum_m \frac{K_m p_m}{z - p_m} Y(z), \tag{4.27}$$

multiplying (4.27) by z^{-1}/z^{-1}, we obtain a realizable equation

$$Y(z) = \sum_m \frac{K_m^* \; I_m(z) - K_m p_m z^{-1} \; Y(z)}{1 - z^{-1} p_m} \tag{4.28}$$

or using a temporary variable $B_m(z)$

$$Y(z) = \sum_m B_m(z) . \tag{4.29}$$

To calculate the filter output, it suffices to add each branch contribution $B_m(z)$. These branches are directed by the same type of time recursive equations as the analysis structure. From 4.28 and 4.29, we obtain the recursive equations needed to calculate the filter output

$$B_m(n) = K_m^* \; I_m(n) + S_m(n-1) \tag{4.30}$$

$$Y(n) = \sum_m B_m(n) \tag{4.31}$$

$$S_m(n) = p_m \; (B_m(n) - K_m \; Y(n)). \tag{4.32}$$

The structure of the synthesis filterbank is drawn in figure 4.2 b).

Once again we use the symmetry properties of this structure to simplify the calculations. Since the input signals $I_m(z)$ are complex conjugates, the output $Y(n)$ is real. Consequently, only half the filters need to be calculated. This filterbank roughly requires the same number of operations as the analysis filterbank.

4.2.4. Parallel between the ASRF and the STFT

The STFT OLA synthesis method we previously discussed can be interpreted in terms of a filterbank operation using DFTs. Actually, when the STFT time index is not decimated (which is equivalent to advancing the window one sample at a time), and a rectangular window is used, the resulting overall transfer function is identical to the ASRF with linear spacing of the center frequencies. This is a direct consequence of the fact that the DFT can be interpreted as the realization of a resonator filterbank with linear spacing. Therefore, the only difference between a linearly-spaced ASRF and the overlap and add discrete STFT method using a rectangular window is the time decimation effect of the window advance rate [39].

4.3. Non-linear processing with the ASRF

4.3.1. Experiments

In order to test the signal reconstruction property of the ASRF, we used the structure of figure 4.2 to perform an experiment in which the filterbank parameters, namely the number of filters and frequency spacing, were varied. In this particular experiment, speech signals are processed by the analysis filterbank and the output magnitudes are calculated at every sample time, then the M weakest of N signals are set to zero before the application of the synthesis filterbank. We believe this experiment is representative of the type of operations a typical speech enhancement algorithm would perform. Its interest is twofold, first it verifies the capability of the structure to absorb fast time-varying gains applied on the spectral envelope, secondly it verifies the capability of the structure to reconstruct the signal when only few spectral components are retained.

We also did the same experiment with a DFT overlap and add (OLA) method which is typical of many analysis-synthesis schemes used in speech enhancement systems and similar to Allen's implementation of the STFT [5,27]. This OLA implementation does not append zeros to the incoming sequence prior to the application of the DFT.

More specifically, we compared speech processed by a mel-spaced ASRF filterbank (34 filters), a sixth-octave logarithmically spaced ASRF filterbank (40 filters) and a linearly spaced ASRF filterbank with the same number of filters. The filter coefficients are given in appendix 1. In addition, a 64 point-DFT overlap and add (OLA 50 %) method with Hanning window was compared to the linearly spaced ASRF filterbank made-up of 64 filters.

4.3.2. Results

In all cases, when 20% of the analysis filterbank outputs are kept for synthesis, the reconstructed speech signal remains very intelligible. The DFT overlap and add method introduces click and pop sounds when less than 50 percent of the spectral components are retained. The ASRF configurations, both logarithmically and linearly spaced, do not suffer from this problem, however, the logarithmically and mel-spaced filterbanks introduce a faint reverb quality to the reconstructed speech. This is not objectionable for speech enhancement applications. For a given computational complexity, these logarithmically spaced structures will allow the implementation of more filters within the formants frequency region where it is important to have good frequency resolution to discriminate between speech and noise spectral components during voicing. Table 4.1 summarizes the informal listening test results.

Type:	ASRF Linear 64 filters	STFT (OLA) 50% overlap	ASRF Mel-spacing 34 filters	ASRF log. 40 filters 6^{th} oct.	ASRF linear 40 filters
Number of spectral components for good perceptual quality:	12 filters	12 filters	7 filters	8 filters	8 filters
Artifacts:	none	pops and clicks	faint reverb	faint reverb	none

Table 4.1.

Figure 4.3 shows a section of the original signal. The x axis on these plots represents the sample time in integers and the y axis is the amplitude of the signal in converted digital integers. The same section processed by the ASRF filterbank where only 5 components are kept at every sample time is shown in figure 4.4. Note that the origins do not correspond to the same signal point from one graph to the others. We see that the ASRF attempts to follow the original curve as closely as it can with only five spectral components. We also observe that the original fast moving transitions are smoothed by the synthesis filterbank. This smoothing effect becomes even more evident when we compare with the waveform of figure 4.5 which was produced by the DFT OLA method. In this case the re-synthesized signal contains more abrupt discontinuities than the original signal. This resulted in pop and click sounds during the informal listening test. One of these pulse-noise like distortions can distinctly be seen in figure 4.5 at position $n=16$.

Figure 4.6 is a spectrogram section of the original utterance which contains both voiced and unvoiced sounds. Another spectrogram of the same signal section is shown in figure 4.7. In this case, the signal was processed by the ASRF configuration where we kept only the 5 maximum spectral magnitude components at every sample time. For vowel sounds, the filterbank tracks the F1 and F2 formants. As expected for unvoiced sounds, because the vocal tract does not exhibit as well defined resonances, the 5 maximum spectral components wander in the high frequency portion of the spectrum.

4.3.3. Relevance of the ASRF configuration for speech enhancement

Many of the speech enhancement algorithms we presented in chapter 3 estimate the spectral amplitude of the speech signal by multiplying the measured spectral amplitude by a gain function. Conceptually, this gain function should change rapidly especially at transition boundaries between noise only sections and phone segments to properly estimates the speech signal. In the precedent paragraph, we have demonstrated that the overlap and add (OLA) technique used in many speech

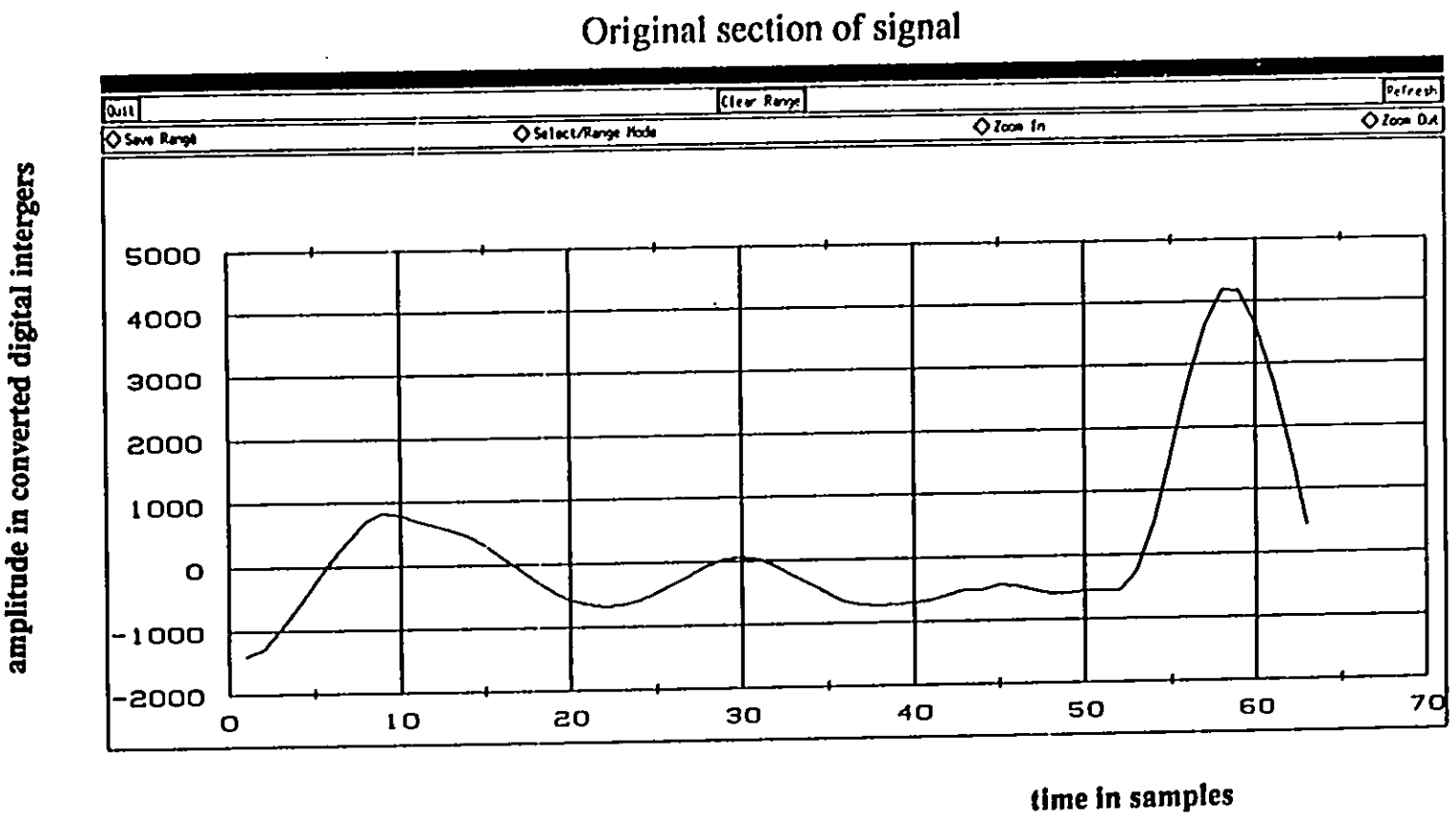
Original section of signal
Quit
Clear Range
Refresh
Save Range
Select/Range Mode
Zoom In
Zoom Out
5000
4000
3000
2000
1000
0
-1000
-2000
0
10
20
30
40
50
60
70
amplitude in converted digital intergers
time in samples

Figure 4.3

Section processed by ASRF

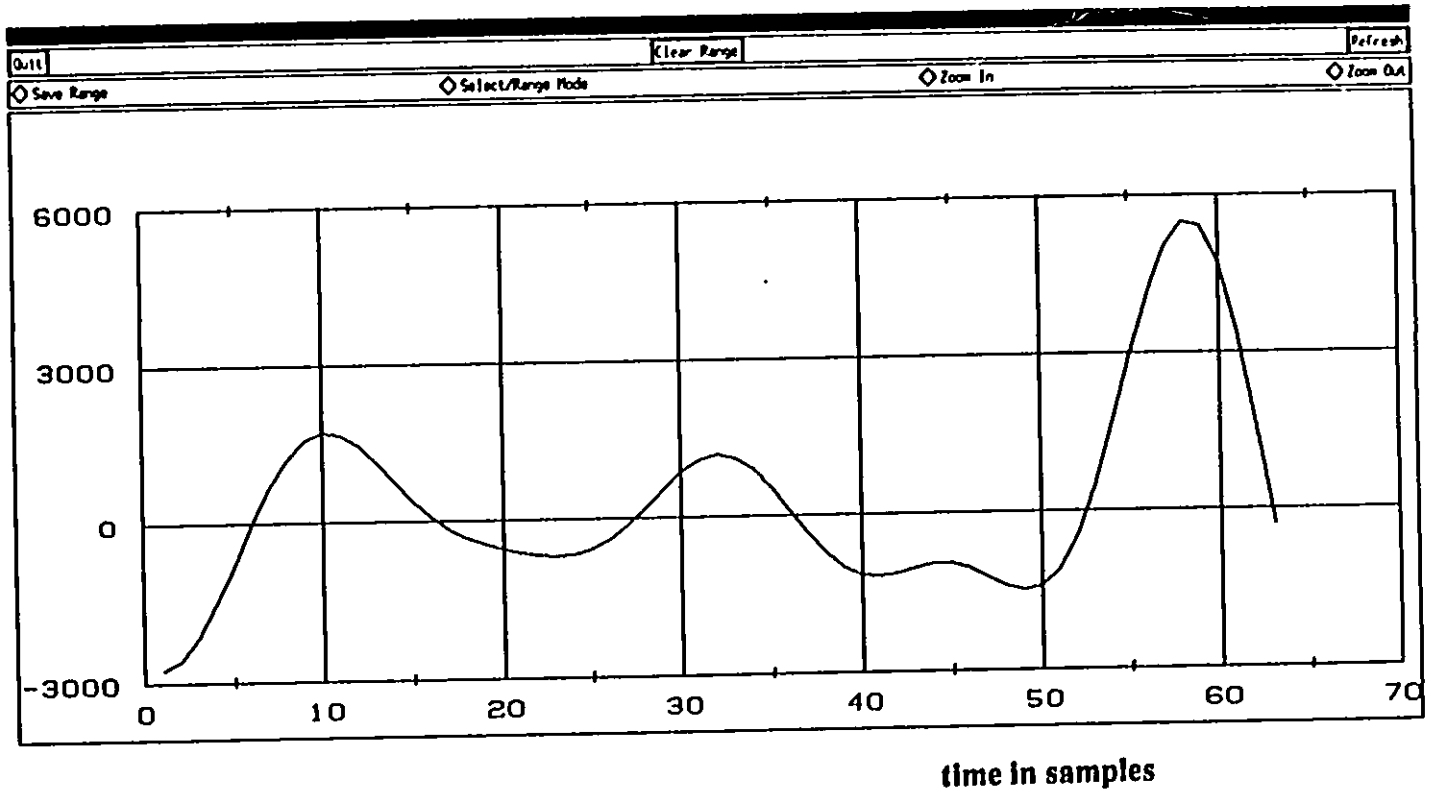

Figure 4.4

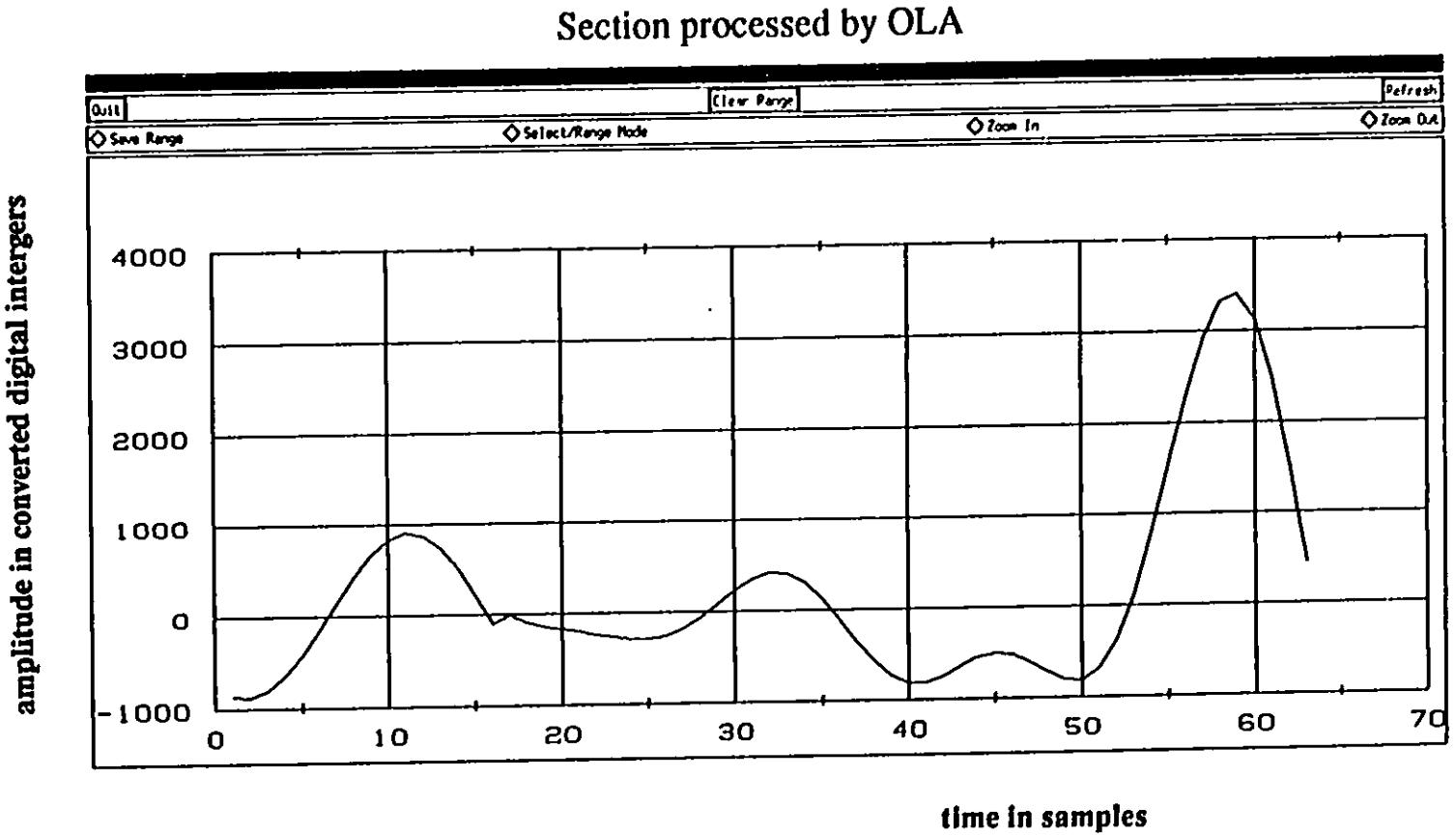
Section processed by OLA
Quit
Clear Range
Refresh
Save Range
Select/Range Mode
Zoom In
Zoom Out
4000
3000
2000
1000
0
-1000
0
10
20
30
40
50
60
70
amplitude in converted digital intergers
time in samples

Figure 4.5

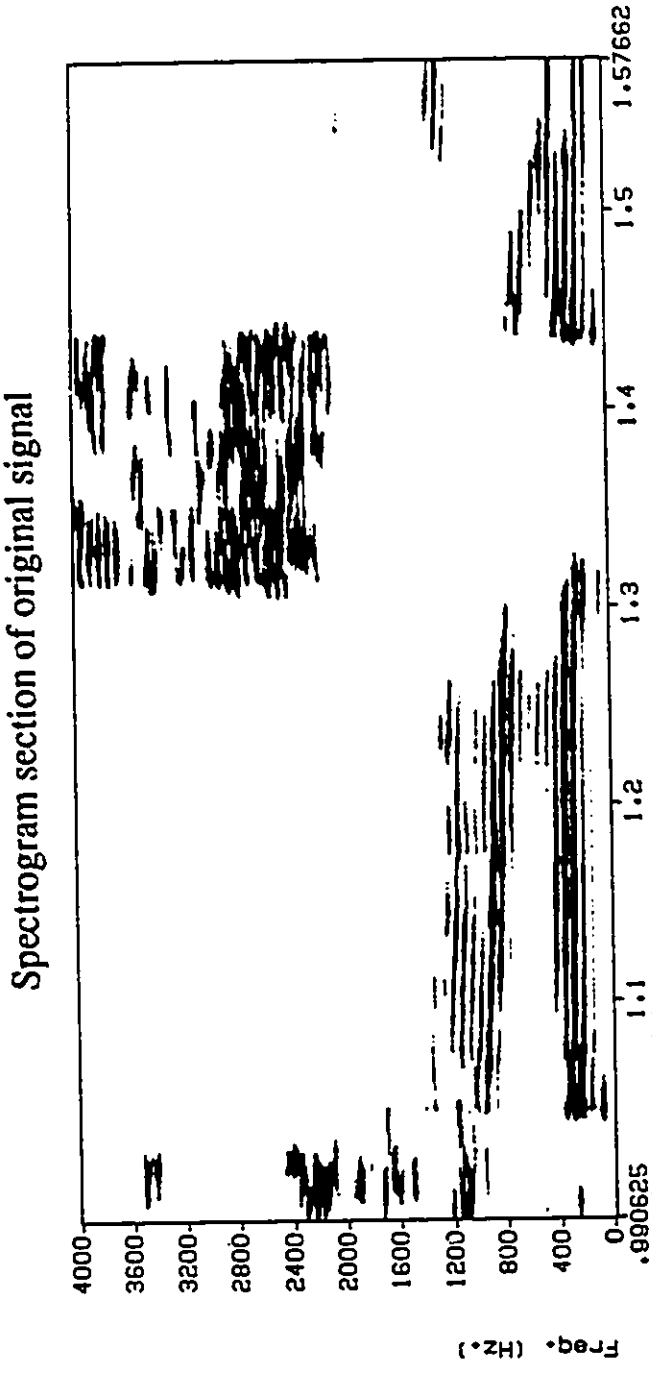
Spectrogram section of original signal
Freq. (Hz.)
4000
3600
3200
2800
2400
2000
1600
1200
800
400
0
.990625
1.1
1.2
1.3
1.4
1.5
1.57662
Time (seconds)

Figure 4.6

Spectrogram section of 5 maximum magnitude

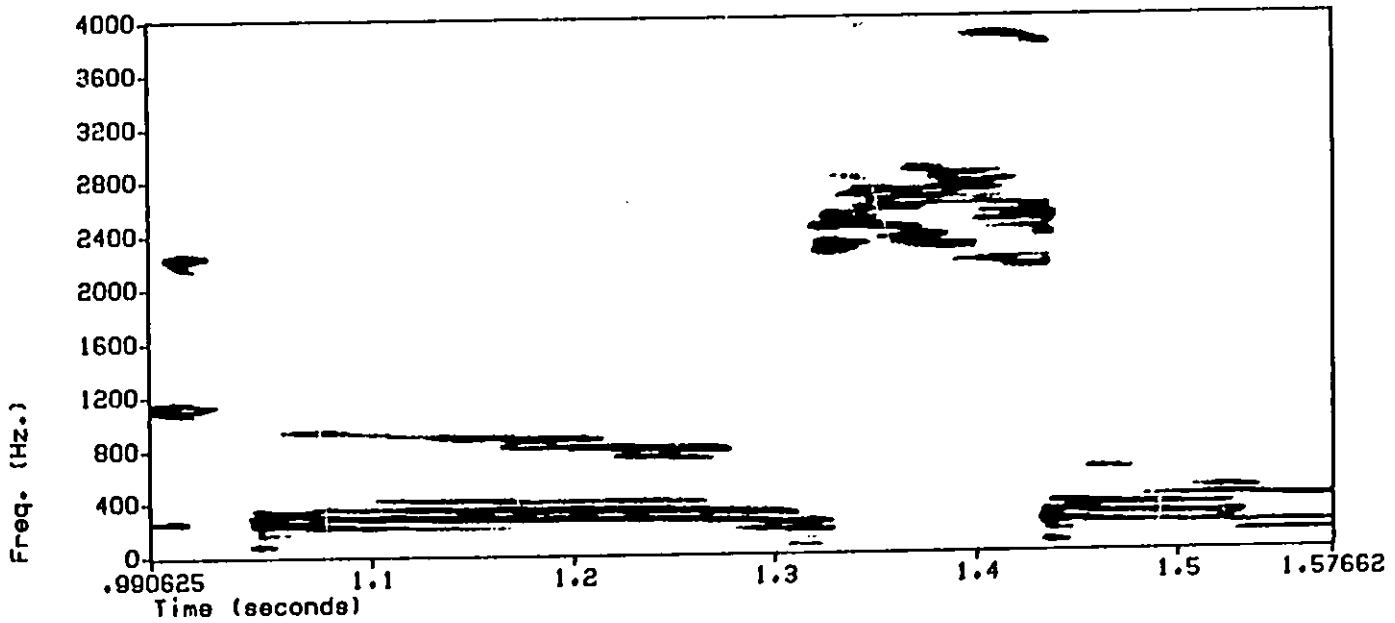

Figure 4.7

enhancement algorithms can introduce artifacts to the processed speech if fast time-varying modifications are allowed in the spectral domain.

One way to inhibit this undesirable effect is to smooth the transitions of the gain function. McAulay *et al.* [22] followed this approach and only used an analysis filterbank with their soft-decision noise suppression curves. Allen [29] suggested appending zeros to the incoming signal sequence to take into account the increased characteristic length due to the filtering effect of the spectral modifications when the STFT is implemented by DFT block processing. One of the drawbacks with this method is that one must evaluate in advance the actual increase in the characteristic length.

The ASRF structure offers an alternative. The synthesis filterbank eliminates fast varying components which would otherwise create distortions in the re-synthesized signal [33]. Moreover, this smoothing effect is inherently performed by the all-pass structure and is not imposed by an arbitrary time constant such as in [22]. In addition, the center frequencies of the resonators that make up the all-pass ASRF can be arbitrarily spaced along the frequency axis. This way, logarithmically spaced or mel-spaced filterbank structures, which may be more suitable for certain speech and music applications, can be implemented at no extra computational cost.

5. A speech enhancement algorithm using the ASRF

In this chapter we introduce our noise reduction system based on the analysis-synthesis resonator filterbank configuration. We describe the various algorithmic elements of the speech enhancement system and the software implementation.

5.1. Block diagram

A block diagram of our proposed speech enhancement system is shown in figure 5.1. The system is similar to the canonical representation for frequency domain noise reduction technique of figure 3.4. In our system, the spectral decomposition is done by the analysis resonator filterbank. The analysis filters produce a complex spectral vector at every sample time. A delay line can be introduced to enable a better estimation of the spectral amplitude. The complex outputs are then transformed into polar form. The measured magnitudes are multiplied by time-varying frequency dependent gain functions to obtain the speech spectral magnitude estimates. In our present implementation the measured spectral phases constitute the estimated phase components. The signal is then reconstructed by the synthesis filterbank applied on the estimated speech spectral vector.

5.2. Spectral amplitude estimation

In chapter 3 we presented some spectral amplitude estimation schemes based on statistical parametric representations for both speech and noise spectral components. These statistical models are used to derive optimum spectral estimators by minimizing arbitrary error criteria which are more or less perceptually meaningful [22,24]. Other researchers [6,27] use variants of the noise gate. Initially, we experimented with many of these spectral estimators, namely the Wiener gain, the power spectral subtraction, the minimum mean-square error short-time spectral amplitude estimator (MMSE STSA) [24], the soft-decision noise suppression filter [22], the parametric Wiener gain [24], the expander function [6] and the non-linear spectrum processor [27]. Because these techniques all possess control parameters which are used to compromise between noise attenuation and speech distortion, they can be set to offer comparable perceptual performance. We retained the parametric Wiener gain in our final implementation because it offers more flexibility.

5.2.1. Parametric Wiener gain

We recollect that the parametric Wiener gain function is given by [26]

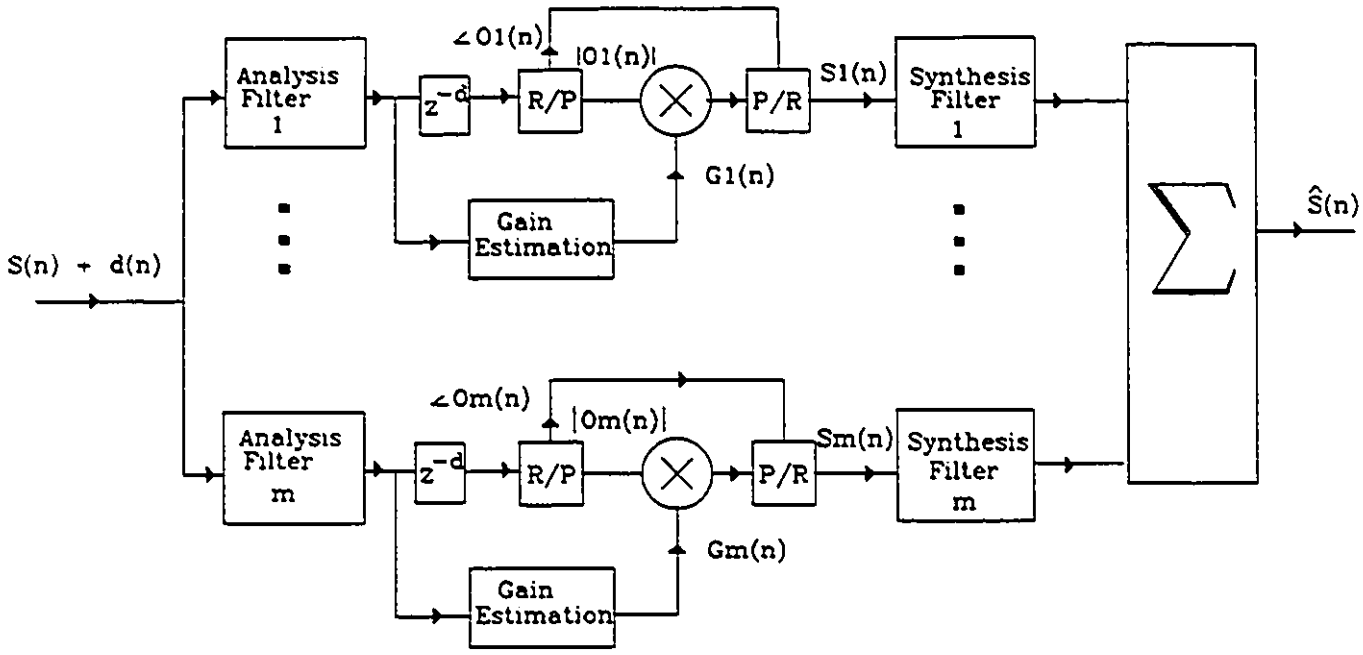

Figure 5.1: Speech enhancement with the ASRF.

$$G(\omega) = \left[\frac{\xi(\omega)}{\xi(\omega) + T}\right]^r \tag{5.1}$$

with

$$\xi(\omega) = \frac{S_{ss}(\omega)}{S_{nn}(\omega)}. \tag{5.2}$$

$S_{ss}(\omega)$ and $S_{nn}(\omega)$ are respectively the speech and noise power spectral densities. This gain function can generate different attenuation curves whose characteristics depend upon three parameters: the *a priori* SNR ξ, the exponent value r and the threshold T.

5.2.2. Exponent

The exponent imposes the curve shape. If the exponent is equal to 1, the classic Wiener gain function attenuation curve depicted in figure 3.7 is produced. If the exponent is large, a noise gate is realized. One of the advantage of the noise gate behavior of this gain function is that if the SNR is high the spectral components are not modified.

5.2.3. Power Spectral estimation

Obviously, the power spectral densities for both the speech and the noise processes are not known. By definition the power spectral density of a wide-sense stationary process is given by

$$S(\omega) = \int_0^\infty R(\tau)\, e^{-j\omega\tau}\, d\tau. \tag{5.3}$$

In our work, we use a simple estimate of the power spectrum based upon the time-averaged periodogram. The periodogram is given by [23]

$$S_T(\omega) = \frac{1}{2T} \mid X_T(\omega) \mid^2 \tag{5.4}$$

where $X_T(\omega)$ is the Fourier Transform of a time segment $(0...T)$ of the signal. In order to reduce the variance of this biased (in the statistical sense) estimate, the average of the squared spectral amplitudes at the output of the analysis filters are taken yielding a smoothed estimate of the periodogram. In the context of our spectral decomposition method, this averaged-periodogram becomes

$$S'(n,m) = E[|O_m(n)|^2] \tag{5.5}$$

where $O_m(n)$ is the output of the analysis filter m at time n. Note that because speech and noise processes are not truly stationary, the power spectra have to be constantly re-evaluated.

5.2.4. A priori SNR estimation

The most critical factor for the performance of this spectral estimation strategy proved to be the computation of the *a priori* SNR. The noise power spectral density is calculated using 5.5 during time segments labeled as noise only. This yields $S_{nn}(\omega)$. $S_{ss}(\omega)$ is not explicitly calculated since the parametric Wiener gain only needs the *a priori* SNR, hence we use the following recursive average of the SNR :

$$\xi_n(\omega) = (1-\lambda)(\gamma_n(\omega) - \xi_{n-1}(\omega)) + \xi_{n-1}(\omega) \qquad 0 < \lambda < 1 \tag{5.6}$$

$$\gamma_n(\omega) = \frac{V_n(\omega)^2}{S_{nn}(\omega)} - 1 . \tag{5.7}$$

Here, λ acts as the learning rate and V is the measured spectral amplitude. $\gamma_n(\omega)$ is interpreted as the measured signal to noise ratio assuming the speech and noise signals are present in the spectral component $V_n(\omega)$. This equation is a computationally efficient way to get the averaged-periodogram estimate $S'_{ss}(\omega)$ when $S'_{nn}(\omega)$ is known.

In the context of our analysis-synthesis structure, this *a priori* SNR calculation method reduced considerably the coloration of the remnant noise. For the case where the learning rate λ is zero, the exponent r equals 0.5, our method is equivalent to the power spectral subtraction and, as expected, the remnant noise has a strong musical quality. The factor $1/\log(\lambda)$ is a time constant which should have a comparable value to the average phone duration in order to get a good estimate for the a *priori* SNR. Perceptually, when λ is too close to 1, the enhanced speech has a reverberant quality whereas when λ is too small, the enhanced speech is corrupted by an annoying colored remnant noise. This recursive average equation is derived by minimizing the following error criteria

$$E_n = \sum_0^{\infty} \lambda^r (X_{n-r} - \mu(X_n))^2 \tag{5.8}$$

which gives

$$\mu(X_n) = (1-\lambda)\, X_n + \lambda\, \mu(X_{n-1}) \qquad 0 < \lambda < 1. \tag{5.9}$$

Substituting the measured SNR for X_n and the *a priori* SNR for the estimated mean value $\mu(X_n)$, we obtain equation 5.6.

5.2.5. Threshold

The threshold T in (5.1) can be interpreted as a multiplicative factor which produces an overestimate of the noise spectral power density in a given channel. In fact rewriting 5.1 in its basic form we observe that the noise power spectral density becomes $T\ S_{nn}(\omega)$:

$$G(\omega) = \left[\frac{S_{ss}(\omega)}{S_{ss}(\omega) + T\, S_{nn}(\omega)}\right]^r. \tag{5.10}$$

Because the noise spectral estimates are obtained by averaging successive squared spectral amplitudes of the output of the analysis filter during non speech activity, the variance (statistical bias) of the estimates is a function of the duration of these time intervals. T attempts to compensate for the fact that under certain conditions, the estimate of the noise spectrum is an under-estimate of the noise power spectral density.

5.2.6. Remnant noise whitener

Several noise reduction systems produce a colored remnant noise artifact when they process speech degraded by broadband noise at SNRs lower than 10 dB. Even though these algorithms reduce the noise power and thereby increase the SNR, the coloration of the remnant noise is often more annoying than the original noise. This is the case notably for the power spectral subtraction method. Here is the nature of the problem. By definition the spectrum of white noise is flat. But when we look at short-term spectra of noise processes, the waveform displays valleys and peaks. These maxima and minima randomly change frequency locations from one time frame to the other. When we attenuate the spectral amplitudes of noisy speech based on a smoothed noise estimate, some noise peaks will remain and more valleys will be created. When the remaining noise peaks are wide, remnant white noise is regenerated by the spectral synthesizer. When these spectral peaks are narrow, a tonal (musical) quality is given to the remnant noise.

In order to render the remnant noise perceptually uncolored even at very low SNRs, we included in our algorithm a scheme previously proposed by Berouti *et al.* [21]. It consists of preventing the estimated spectral components from descending

below a certain bound. This way, in a given frame, or equivalently at a certain sampling time, residual amplitude spectral peaks below a certain threshold are set to a constant value termed the spectral floor. This precludes spectral excursions, giving the artifact noise a white perceptual quality. This constant value is conveniently expressed as a fraction of the channel noise power

$$\text{constant}(\omega) = \beta\, S_{nn}(\omega). \tag{5.11}$$

The spectral estimation strategy is modified by adding few more steps:

$$\text{Let } |D_m(n)| = G_m(n) \quad |O_m(n)| \tag{5.12}$$

$$\text{if } |D_m(n)|^2 > \beta\, S_{nn}(\omega) \text{ then } I_m(n) = |D_m(n)| e^{j\theta} \tag{5.13}$$

$$\text{if } |D_m(n)|^2 <= \beta\, S_{nn}(\omega) \text{ then } I_m(n) = \sqrt{\beta\ S_{nn}(\omega)}\, e^{j\theta}. \tag{5.14}$$

$O_m(n)$ is the output produced by the analysis filter m at time n and $I_m(n)$ is the speech spectral estimate presented to the synthesis filter m.

Berouti *et al.* only experimented with computer-generated white noise. Moreover the spectral noise estimate they used was averaged in both time and frequency. Therefore the multiplicative factor was forced to produce equal constant values for all frequencies yielding the desired whitening effect.

In our case we have experimented with different types of broadband noise which do not always exhibit a flat spectrum. Therefore, in order to get a spectral floor which is constant for all frequencies but still characteristic of the overall spectra, we average one third of the smallest noise spectral estimates. Thus, even if strong periodic components are present among the broadband noise, the regenerated remnant noise will have the desired white perceptual quality.

5.2.7. Typical values

Table 5.1 shows typical values of the parametric Wiener gain function and the whitening threshold determined by informal listening tests. They do not represent the best values since the perceptual optima proved to be signal and user dependent. This is why we chose to include ranges instead of specific values in the table.

SNR	r	T	λ	Whitener β
> = 20 dB	2 to 4	1 to 2	0. 99 to 0. 997	0
15 dB	2 to 3	1 to 2	0. 99 to 0. 997	0. 0001
10 dB	2 to 2. 5	1 to 2	0. 99 to 0. 997	0. 001 to 0. 0001
5 dB	1 to 3	1 to 3	0. 99 to 0. 997	0. 003 to 0. 001
0 dB	1 to 3	1 to 3	0. 99 to 0. 997	0. 008 to 0. 003
<= −5 dB	1 to 3	1 to 3	0. 99 to 0. 997	0. 01 to 0. 008

Table 5.1: Typical values used in speech enhancement experiments with the parametric Wiener gain.

5.3. Non causal signal estimation

The estimation of the SNR necessarily involves some delay. If the speech enhancement application can tolerate some extra processing delay, the spectral signal could be delayed before the application of the gain function. The estimation of the signal at a given time would depend on the past values of the observations plus some future values. In figure 5.1 the delays are located just before the rectangular-polar transform. Therefore, the estimation of the *a priori* SNR is a function of the past and future values of the signal.

Because the signal is now known in advance, the gain rise at the beginning of uttered words may be faster. Hence, a better estimation of leading, low energy speech sounds is possible while keeping the coloration of the remnant noise to a minimum. This approach is appropriate for speech recognition applications where the entire utterance is used to perform the classification of the acoustic signal. However, this *look ahead capability* is not well suited for some communication applications where the signal must be delivered with minimum delay.

5.4. Noise estimation

We obtain the noise spectral density estimates by processing noise segments of a duration of about 1 second by the analysis filterbank and averaging the squared magnitude of the outputs. The noise samples used to get the estimates are obviously not used to create the noisy speech utterances. The underlying idea is that in a real-time implementation, a noise/speech discriminator would be used to detect noise-only activities in order to estimate the noise power at the output of the analysis filterbank. This approach is exploited in the SEU [15] which uses a discriminator based on pitch detection. Interestingly, their algorithm termed NUPITCH [15] could

be applied directly to the output of our analysis filterbank because it necessitates fine resolution spectrograms to determine speech presence or absence.

5.5. Software implementation

The algorithm was implemented in C on a Masscomp 6655 computer. However, because of the portability provided by the C language, we compiled and ran versions of the algorithm on a SUN Sparc station and on a 386 PC. The program *gdnt_noise.c* is called with a noise file and produces the noise spectral density estimates. The program *gdnt_d.c* is called with the noisy speech file, the noise estimate file and all the variable parameters needed to evaluate the spectral gain function and then produces the enhanced speech file.

The software programs were not optimized for speed but rather written in an structured manner to allow easy experimentations with the algorithms.

5.5.1. Software validation

In the absence of modifications of the spectral components, logarithmically spaced filterbanks are all-pass structures and linearly spaced filterbanks are linear phase systems. Hence, to test the software functionality it suffices to give a zeroed noise estimate to the program. The produced file should be identical minus the trivial filterbank delays to the original speech file. When complicated spectral gain functions such as the MMSE STSA were implemented, we independently tested these routines and compared them with the attenuation curves provided by the authors.

6. Speech enhancement experiments

In this chapter we report on speech enhancement experiments we conducted with the ASRF and other speech enhancement algorithms which we also implemented in C. First we discuss the experimental conditions and then we present the results provided by the different algorithms.

6.1. Speech material

Two sentences from the DARPA TIMIT database [34] were selected for the experiment; sentence labeled *si1757* spoken by a male and sentence *sx36* spoken by a female. These utterances were chosen to be representative of typical English phrases in the sense that they contain different phoneme types. We feel it would be unfair to test the algorithms on utterances comprising almost only voiced sounds such as in the phrase: "She was away a year ago". Moreover, the spoken sentences are uttered naturally, with no emphasis on consonants. Consequently, the informal listening test will give a good idea of the type of performance the systems would offer under real conditions.

The male sentence is: " His sarcasm was followed by a stupid grin of his thick mouth and bad teeth."; and the sentence uttered by the woman speaker is: "Only the most accomplished artists obtain popularity. " The digitized speech data are sampled at 16 kHz and coded with a linear quantizer on 16 bit words. The maximum amplitude of the selected utterances only use 14 bits. Consequently, noise can be added to the signal down to approximately -10 dB SNR without saturation of the dynamic range.

6.2. Noise material

Almost all the speech enhancement algorithms described in the literature were tested using computer generated noise. This is not objectionable *per se* if the noise is generated by a proper algorithm which ensures adequate randomness of the noise samples. However, many real life noise environments are just partially modeled by computer generated Gaussian or uniform distributed random numbers.

In our experiments, we used noise samples from the RSG-10 database [17]. This database was recently made available to the public by the NATO Research Study Group on Speech Processing (RSG-10) in the same CD ROM format as the TIMIT database. One of the RSG -10's interests is the evaluation of automatic speech recognizers and speech communication channels in military situations such as high noise environments. They have put together a database of noise recordings from various noise sources such as jet-airplane, helicopters, wheel carriers, tanks, thermal

noise generators, HF radio noise, and car noise. We have selected white noise and HF noise samples from this database because they are typical of many communication channels.

The NATO database was sampled at 20 kHz on 16 bit words. Consequently the noise samples cannot be added directly to the speech material since their sampling frequency differ. Hence, we resampled the noise database to make it compatible with the TIMIT database.

6.2.1. Re-sampling the noise files

In order to re-sample the noise database, a few more C programs had to be written. We followed the approach suggested in [10]: To change the sampling rate by a fractional factor such as M/N, the signal is first interpolated by inserting M-1 zeros after each sample then a lowpass filter with cutoff frequency π/M or π/N (whichever is the smallest) is applied to the interpolated signal. Finally the signal is decimated by keeping only every N^{th} sample.

In our specific case, the 20 kHz noise signal was interpolated to 80 kHz by padding 3 zeros after each samples. Then a 200 tap FIR lowpass filter with cut-off frequency of 8 Khz was applied on the data stream. The purpose of this filter is twofold, first it filters out the frequencies over 8 kHz present in the original noise database but which would create aliasing distortions in the 16 kHz format; secondly, it interpolates the signal so that the interpolating zeros are replaced by smoothed samples. Finally, we decimated the signal to 16 Khz by keeping one sample every 5 samples.

The resulting down-sampled noise files were compared with the originals by doing a spectral analysis comparison with spectrograms and a listening test. In addition, a simple signal composed of two sinusoidal signals at 9 kHz and 1 kHz (sampling frequency 20 kHz) was down-sampled to 16 Khz. As expected the resulting file contained only one tone at 1Khz.

6.3. Putting together the noisy speech material

The noisy corpus is made by adding the speech samples to the appropriately weighted noise samples to obtain the desired SNRs. There is no *de facto* standard to determine the SNR ratio because there are a few problems with the calculation of SNRs for speech signals. Vowels have more energy than consonants and the speech signal has a duty cycle which is dependent on the speaking style. Therefore, the speech signal is not stationary and the use of the definition of the signal to noise ratio will lead to values which are signal dependent and often inconvenient to compare with other noisy utterances.

Other methods have been devised to take the speech variability into account but the method based upon the SNR definition is more often encountered in the

literature and is far easier to implement. Assuming the same time length for both signals, the SNR definition is given by

$$\mathrm{SNR} = 10\,Log\left[\frac{\sum_{0}^{N-1} s(t)^2}{\sum_{0}^{N-1} n(t)^2}\right]. \qquad (6.1)$$

We added the noise samples to the speech signals to produce corrupted speech files from -5 dB to 30 dB SNR in increment of 5 dB using the SNR definition to get the weights which multiply the noise samples.

6.4. Experiments

The ASRF speech enhancement algorithm was then used to enhance the noisy speech. In order to compare the results, we also implemented in C, the INTEL [14] method which is used in a commercial speech enhancement system [15], the MMSE STSA proposed by Malah and Ephraim [24] and a generalized spectral subtraction method [21].

We performed informal listening tests to adjust the 4 different algorithms for optimality. These algorithms have diverse variable parameters. We will briefly describe how we made their selection.

6.4.1. ASRF

The ASRF speech enhancement system is comprised of linearly spaced filterbanks made up of 128 filters. The other parameters are set-up according to table 5.1.

6.4.2. INTEL

Subtraction in the pseudo-cepstral domain was performed with different subtraction factors ranging from 0 to 1.4. The perceptual optima were found to be between 0.7 and 0.8. The other parameters were set-up according to the values suggested in [14]. Therefore the root compression factor is 0.5 for all FFT bins except for the origin bin which is compressed by a factor of 0.75. The input signal is multiplied by a Bartlett (triangular) window, the frame advance rate is 50% of the window length, and FFTs of 1024 points are performed in the spectral and pseudo-cepstral domain. The window length is therefore 64 ms.

6.4.3. Generalized power subtraction

The method proposed by Berouti *et al.* [21] (equation 3.7) was also implemented. A 256 STFT OLA (50% Hanning window) method was used for spectral decomposition in order to get the same spectral resolution as the ASRF. The overestimate factor was varied from 1 to 4. The exponent was fixed to 2 and the whitening factor ranged from 0.0001 to 0.01.

6.4.4. MMSE STSA

The MMSE STSA we described in chapter 3 was also implemented in C using the same short-time Fourier transform implementation as the generalized power subtraction method. Actually, we tried all the different algorithms put forth in [24]. Interestingly, when we compared these methods in an informal listening test, we drew the same conclusions as Ephraim and Malah. This exercise really assisted us in understanding the speech enhancement jargon used to describe the perceptual characteristics of the remnant noise sometimes produced by these intricate algorithms. Ultimately, the best algorithm suggested in their paper was retained for comparison purpose: It is the MMSE STSA with speech presence uncertainty seconded by the decision directed *a priori* SNR calculation method. q_k was set to 0.2 and λ was adjusted to 0.99.

6.5. Results

6.5.1. Informal listening test

The processed material was compared under these subjective quality criteria: speech intelligibility, amount of noise reduction, inconspicuousness of remnant noise and overall signal quality (audible distortion of speech signal). We found that all the algorithms we tried can be optimized to produce excellent enhanced speech quality with no perceptual remnant noise down to a level of about 20 dB SNR. At these moderate SNRs we found that the ASRF produced enhanced speech of a slight better quality. We believe this is due to the combination of the reconstruction property of the structure, the utilization of the smoothing technique which provides the look ahead capability and the noise gate behavior of the parametric Wiener gain at high SNRs.

Between 20 dB and 5 dB, the algorithms try to compromise between noise reduction and speech distortion. When total noise attenuation is desired, unvoiced sounds, especially fricatives, disappear, thus severely decreasing intelligibility. When the intelligibility of the signal must be maintained, noise attenuation is attainable at a lower level and a background remnant noise is produced. The major difference between the algorithms is the quality of this remnant noise.

We selected these particular algorithms for comparison because they are known to produce perceptual white remnant noise down to a SNR of about 5 dB. Nevertheless, we found that as the SNR decreases from 20 to 5 dB, the variance of these remnant noises increases leading to an accrued coloration. However, these artifacts are far from being as strong or annoying as the colored noise produced by the classic power subtraction method.

6.5.2. SNR improvement

The SNR of the enhanced speech file can be evaluated by calculating the energy of the whole utterance and the energy of a known noise only segment. As mentioned earlier, these values have to be compared with caution because of the method by which the SNR is calculated. Nevertheless, these estimated values give a good idea of the filtering effect of the speech enhancement system for a given acceptable perceptual performance. Table 6.1 summarizes the results for the male speaker utterance degraded by white noise.

SNR before enhancement	SNR after enhancement	net SNR increase
5 dB	26 dB	21 dB
10 dB	35 dB	25 dB
15 dB	44 dB	29 dB

Table 6.1: Typical SNR improvements provided by the ASRF speech enhancement system.

6.5.3. Illustrative results

We have also included plots of processed and unprocessed speech signals to better demonstrate the capabilities of our speech enhancement algorithm. Figure 6.1 is a time plot of the whole utterance *si1757* with no noise added. The same speech signal degraded by white noise at 5 dB SNR is shown in figure 6.2. We observe that because speech is non-stationary, some segments of the signal are totally buried in the noise. In figure 6.3 we show the time signal resulting from the application of the ASRF algorithm on the noisy speech. Graphically, the enhanced speech is very similar to the uncorrupted speech signal.

However, if we zoom in on the same signals, the waveforms will reveal interesting details. In figure 6.4 we show a zoomed portion of the undegraded signal. The noisy signal is plotted in figure 6.5. The same segments processed by the ASRF and the MMSE STSA algorithm are shown respectively in figure 6.6 and 6.7. Here we observe that the plosive sound found around position n=6100 in figure 6.4 is zeroed in both 6.6 and 6.7. Both algorithms believed it was noise. This particular low energy short duration sound is a realization of the phoneme /t/ of the word *stupid*. This

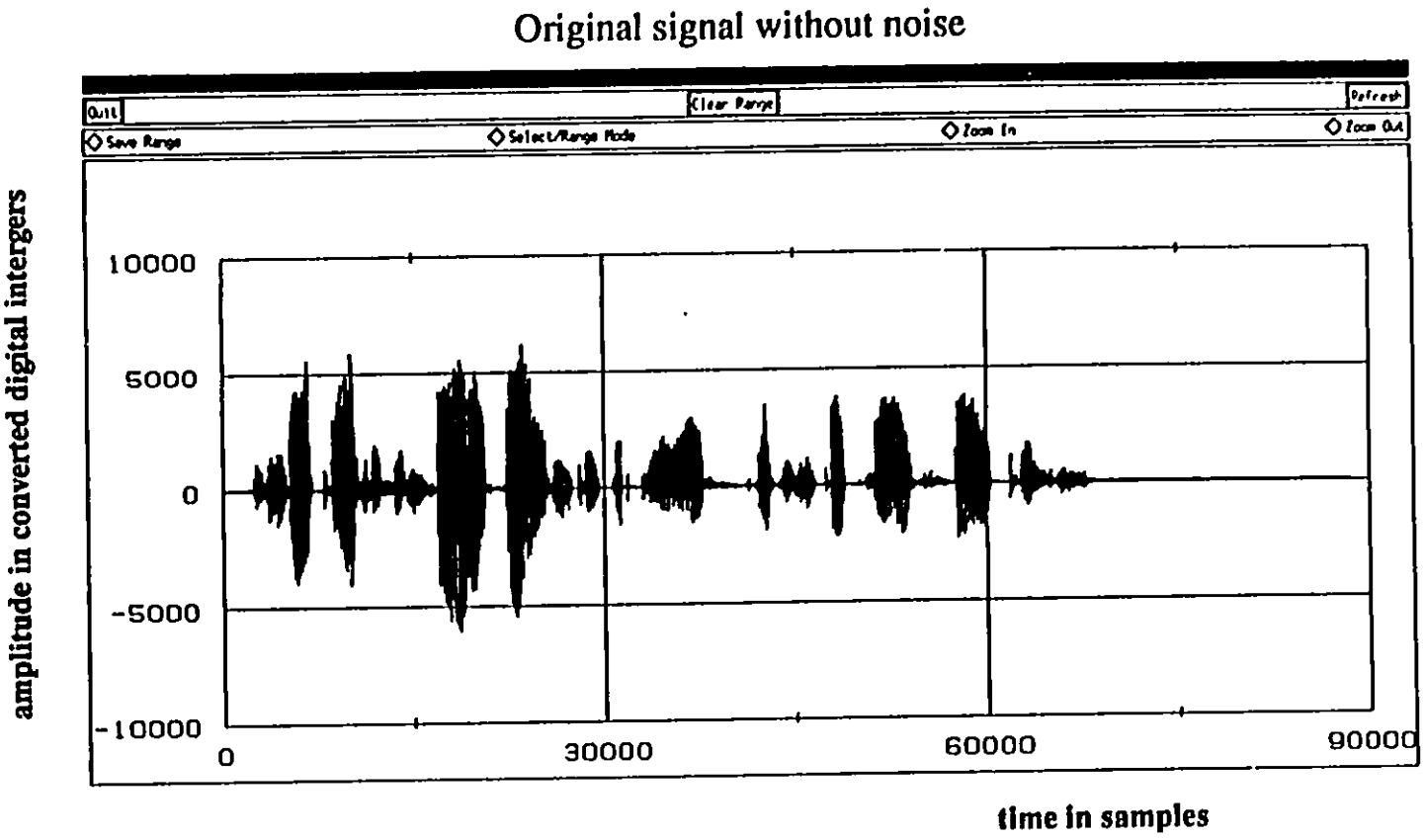
Original signal without noise
Quit
Clear Range
Refresh
Save Range
Select/Range Mode
Zoom In
Zoom Out
10000
5000
0
-5000
-10000
0
30000
60000
90000
amplitude in converted digital intergers
time in samples

Figure 6.1

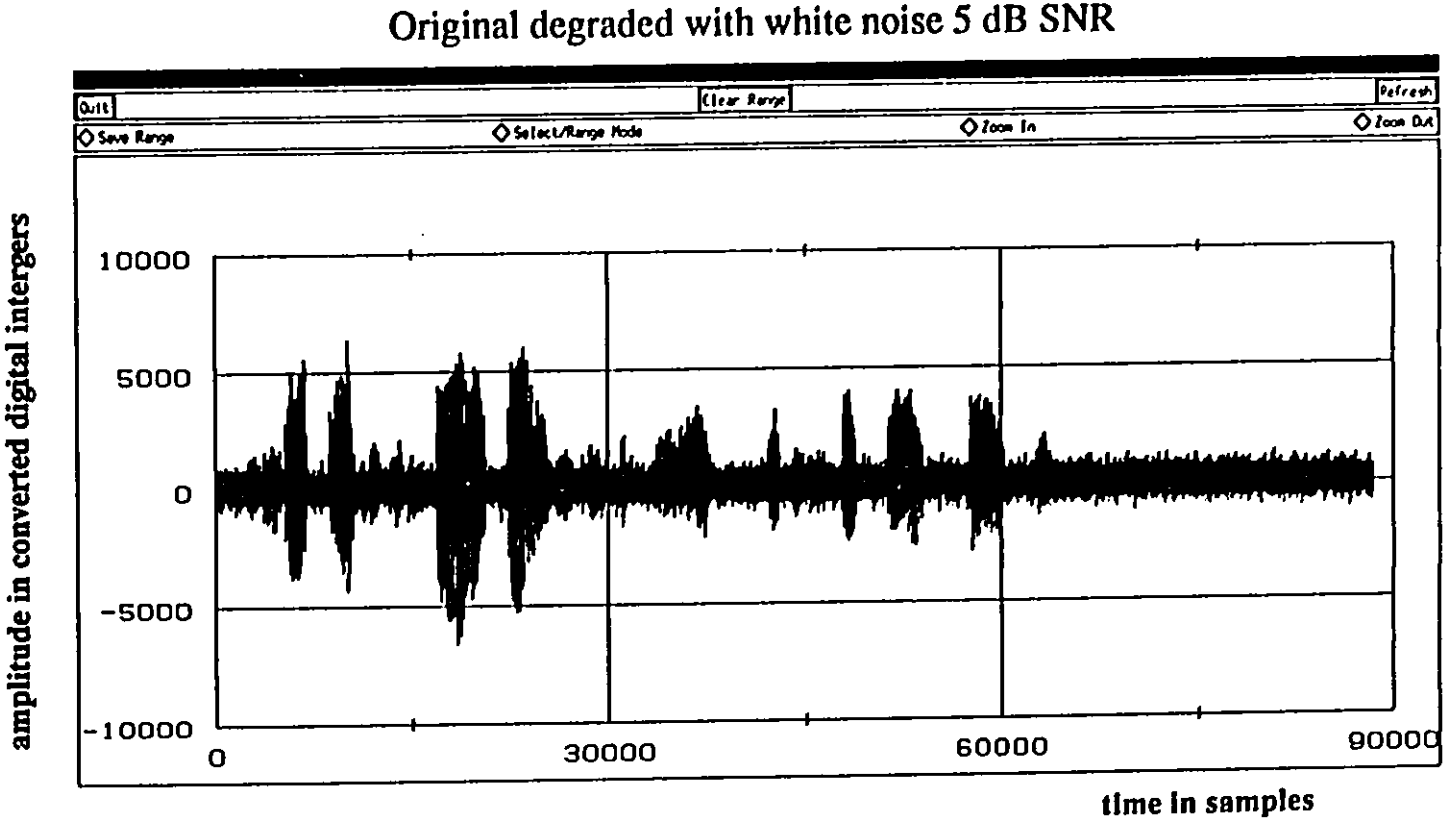
Original degraded with white noise 5 dB SNR
Quit
Clear Range
Refresh
Save Range
Select/Range Mode
Zoom In
Zoom Out
10000
5000
0
-5000
-10000
0
30000
60000
90000
amplitude in converted digital intergers
time in samples

Figure 6.2

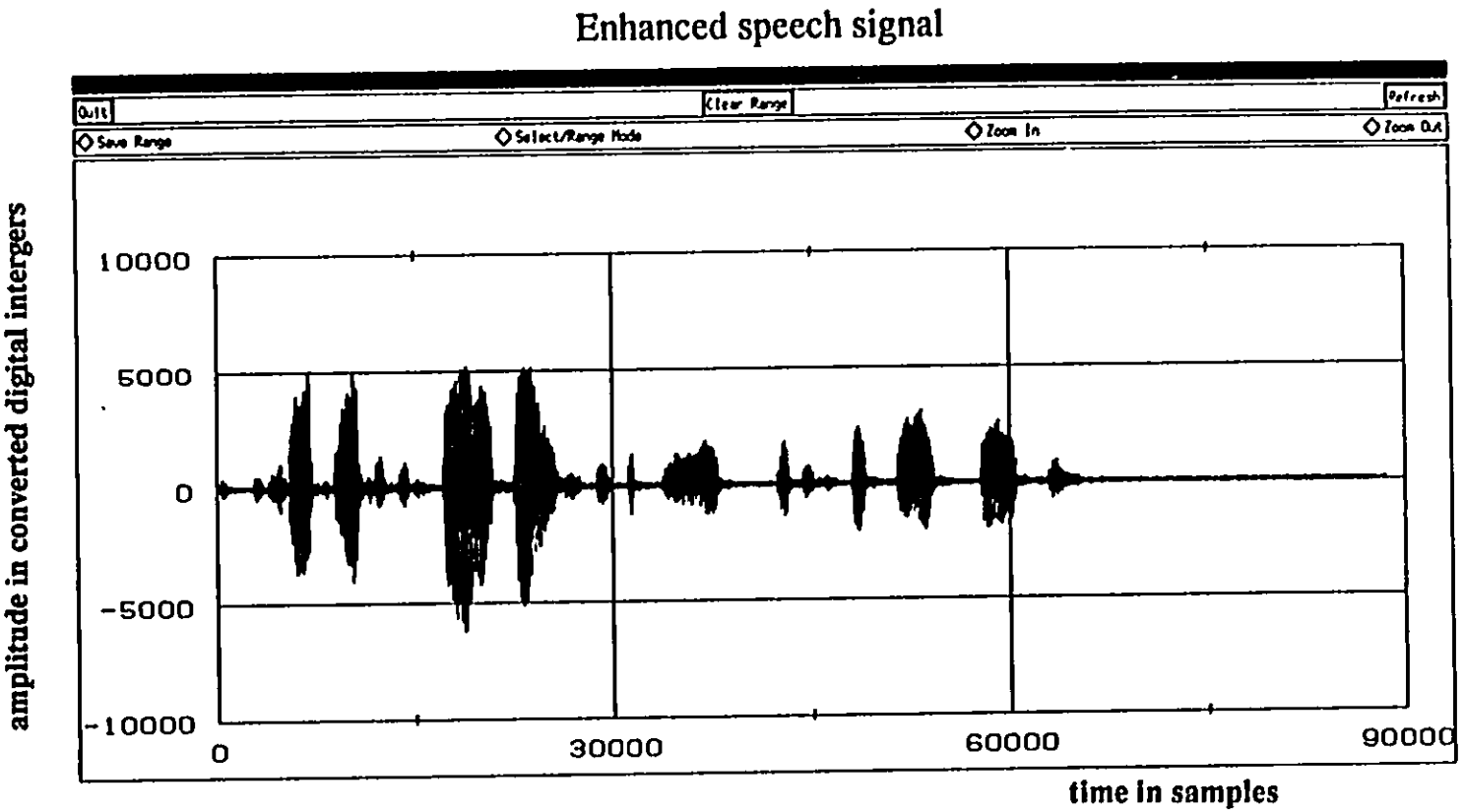
Enhanced speech signal
Quit
Clear Range
Refresh
Save Range
Select/Range Mode
Zoom In
Zoom Out
10000
5000
0
-5000
-10000
0
30000
60000
90000
amplitude in converted digital intergers
time in samples

Figure 6.3

Original without noise
Quit
Save Range
Select Range Mode
Clear Range
Zoom In
Zoom Out
Refresh
10000
5000
0
-5000
-10000
0
3000
6000
9000
amplitude in converted digital integers
time in samples

Figure 6.4

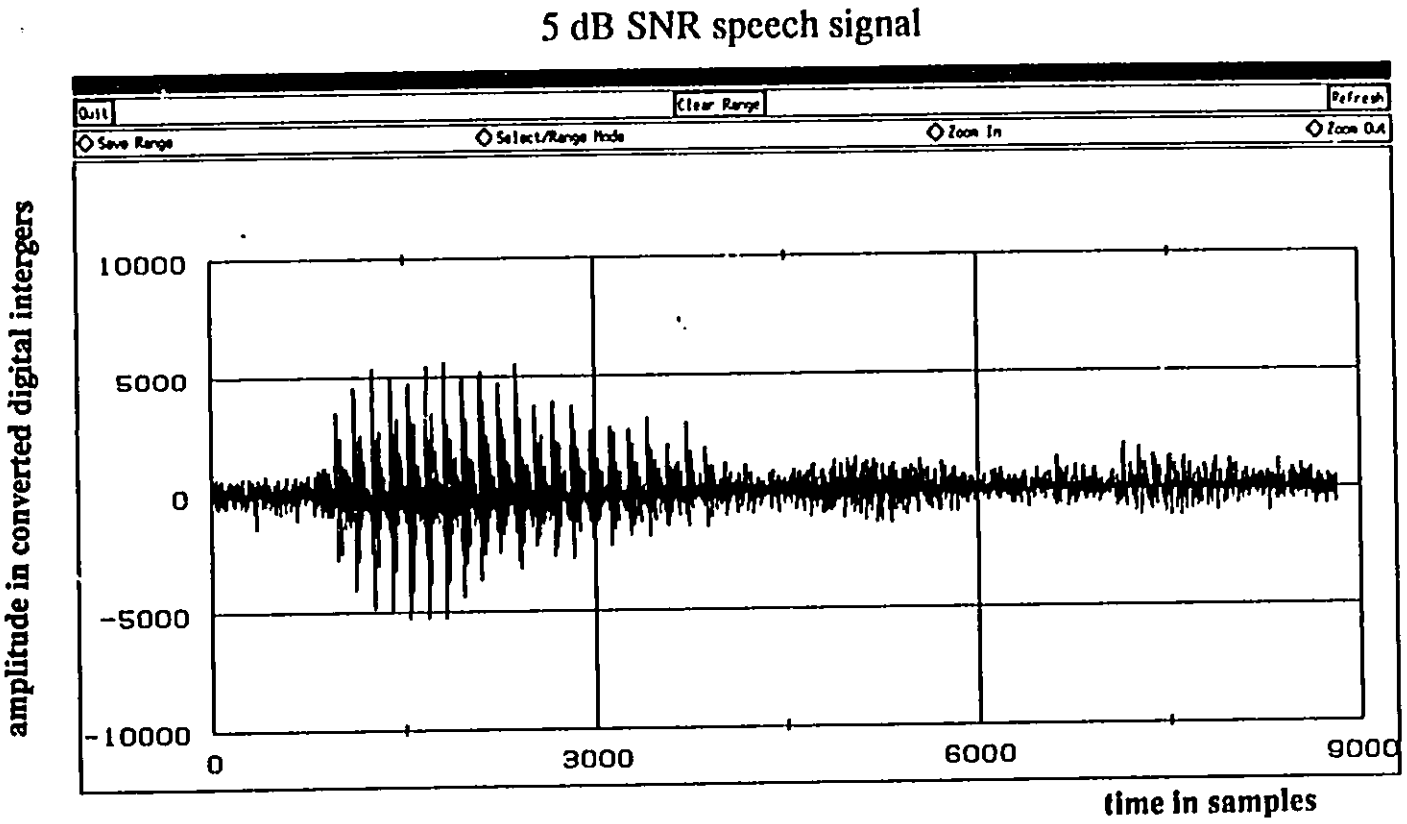
5 dB SNR speech signal
Quit
Clear Range
Refresh
Save Range
Select/Range Mode
Zoom In
Zoom Out
amplitude in converted digital intergers
10000
5000
0
-5000
-10000
0
3000
6000
9000
time in samples

Figure 6.5

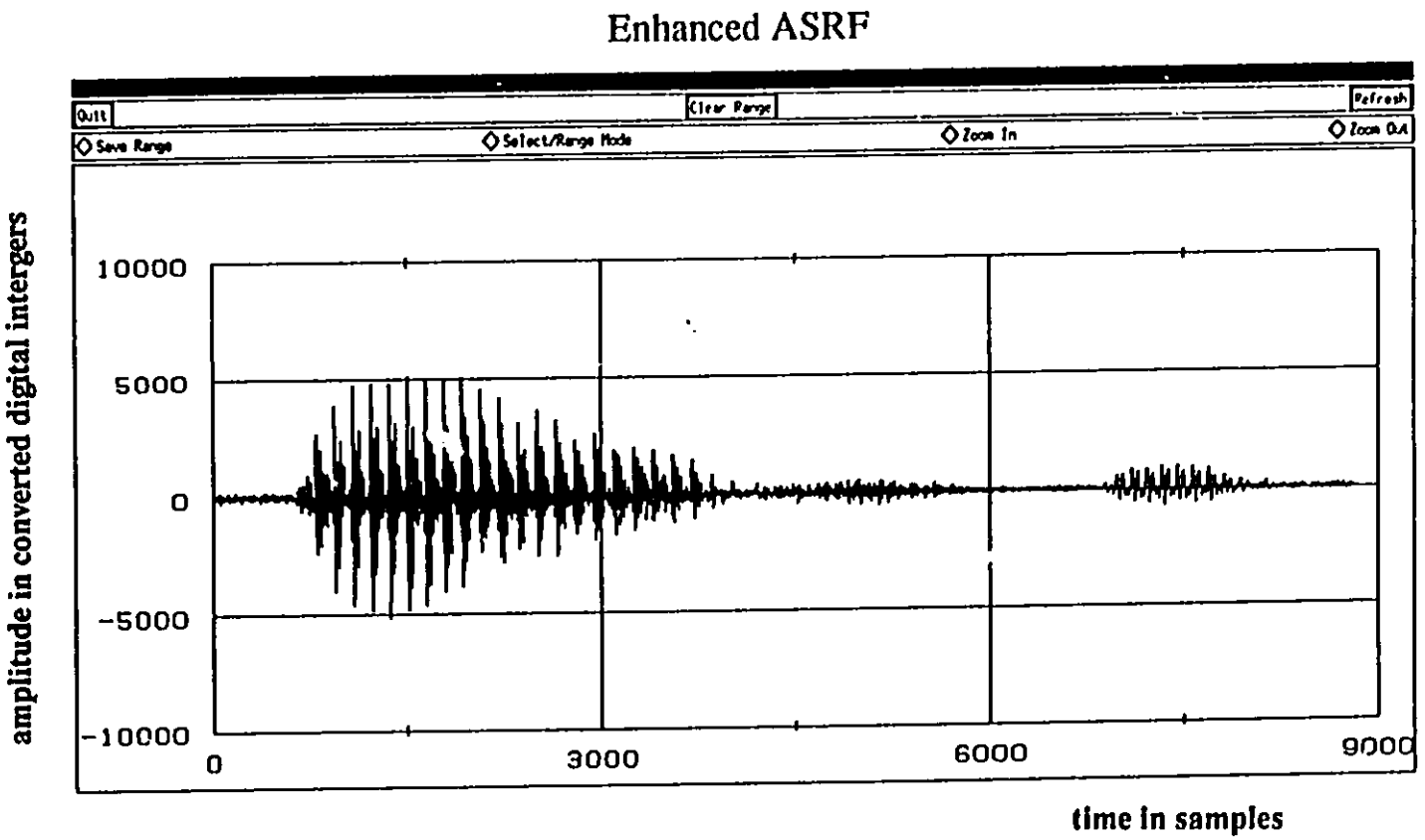
Enhanced ASRF
Quit
Clear Range
Refresh
Save Range
Select/Range Mode
Zoom In
Zoom Out
10000
5000
0
-5000
-10000
0
3000
6000
9000
amplitude in converted digital intergers
time in samples

Figure 6.6

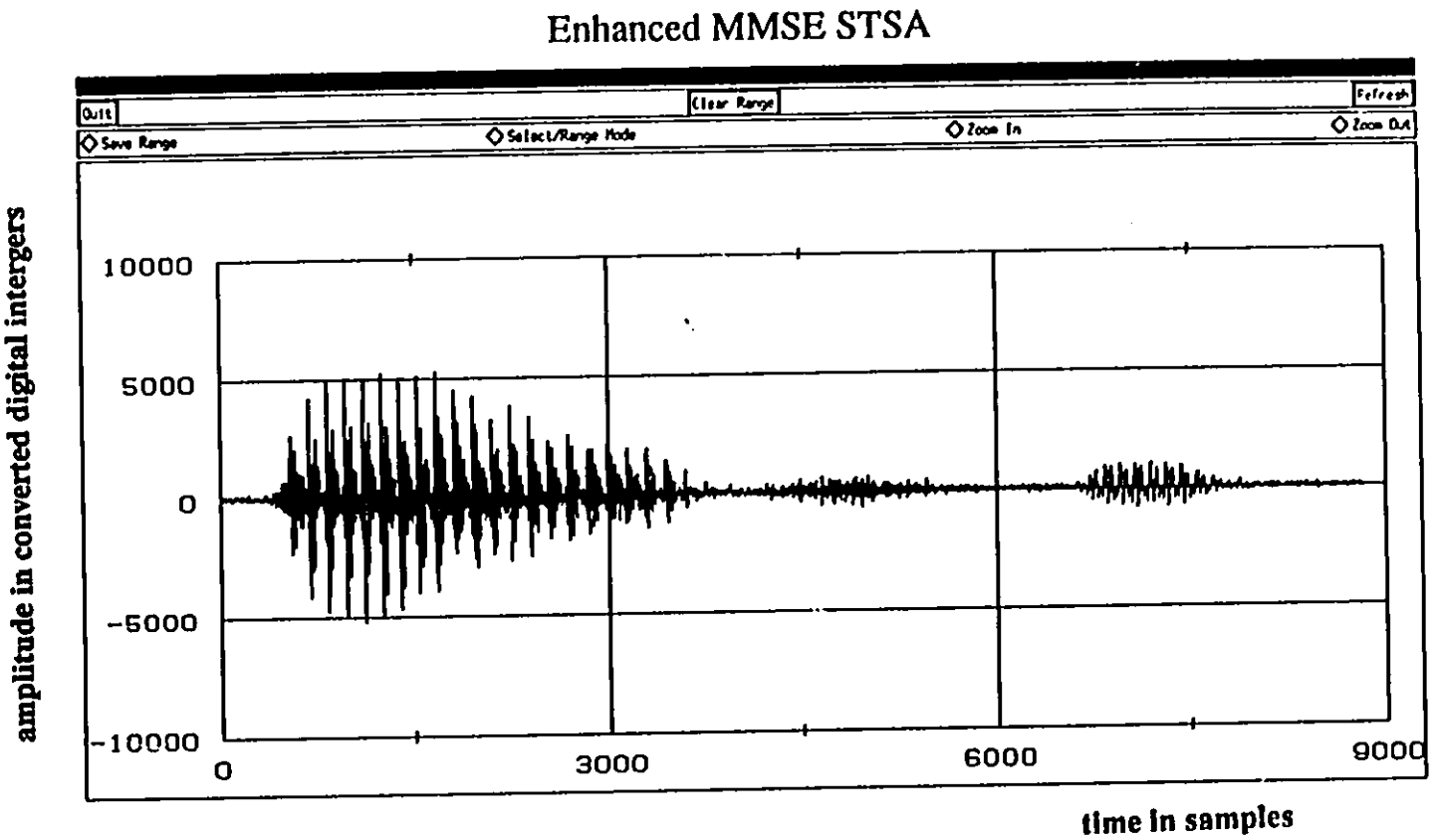
Enhanced MMSE STSA
Quit
Clear Range
Refresh
Save Range
Select/Range Mode
Zoom In
Zoom Out
10000
5000
0
-5000
-10000
0
3000
6000
9000
amplitude in converted digital intergers
time in samples

Figure 6.7

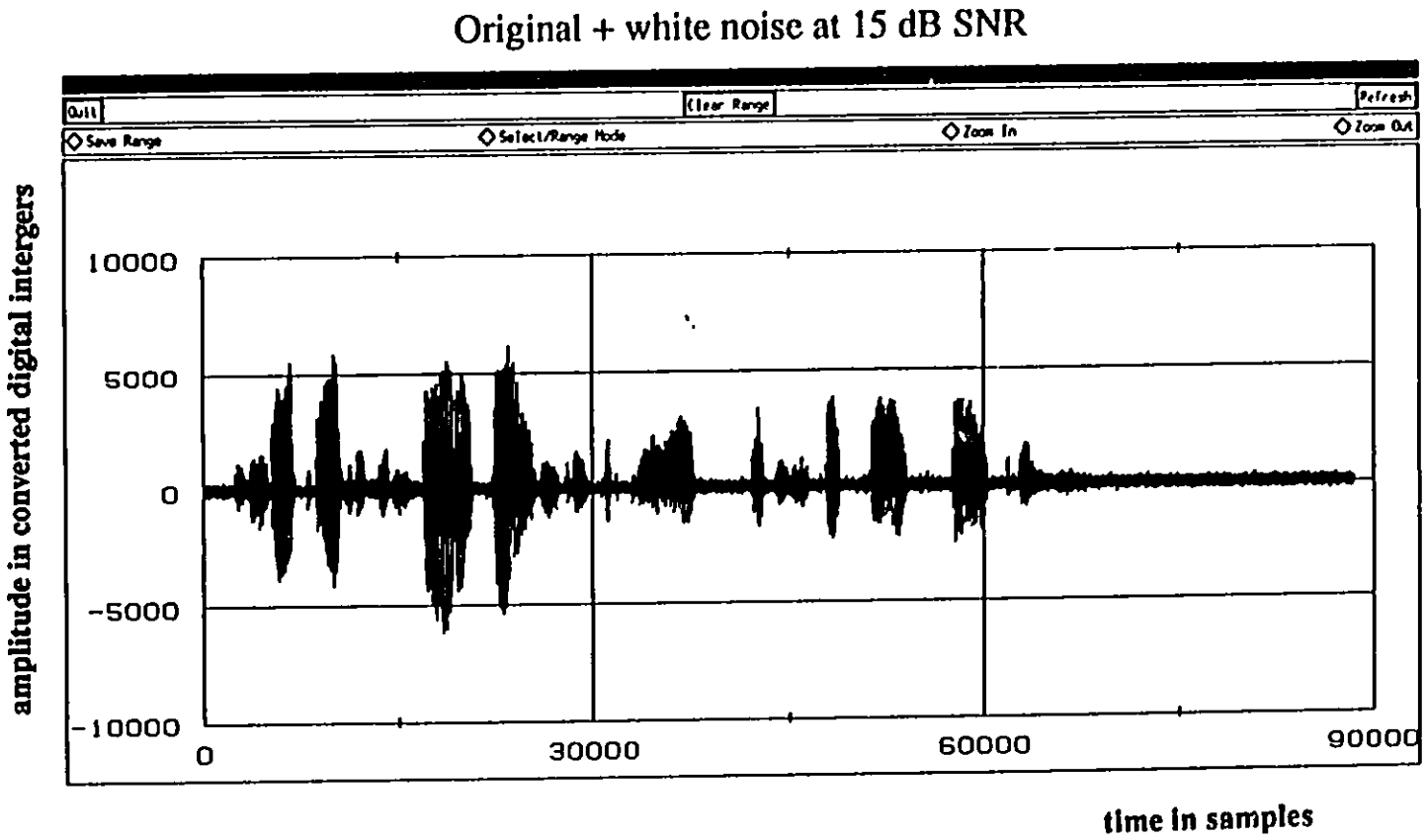
Original + white noise at 15 dB SNR
Quit
Clear Range
Refresh
Save Range
Select/Range Mode
Zoom In
Zoom Out
amplitude in converted digital intergers
10000
5000
0
-5000
-10000
0
30000
60000
90000
time in samples

Figure 6.8

Enhanced with ASRF algorithm : 15dB SNR

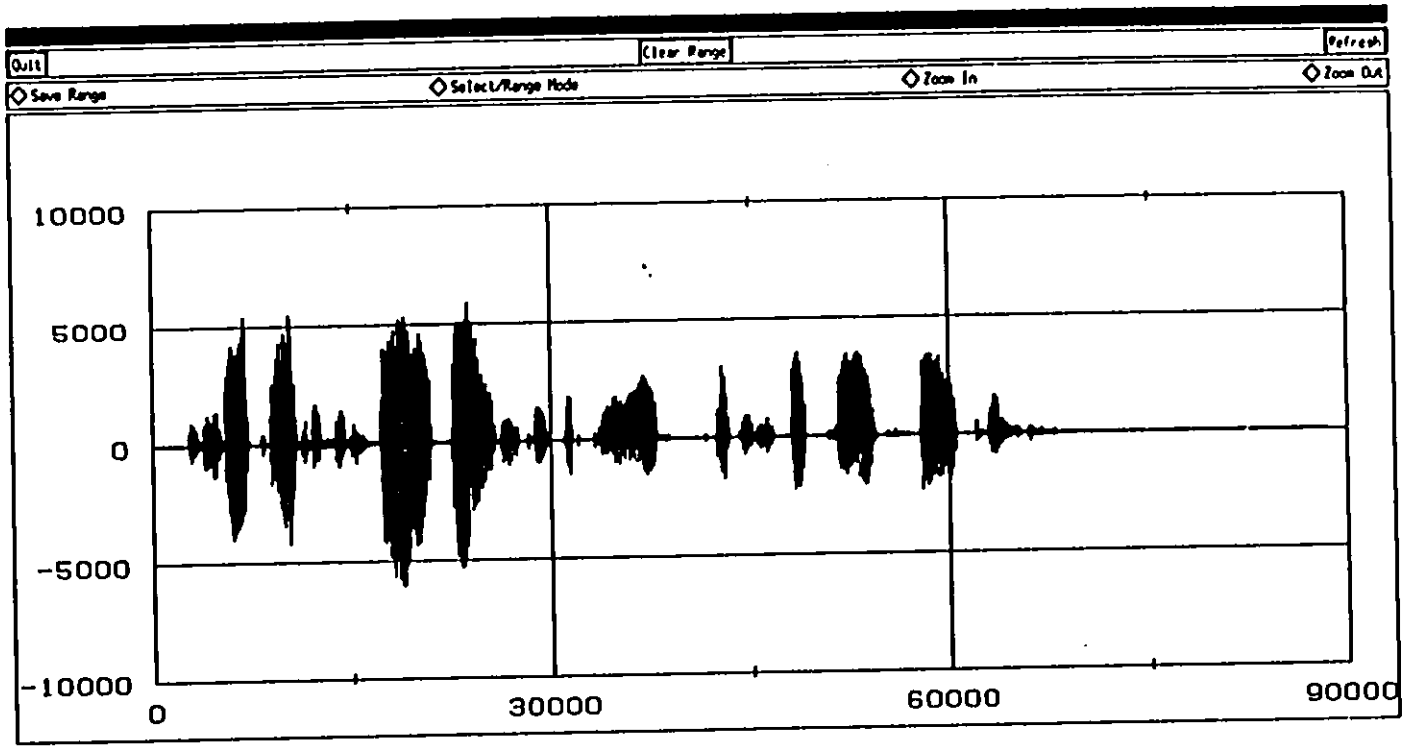

Figure 6.9

Enhanced with MMSE STSA algorithm

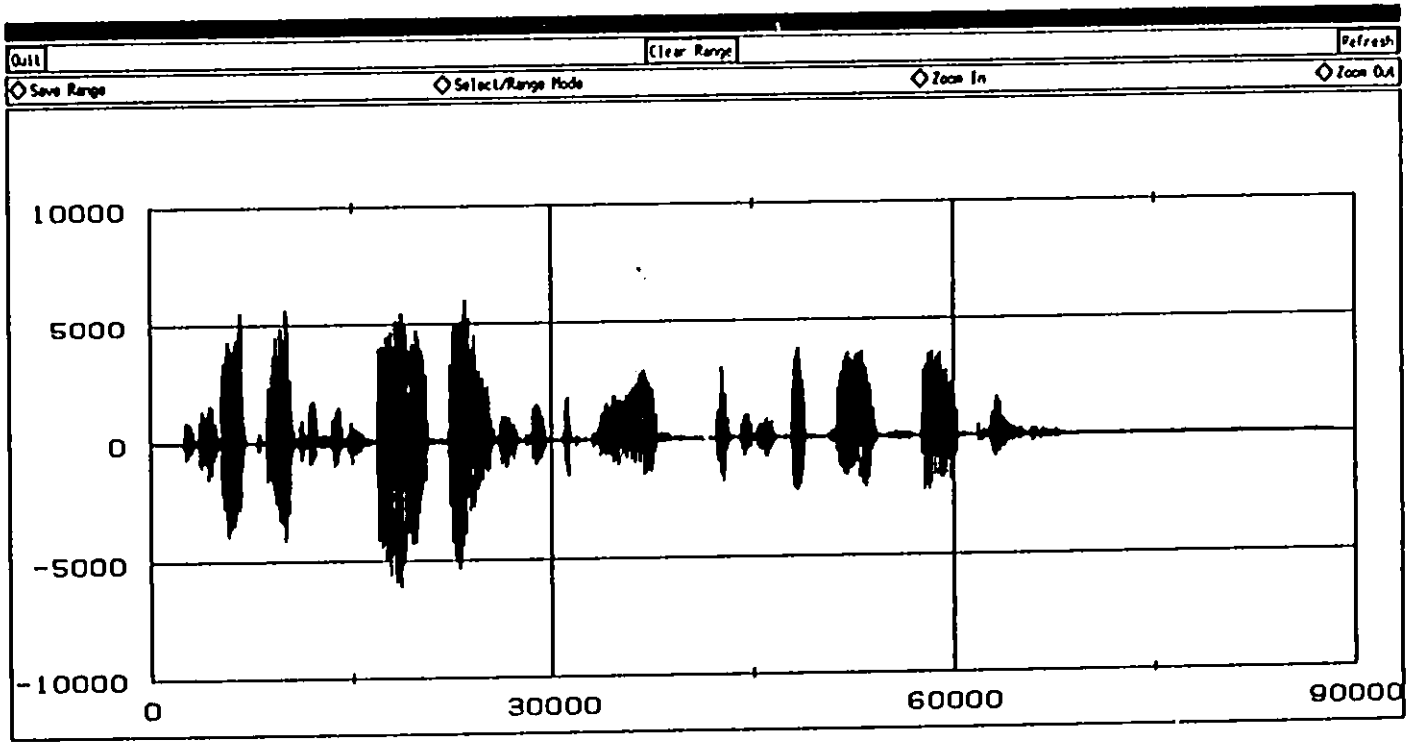

Figure 6.10

phone lasts only 459 samples. Depending on the phonetic and semantic context, this loss of acoustic information can lead to a reduction in the message intelligibility.

In figure 6.8 the same utterance degraded by white noise at a ratio of 15 dB is shown. This level of noise would be considered unacceptable for many voice communication systems. In 6.9 we show the enhanced version processed by our ASRF system. The same noisy signal was processed by the MMSE STSA method; the result is plotted in figure 6.10. There are no major differences between the enhanced speech signals.

Next, we magnify in the neighborhood of the sample n=42000 (the start of the utterance *"of"*, phonetically: /AH/ /V/) to demonstrate the effect of the non-causal estimation. First, in figure 6.11 we show the original signal without degradation. The same section enhanced by the ASRF from the noisy signal degraded by white noise at a 15 dB SNR is shown in figure 6.12. In figure 6.13, we show the resulting signal when the same ASRF algorithm is used with a delay of 10 ms before the application of the gain (smoothing). Clearly, the start of the phone /AH/ is better estimated in virtue of the look ahead capability of our algorithm.

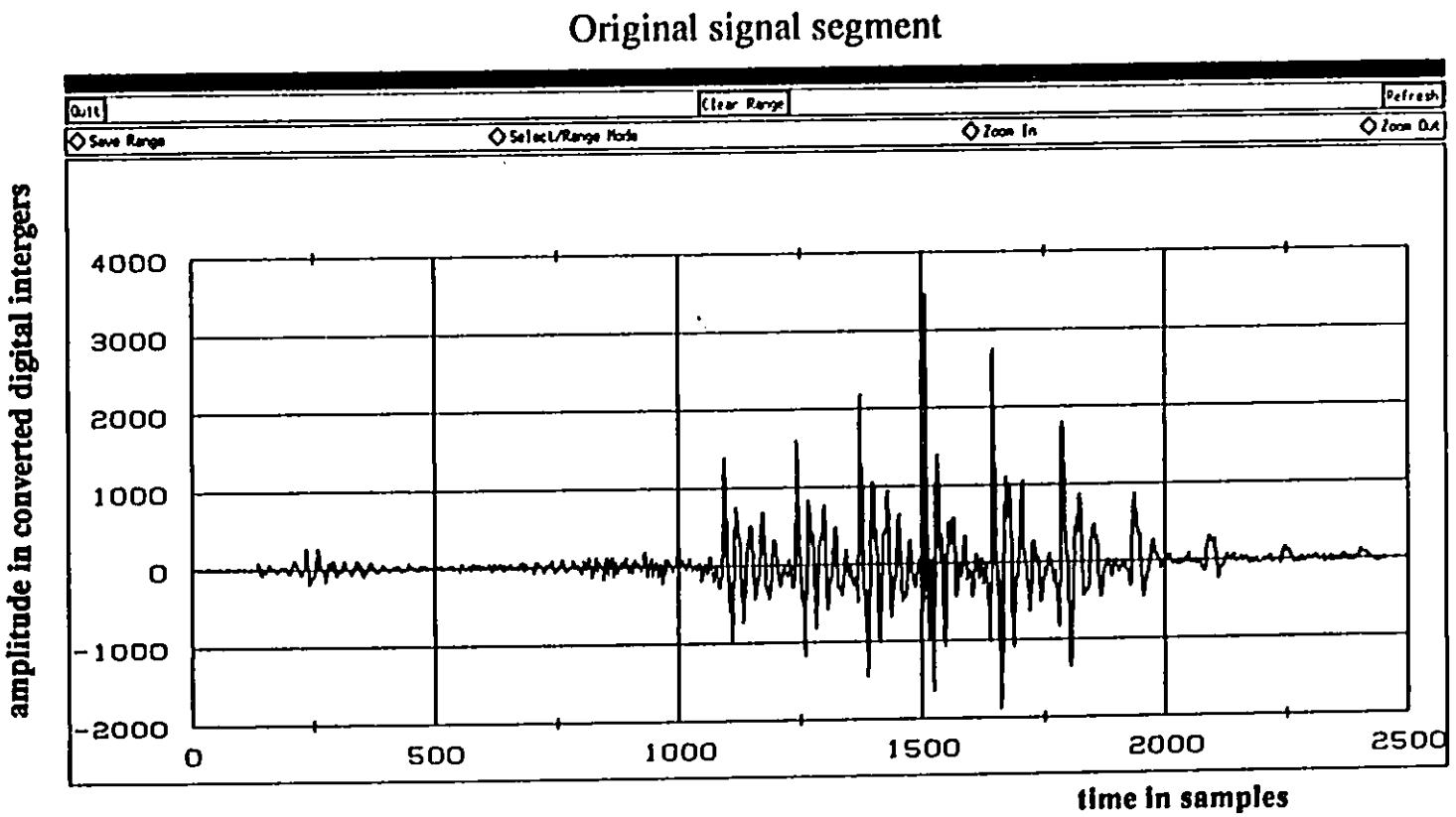
Original signal segment
Quit
Clear Range
Refresh
Save Range
Select/Range Mode
Zoom In
Zoom Out
amplitude in converted digital intergers
4000
3000
2000
1000
0
-1000
-2000
0
500
1000
1500
2000
2500
time in samples

Figure 6.11

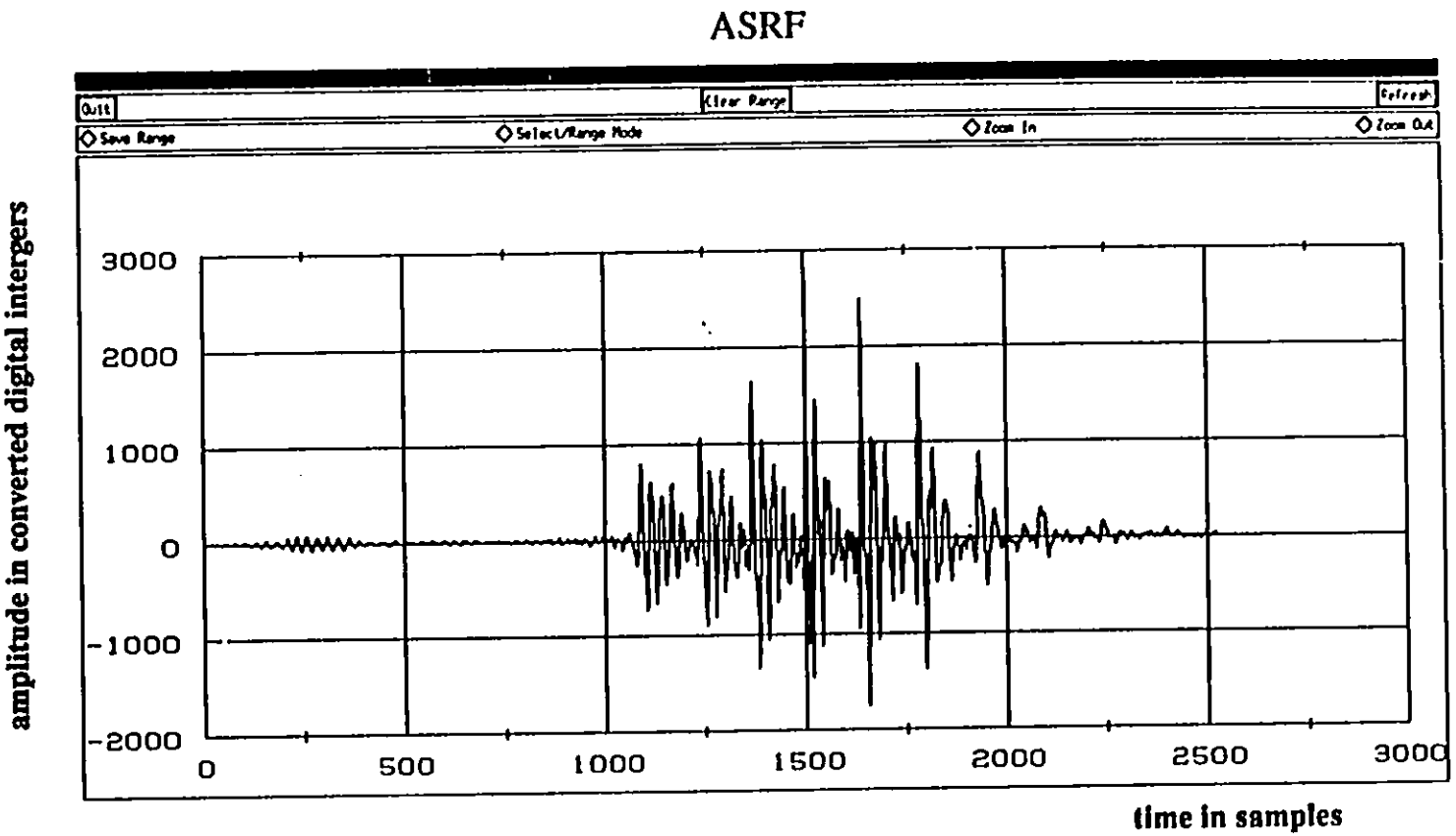
ASRF
Quit
Clear Range
Refresh
Save Range
Select/Range Mode
Zoom In
Zoom Out
amplitude in converted digital intergers
3000
2000
1000
0
-1000
-2000
0
500
1000
1500
2000
2500
3000
time in samples

Figure 6.12

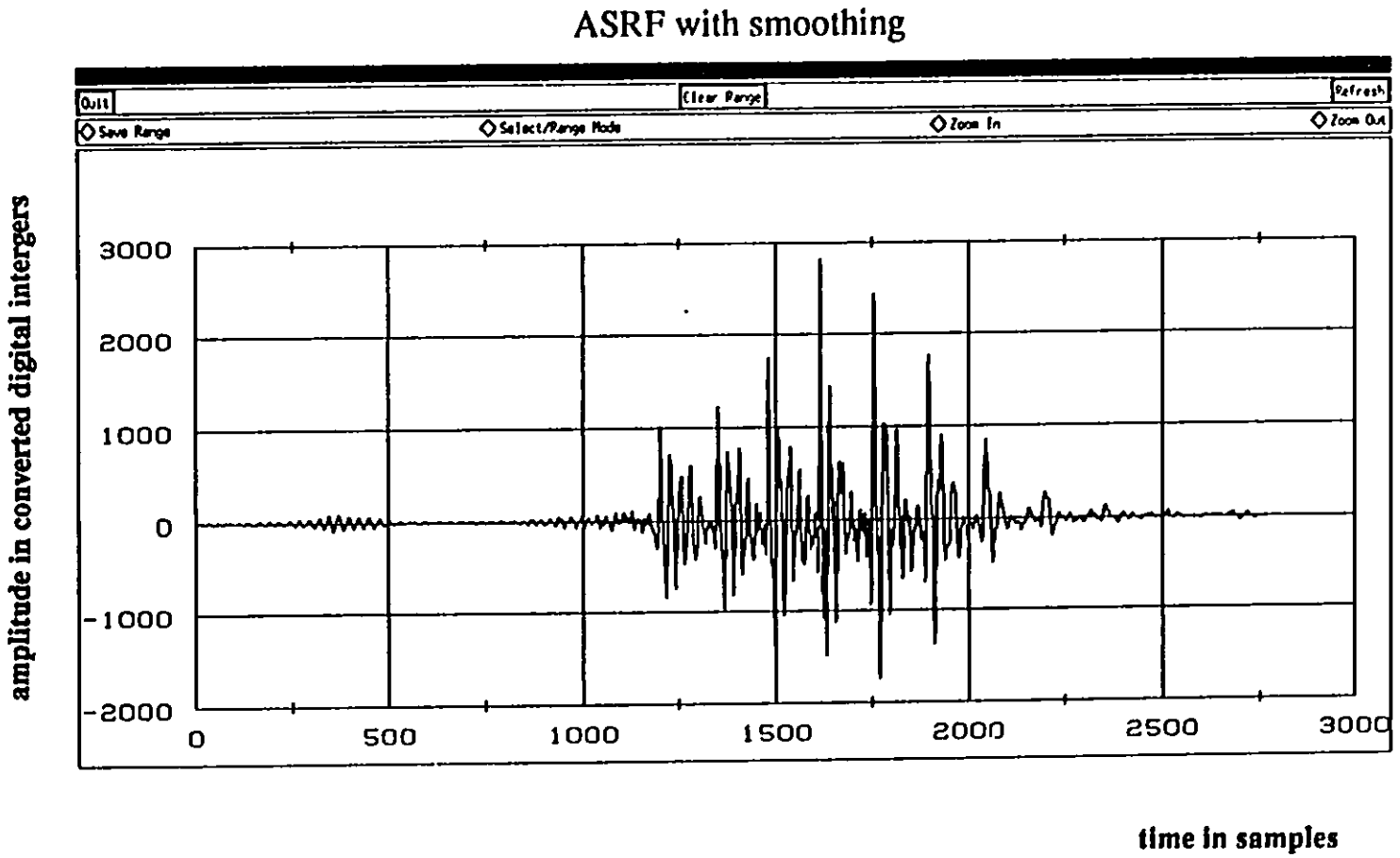
ASRF with smoothing
Quit
Clear Range
Refresh
Save Range
Select/Range Mode
Zoom In
Zoom Out
3000
2000
1000
0
-1000
-2000
0
500
1000
1500
2000
2500
3000
amplitude in converted digital intergers
time in samples

Figure 6.13

7. Conclusion

Speech enhancement provides a fertile and largely unexplored area of research. This thesis focused on the development of a speech enhancement algorithm for wideband noise reduction which can be implemented in real-time on DSP processors. We will conclude by presenting a summary of the work we accomplished and by suggesting possible extensions to this study.

7.1. Summary of results

We have presented a new analysis-synthesis structure for speech enhancement based on the theory of resonator filterbanks. Our speech enhancement system uses a parametric Wiener filter gain function to estimate the short-time spectral amplitude. Because of its generic nature, this gain function offers more flexibility than the other amplitude estimators we tried. This way, the gain function can be easily tailored for different types and levels of broadband noise. Because of the subjective nature of human perception, it is hard to compare speech enhancement systems designed to increase the quality of speech. Different listeners gave us different opinions on the quality and intelligibility of the enhanced speech produced in the experiments.

The synthesis advantage the ASRF has demonstrated in the spectral reconstruction experiments proved to be relevant for speech enhancement applications when the SNR is over 20 dB. Below 20 dB SNR, the ASRF configuration produces enhanced speech of the same quality as the other algorithms we implemented and tested.

When non-causal filtering is utilized, a better estimation of the speech parameters is possible at the cost of an increased processing delay. The available technology allows the real-time implementation of a 60 filter ASRF speech enhancement system on a single DSP processor.

7.2. Possible extensions to this work

This work addressed two important issues of the speech enhancement theory: the study of analysis and synthesis structures used to spectrally decompose and modify the speech signal; and the estimation of the speech spectral components from the noisy observations. We will propose new strategies for both issues.

7.2.1. Filterbank spacing and number of filters

Even though we experimented with Mel and logarithmic spaced filterbanks, we did not thoroughly determined the optimum number of filters and their spacing required for speech enhancement systems. We believe there are three regions to

explore to study the number of required filters: First, when the number of bands is equal or smaller than the number of critical bands; secondly, when the number of filters is such that the frequency resolution enables pitch determination and separations of the harmonics; and thirdly, the middle range where the pitch is not resolved in the frequency domain but the number of bands is larger than the number of critical bands.

7.2.2. Enhancement based upon perceptual modeling

In our perceptual experiments with the filterbanks, we demonstrated that the logarithmically spaced filterbanks introduce a faint reverb quality to the re-synthesized speech signal. We also implemented logarithmically spaced versions of the speech enhancement system and we drew the same conclusions in the speech enhancement scenario.

However, the spectral amplitude estimators we employed may not be well suited for these logarithmically spaced structures which are good models of the human auditory periphery. We think that perceptual estimation techniques based upon our knowledge of the masking theory could be used with logarithmically spaced structures. Recently, these perceptual techniques have been used for coding of high quality audio at 128 Kbs.

7.2.3. Phase estimation

As we stated in chapter 2, it is not totally true that the ear is insensitive to the phase spectral information. In fact, when the original undegraded spectral phase is used in tandem with the estimated spectral amplitudes, the produced enhanced speech is of a better quality. Therefore if the phase information could be accurately estimated, better enhanced speech would be produced.

7.2.4. Speech non-speech discrimination algorithm

Finally, in order to get a speech enhancement system which can be used in real-time, a speech non-speech activity detector has to be devised. We think NUPITCH, the scheme used in the SEU should be evaluated for potential utilization with the ASRF.

References

[1] CHABA Panel on Removal of Noise from a Speech/Noise Signal, "Removal of Noise from Noise-Degraded Speech Signals," Committee on Hearing, Bioacoustics, and Biomechanics (CHABA), National Research Council, published by National Academy Press, Washington D.C.,1989.

[2] Harris Drucker, "Speech Processing in a High Ambient Noise Environment," IEEE Transactions on Audio and Electroacoustics, Vol. AU-16, No. 2, pp. 165-168, June 1968.

[3] William D. Voiers, "Diagnostic Evaluation of Speech Intelligibility, " in *Speech Intelligibility and Speaker Recognition,* Dowden, Hutchinson and Ross, inc. Stroudsburg, Penn., pp. 374-387, 1977.

[4] William M. Kushner, Vladimir Goncharoff, Chung Wu, Vien Nguyen, and John N. Damoulakis, "The Effects of Subtractive-type Speech Enhancement /Noise Reduction Algorithms on Parameter Estimation for Improved Recognition and Coding in High Noise Environments," Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, pp. 211-214, May 1989.

[5] Steven F. Boll, "Suppression of Acoustic Noise in Speech Using Spectral Subtraction," IEEE Transactions on Acoustics, Speech, and Signal Processing,Vol. ASSP-27, No. 2, pp. 113-120, April 1979.

[6] James A. Moorer and Mark Berger, "Linear-Phase Bandsplitting: Theory and Applications," Journal of the Audio Engineering Society, Vol. 34, No. 3, pp. 143-152, March 1986.

[7] Douglas O'Shaughnessy, "Enhancing Speech Degraded by Additive Noise or Interfering Speakers, " IEEE Communications Magazine, pp. 46-52, February 1989.

[8] Thomas W. Parsons,
Voice and Speech Processing,
McGraw-Hill, New York, 1986.

[9] B. Widrow *et al.*,
"Adaptive Noise Cancelling: Principles and Applications, "
Proceedings of the IEEE, Vol. 63, No. 12, pp. 1692-1716, December 1975.

[10] Douglas O'Shaughnessy,
Speech Communication -Human and Machine,
Addison-Wesley Publishing Company, Reading, Mass., 1987.

[11] E. F. Evans,
"Representation of Complex Sounds in the Peripheral Auditory System with Particular Reference to Pitch Perception, "
in *Structure and Perception of Electroacoustic Sound and Music*,
Eds. S. Nielzen and O. Olsson, Lund, Sweden, August 1988.

[12] C. L. Searle, J. Zachary Jacobson and S.G. Rayment,
"Stop Consonant Discrimination based on Human Audition, "
J. Acoust. Soc. Am., Vol. 65, No. 3, pp. 799-809, March 1979.

[13] J. C. R. Licklider,
"Effects of Amplitude Distortion upon the Intelligibility of Speech, "
J. Acoust. Soc. Am., Vol. 16, No. 2, pp. 429-434, October 1946.

[14] M. R. Weiss, E. Aschkenasy and T. W. Parsons,
"Study and development of the INTEL technique for improving speech intelligibility,"
Nicolet Scientific Corporation, Final Technical Report, RADC - TR-75 -108, April 1975.

[15] M. R. Weiss, and E. Aschkenasy,
"The Speech Enhancement Advanced Development Model,"
Final Technical Report, RADC - TR-78 -232, November 1978.

[16] S. V. Vaseghi and P. J. W. Rayner,
" Detection and Suppression of Impulsive Noise in Speech Communication Systems, "
IEE Proceedings, Vol. 137, Part I, Number 1,pp. 38-46, February 1990.

[17] H.J.M. Steeneken and F.W.M. Geurtsen,
"Description of the RSG-10 Noise Database,"
File included in the CD-ROM. Canadian point of contact to get the database: Dr. M. Taylor, Defence and Civil Institute for Environmental Medicine, Downsview, Canada.

[18] Jae S. Lim, Alan V. Oppenheim and Louis D. Braida,
"Evaluation of an Adaptive Comb Filtering Method for Enhancing Speech Degraded by White Noise Addition," IEEE Transactions on Acoustics, Speech, and Signal Processing,Vol. ASSP-26, No. 4, pp. 354-358, August 1978.

[19] T. F. Quatieri and R. J. McAulay,
"Noise Reduction Using a Soft-Decision Sine-Wave Vector Quantizer,"
Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, pp. 821-824, April 1990.

[20] Yariv Ephraim,
"A Minimum Mean-Square Error Approach For Speech Enhancement,"
Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, pp. 829-832, April 1990.

[21] M. Berouti, R. Schwartz, and J. Makhoul,
"Enhancement of Speech Corrupted by Acoustic Noise,"
Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, pp. 208- 211, April 1979.

[22] Robert J. McAulay and Marilyn L. Malpass,
"Speech Enhancement Using a Soft-Decision Noise Suppression Filter,"
IEEE Transactions on Acoustics, Speech, and Signal Processing,Vol. ASSP-28, No. 2, pp. 137-145, April 1980.

[23] Athanasios Papoulis,
Probability, Random Variables, and Stochastic Processes,
McGraw-Hill, New York, 1984.

[24] Yariv Ephraim and David Malah,
"Speech Enhancement Using a Minimum Mean-Square Error Short-Time Spectral Amplitude Estimator,"
IEEE Transactions on Acoustics, Speech, and Signal Processing,Vol. ASSP-32, No. 6, pp. 1109-1121, December 1984.

[25] Mischa Schwartz and Leonard Shaw,
Discrete Spectral Analysis, Detection, and Estimation,
McGraw-Hill, New York, 1975.

[26] J. S. Lim and A. V. Oppenheim,
"Enhancement and Bandwidth Compression of Noisy Speech,"
Proceedings of the IEEE, Vol. 67, No. 12, pp. 1586-1604, December 1979.

[27] Thomas E. Eger, James C. Su, and L. William Varner,
"A Nonlinear Spectrum Processing Technique for Speech Enhancement,"
Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, paper 18A, March 1984.

[28] M.R. Portnoff,
"Time-Frequency Representation of Digital Signals and Systems Based on Short -Time Fourier Analysis,"
IEEE Transactions on Acoustics, Speech, and Signal Processing,Vol. ASSP-28, No. 1, pp. 55-69, February 1980.

[29] J. B. Allen,
"Short-term Spectral Analysis, Synthesis, and Modification by Discrete Fourier Transform,"
IEEE Transactions on Acoustics, Speech, and Signal Processing, Vol. ASSP-25, No. 3, pp. 235-238, June1977.

[30] M. R. Portnoff,
"Implementation of the Digital Phase Vocoder using the Fast Fourier Transform," IEEE Transactions on Acoustics, Speech, and Signal Processing.,Vol. ASSP 24, pp. 243-246, June 1976.

[31] J. L. Flanagan and R. M. Golden,
"Phase vocoder,"
Bell System Tech. J., Vol 45, pp. 1493-1509, Nov. 1966.

[32] W.F. McGee and Genzao Zhang,
"Logarithmic Filterbanks,"
International Symposium on Circuits and Systems, pp. 661-664,New Orleans 1990.

[33] L. Gagnon and W. F. McGee,
Nonlinear Processing of Phase Vocoded Speech,"

Canadian Conference on Electrical and Computer Engineering, pp. 5.1.1-5.1.4,Ottawa, Sept. 90

[34] J.S. Garofolo,
"The structure and format of the DARPA TIMIT CD-ROM Prototype,"
NIST publication,Gaithersburg, Maryland, telephone: 301-975-3193, 1988.

[35] J. S. Lim,
Speech Enhancement,
Prentice-Hall, Englewood Cliffs, New Jersey, 1983.

[36] L.R. Rabiner and R.W. Schafer,
Digital Processing of Speech Signals,
Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[37] Yariv Ephraim and David Malah,
"Speech Enhancement Using A Minimum Mean-Square Error Log-Spectral Amplitude Estimator,"
IEEE Transactions on Acoustics, Speech, and Signal Processing,Vol. ASSP-33, No. 2, pp. 443-445, April 1985.

[38] J. L. Flanagan,
"Voices of Men and Machines,"
J. Acoust. Soc. Am., Vol. 51, No. 5, pp. 1375-1387, March 1972.

[39] R. E. Crochiere and L. R. Rabiner,
Multirate Digital Signal Processing,
Prentice-Hall, Englewood Cliffs, New Jersey, 1983.

Appendix A: Filter parameters and coefficients used in experiments

Table A.1: Sixth-octave logarithmic filterbank with 40 filters

Filter number	Weight: real part	Weight: imaginary part
1	0.0554571917	-0.0011590086
2	0.0494108659	-0.0008117714
3	0.0440222962	-0.0005729019
4	0.0392207766	-0.0003908143
5	0.0349426164	-0.0002446299
6	0.0311308464	-0.0001242545
7	0.0277346414	-0.0000236360
8	0.0247086884	0.0000613209
9	0.0220125998	0.0001335870
10	0.0196103832	0.0001954137
11	0.0174699656	0.0002485588
12	0.0155627649	0.0002944269
13	0.0138633023	0.0003341516
14	0.0123488833	0.0003686542
15	0.0109992815	0.0003987046
16	0.0097964692	0.0004249432
17	0.0087243787	0.0004479078
18	0.0077686873	0.0004680531
19	0.0069166266	0.0004857659
20	0.0061568110	0.0005013776
21	0.0054790860	0.0005151737
22	0.0048743915	0.0005274026
23	0.0043346400	0.0005382815
24	0.0038526078	0.0005480031
25	0.0034218361	0.0005567403
26	0.0030365429	0.0005646504
27	0.0026915419	0.0005718808
28	0.0023821660	0.0005785721
29	0.0021041978	0.0005848622
30	0.0018537988	0.0005908917
31	0.0016274368	0.0005968080

32	0.0014218028	0.0006027710
33	0.0012337066	0.0006089578
34	0.0010599273	0.0006155643
35	0.0008969676	0.0006227979
36	0.0007405936	0.0006308343
37	0.0005848348	0.0006396490
38	0.0004193900	0.0006483256
39	0.0002208966	0.0006514753
40	-0.0000944401	0.0006055348

Note: The center frequencies are selected as follows:

$\omega_i = \omega_o \; e^{i \ln 2 \, 1/6}$

where i is the filter number ($0 <= i <= 39$) and $\omega_o = 0.0110485435 \, \pi \, 2^{-1/12}$.

Table A.2: Mel-spaced filterbank with 34 filters

Center normalized frequency. $\omega i / 2\pi$	Weight: real part.	Weight : imaginary part.	Filter number.
0.4713736096	0.0556427329	0.0012438415	34
0.4188506805	0.0495738641	0.0010232557	33
0.3720580703	0.0441659747	0.0008779193	32
0.3303705939	0.0393477248	0.0007661498	31
0.2932312745	0.0350550125	0.0006735209	30
0.2601439026	0.0312305749	0.0005943286	29
0.2306664053	0.0278233527	0.0005256061	28
0.2044049407	0.0247878415	0.0004654870	27
0.1810086356	0.0220834948	0.0004126430	26
0.1601648974	0.0196741865	0.0003660531	25
0.1415952377	0.0175277302	0.0003248936	24
0.1250515518	0.0156154503	0.0002884797	23
0.1103128031	0.0139117989	0.0002562307	22
0.0971820708	0.0123940154	0.0002276481	21
0.0854839183	0.0110418218	0.0002022998	20
0.0750620492	0.0098371526	0.0001798093	19
0.0657772193	0.0087639129	0.0001598469	18
0.0575053764	0.0078077640	0.0001421232	17

0.0501360020	0.0069559312	0.0001263834	16
0.0435706359	0.0061970336	0.0001124029	15
0.0377215596	0.0055209322	0.0000999834	14
0.0325106251	0.0049185936	0.0000889497	13
0.0278682102	0.0043819706	0.0000791469	12
0.0237322887	0.0039038933	0.0000704383	11
0.0200476015	0.0034779745	0.0000627029	10
0.0167649184	0.0030985235	0.0000558343	9
0.0138403803	0.0027604706	0.0000497391	8
0.0112349130	0.0024592993	0.0000443365	7
0.0089137056	0.0021909856	0.0000395579	6
0.0068457448	0.0019519443	0.0000353499	5
0.0050034012	0.0017389813	0.0000316804	4
0.0033620597	0.0015492500	0.0000285627	3
0.0018997906	0.0013802118	0.0000261397	2
0.0005970570	0.0012295995	0.0000247947	1

Table A.3: Linear filterbanks with N filters

Center frequency	Weight (no imaginary part)
$\omega_i = \pi i / N$	$1/N$

Appendix B: Software programs

The following pages are listings of the main C programs which were written for this thesis.

Name of program	Function(s)
gdnt_d.c	Speech enhancement using the ASRF/Parametric Wiener configuration.
gdnt_noise.c	Noise estimation for the gdnt algorithm.
block.c	Speech enhancement algorithms sharing the STSA OLA analysis-synthesis method.
noise_stsa.c	Noise estimation for the algorithms implemented in the block.c program.
subceps.c	Speech enhancement with the INTEL method.
noise_comceps.c	Noise estimation in the pseudo-cepstral domain for the INTEL method.

```
/*********************************************************/
/* gdnt_d.c
Speech Enhancement using an analysis and synthesis
filterbank.
Allpass structure is used.
Filterbank proposed by McGee :  see referencec.
delay is implemented for smoothing capability.
                                                         */
/*********************************************************/

#include <stdio.h>
#include <math.h>
#include "complex2.h"              /*  complex routine library.*/
#define SIZE            128        /* filter bank size*/
#define FS              16000      /* sampling frequency */
#define BUFF            1024       /* buffer to read from file*/
#define MAXSHORT        32767      /* max value for short*/
#define MINSHORT        -32768     /* min value for short*/
#define MAXDELAY        800        /* max delay line length*/
/***************************/
/* typedef and structures.*/
/***************************/

/*  filterbank structure */
typedef struct
        {
        COMPLEX poles[SIZE];       /* poles of filterbank*/
        COMPLEX weight[SIZE];      /* weight of filters */
        float freqs[SIZE];         /* resonant freqs of filterbank*/
        float wdig[SIZE];          /* freqs in radians of filterbank*/
        int size;                  /* size of filter bank.*/
        } FILTERBANK;

/* status of  analysis filterbank*/
typedef struct
        {
        COMPLEX  error;                    /* sum of filters */
        COMPLEX HK[SIZE];                  /* output of filters*/
        } AN_STATUS;                       /* status of filters*/

/* status of synthesis filterbank*/
typedef struct
        {
        COMPLEX sum[SIZE];                 /* sum of filters */
        } SY_STATUS;                       /* status of filters*/

typedef struct
        {
        int index;
        float wre;
        float wima;
        } TAB;                     /* table containing weights of FB*/

typedef struct
        {
        COMPLEX fil[SIZE];
        } SYNFIL;                  /* data passed to synthezizer.*/

typedef struct
        {
        AN_STATUS tap[MAXDELAY];
        } DELAYLINE;               /* delay line.*/
```

```
/************************************/
/* Global variables and constants */
/************************************/


FILE * f1;                  /* input  speech file*/
FILE * f2;                  /* output speech file*/
FILE * f3;                  /* noise file*/
long   ii;                  /* counter */
int   k;                    /* counter */
long  rest;                 /* rest of the input files*/
long nsamples,niterations ;/* counter variables */
float pi=3.141592658;       /* float is used for computational speed reasons */
float estimate[SIZE];       /* noise estimate:variance envelope **2  */
short in[BUFF];             /* input signal from file*/
int strategy;               /* strategy for spectral modifications.*/
float  epsilon[SIZE];       /* estimated a priori SNR*/
float  envelope[SIZE];      /* estimated envelope */
float overestimate;         /* to multiply noise estimates by a factor . */
float learn ;               /* slowfast learning rate*/
float whitener  ;           /* whitening factor a la BBN*/
float exponent  ;           /*  for noise trap */
FILTERBANK *analyse;        /* ptr  to analisys filterbank used.*/
FILTERBANK analysis;        /*  analisys filterbank used.*/
AN_STATUS *an ;             /* ptr to status of filtbank 1.*/
AN_STATUS an_state ;        /* status of filtbank 1.*/
FILTERBANK *synthese;       /* ptr  to  synthesis filterbank used.*/
FILTERBANK synthesis;       /*  synthesis  filterbank used.*/
SY_STATUS *sy ;             /* ptr to status of filtbank 2.*/
SY_STATUS syn_state ;       /* status of filtbank 2.*/
SYNFIL  *syin;              /*  input to synthesizer */
SYNFIL  syinput;
DELAYLINE delayline;        /* delay line for looking unto the future.*/
int current=0;              /* pointer to delay line*/
int delay;                  /* pointer to delay line*/
TAB tab[SIZE] = {

1,      0.0554571917,   -0.0011590086,
2,      0.0494108659,   -0.0008117714,
3,      0.0440222962,   -0.0005729019,
4,      0.0392207766,   -0.0003908143,
5,      0.0349426164,   -0.0002446299,
6 ,     0.0311308464,   -0.0001242545,
7,      0.0277346414,   -0.0000236360,
8,      0.0247086884,   0.0000613209,
9,      0.0220125998,   0.0001335870,
10,     0.0196103832,   0.0001954137,
11 ,     0.0174699656,  0.0002485588,
12,     0.0155627649,   0.0002944269,
13,     0.0138633023,   0.0003341516,
14,     0.0123488833,   0.0003686542,
15,     0.0109992815,   0.0003987046,
16 ,    0.0097964692,   0.0004249432,
17,     0.0087243787,   0.0004479078,
18,     0.0077686873,   0.0004680531,
19,     0.0069166266,   0.0004857659,
20 ,     0.0061568110,  0.0005013776,
21,     0.0054790860,   0.0005151737,
22,     0.0048743915,   0.0005274026,
23,     0.0043346400,   0.0005382815,
24,     0.0038526078,   0.0005480031,
25,     0.0034218361,   0.0005567403,
26,     0.0030365429,   0.0005646504,
27,     0.0026915419,   0.0005718808,
28,     0.0023821660,   0.0005785721,
29,     0.0021041978,   0.0005848622,
```

```
30,     0.0018537988,   0.0005908917,
31,     0.0016274368,   0.0005968080,
32,     0.0014218028,   0.0006027710,
33,     0.0012337066,   0.0006089578,
34,     0.0010599273,   0.0006155643,
35,     0.0008969676,   0.0006227979,
36,     0.0007405936,   0.0006308343,
37,     0.0005848348,   0.0006396490,
38,     0.0004193900,   0.0006483256,
39,     0.0002208966,   0.0006514753,
40 ,    -0.0000944401,  0.0006055348,
};


/***************************/
/* end of global variable */
/***************************/


/*****************************************/
/* FUNCTIONS                             */
/*****************************************/

/***************************/
/* file functions          */
/***************************/
void open_input(fichier)
        char *fichier;
{
        f1= fopen(fichier,"r+b");
}


void open_output(fichier)
char *fichier;
{
        f2= fopen(fichier,"w+b");
}


void readblock(buffer, length)
short *buffer; int length;
{
        fread(buffer,2,length,f1);
}     /* reads integers of 16 bits compatible the A/D */


void writeblock(buffer,  length)
short *buffer; int length;
{
        fwrite(buffer,2,length,f2);
}           /* writes integers of 16 bits compatible with the D/A */


/***********************************/
/* spectral estimation function   */
/***********************************/


id(envelope,filter,estimate,state)
/*  id system verification */
```

```
float *envelope ;         /* envelope to estimate*/
FILTERBANK *filter;      /* filter*/
float *estimate;         /* estimate of noise*/
AN_STATUS *state;        /* state of filter*/
{
int i;
COMPLEX ctemp ;
POLAR outpol;    /* output in polar form*/
POLAR est;       /* estimated value of channel envelope*/
float gain;      /* non linear gain*/
for(i=0;i<filter->size;i++)
        {    /* for all channels*/
        ctemp=  state->HK[i];
        outpol=transform_polar(ctemp); /* output of filter in polar form*/
        gain= 1.0;
        est.r = outpol.r * gain;        /* estimated envelope*/
        envelope[i]=est.r;
        }
}



gdnt(envelope,filter,estimate,state,epsilon,whitener,exponent)
/* Wiener filtering approach to speech enhancement with whitener */

float *envelope ;         /* envelope to estimate*/
FILTERBANK *filter;      /* filter*/
float *estimate;         /* estimate of noise*/
AN_STATUS *state;        /* state of filter*/
float *epsilon;          /* a priori SNR 's */
float whitener;          /* min gain*/
float exponent;          /* exp for noise trap effect.*/
{
int i;
COMPLEX ctemp ;
float temp;
POLAR outpol;    /* output in polar form*/
POLAR est;       /* estimated value of channel envelope*/
float gain,a;    /* channel gain*/
for(i=0;i<filter->size;i++)
        {    /* for all channels*/
        ctemp=  state->HK[i];
        outpol=transform_polar(ctemp); /* output of filter in polar form*/
        temp =  epsilon[i] / ( epsilon[i] +1.0  );
        /* gain is computed..*/
        gain= pow(temp,exponent);
        est.r = outpol.r * gain;        /* estimated envelope*/
        envelope[i]=est.r;
        if (envelope[i]<= (a=sqrt(whitener * estimate[i])) )
                envelope[i]=a;
        }
}



gdnt_half(envelope,filter,estimate,state,epsilon,whitener,exponent)
/* Wiener filtering approach to speech enhancement with whitener */

float *envelope ;         /* envelope to estimate*/
FILTERBANK *filter;      /* filter*/
float *estimate;         /* estimate of noise*/
AN_STATUS *state;        /* state of filter*/
float *epsilon;          /* a priori SNR 's */
```

```
float whitener;          /* min gain*/
float exponent;          /* exp for noise trap effect.*/
{
int i;
COMPLEX ctemp ;
float temp;
POLAR outpol;    /* output in polar form*/
POLAR est;       /* estimated value of channel envelope*/
float gain,a;    /* channel gain*/
for(i=(filter->size/2);i<filter->size;i++)
        envelope[i]=0.0;                    /* frequency >4 Khz are zeroized.*/
for(i=0;i<(filter->size/2);i++)
        {    /* for only half the channels  */
        ctemp=  state->HK[i];
        outpol=transform_polar(ctemp); /* output of filter in polar form*/
        temp =  epsilon[i] / ( epsilon[i] +1.0  );
        /* gain is computed..*/
        gain= pow(temp,exponent);
        est.r = outpol.r * gain;          /* estimated envelope*/
        envelope[i]=est.r;
        if (envelope[i]<= (a=sqrt(whitener * estimate[i])) )
                envelope[i]=a;
        }
}


slowfast(envelope,state,estimate, epsilon, filter,learning_rate)
/* evaluation of a priori SNR with the avg recursion*/
/* measured  value is P=v2/sigma*2 -1 */

float *envelope;         /* estimated envelope of signal*/
AN_STATUS *state;                /* state of filter*/
float *estimate;         /* noise estimates*/
float *epsilon;          /* a priori SNR's*/
FILTERBANK *filter;      /* to get the size*/
float learning_rate;     /* for recursive estimation*/
{
int i;
POLAR v;                 /* polar output of filter.*/
float a,v2,P;
for(i=0;i< filter->size;i++)
        {
        v = transform_polar(state->HK[i]);
        v2 = v.r * v.r;
        P= (v2/estimate[i]) - 1.0; /* try sn-1/lamb*/
        if (P< 0.0) P=0;
        a= (1.0 - learning_rate) * (P-epsilon[i]) + epsilon[i];
        epsilon[i]= a;
        }
}


slowfast2(envelope,state,estimate, epsilon, filter,learning_rate)
/* evaluation of a priori SNR with the avg recursion*/

float *envelope;         /* estimated envelope of signal*/
AN_STATUS *state;                /* state of filter*/
float *estimate;         /* noise estimates*/
float *epsilon;          /* a priori SNR's*/
FILTERBANK *filter;      /* to get the size*/
float learning_rate;     /* for recursive estimation*/
{
int i;
POLAR v;                 /* polar output of filter.*/
float a,v2,P,P1,P2;
```

```
for(i=0;i< filter->size;i++)
        {
        v = transform_polar(state->HK[i]);
        v2 = v.r * v.r;
        P1= (v2/estimate[i]) - 1.0;
        P2 =  envelope[i]* envelope[i] / estimate[i];
        P = (P1 +P2)/2;
        if (P< 0.0) P=0;
        a= (1.0 - learning_rate) * (P-epsilon[i]) + epsilon[i];
        epsilon[i]= a;
        }
}


/*************************************************************/
/* Filter functions                                          */
/*************************************************************/




selection_DFT(filter)
/* selection of parameters for  DFT filterbank*/
FILTERBANK *filter;
{
int i;
int size=SIZE;
filter->size=size;
for(i=0;i<(size);i++)
        {
        filter->freqs[i] = (float)i/(float)(2*size); /* digital freqs*/
        filter->wdig[i]=2.0*pi*(float)i/(float)(2*size); /* digital freqs in rad
        }
for(i=0;i<(size);i++)
        {
        filter->poles[i].re=cos(filter->wdig[i]);
        filter->poles[i].ima=sin(filter->wdig[i]);
        }

for(i=0;i<(size);i++)
        {
        filter->weight[i].re=2.0/(float)(2*size);
        filter->weight[i].ima=0.0;
        }
}




selection_LOG40(filter)
/* selection of parameters for  logarithmic  filterbank*/
FILTERBANK *filter;
{
int i;
POLAR coco;
float summ;
float factor= 0.8908987181  ;                   /* 2**-1/6  octave spacing.*/
filter->size= 40;
filter->wdig[39] = 0.9438743127 * pi;   /*  by definition  pi* 2**(-1/12) */
for(i=38;i>=0;i--)
        {
        filter->wdig[i]= pow((double)factor,(double)(39 -i))  * filter->wdig[39]
        }
for(i=0;i<(filter->size);i++)
```

```
        {
        filter->poles[i].re=cos(filter->wdig[i]);
        filter->poles[i].ima=sin(filter->wdig[i]);
        }
summ=0;
for(i=0;i<(filter->size);i++)
        {
        filter->weight[i].re= 2* tab[39-i].wre;          /* table in ascending or
        filter->weight[i].ima=2*  tab[39-i].wima;
        coco= transform_polar(filter->weight[i]);
        summ += coco.r;
        }
printf(" sum= %f\n ",summ);
}
```

```
float synthesis_filter(inc ,state, filter )
/* synthesis filterbank operation*/

SYNFIL *inc ;            /* incoming signals in complex*/
SY_STATUS *state;                /* status of filter at entry time*/
FILTERBANK *filter;      /*  filterbank used */
{
int n,k;
COMPLEX temp1,temp2,temp3;
COMPLEX OM[SIZE];
COMPLEX outcomplex;
float out;
for(n=0;n<filter->size;n++)
        {
        temp1.re= filter->weight[n].re;
        temp1.ima= -1.0 * filter->weight[n].ima;
        temp2=mult_complex(temp1, inc->fil[n]);
        OM[n]= sum_complex(temp2, state->sum[n]);
        }
/* filtering synthesis operation*/
out=0;

for(n=0;n<filter->size;n++)
        {
        out += OM[n].re;
        }

outcomplex.re=out;
outcomplex.ima=0.0;
for(n=0;n<filter->size;n++)
        {
        temp1= mult_complex(filter->weight[n],outcomplex);
        temp2= diff_complex(OM[n],temp1);
        state->sum[n]= mult_complex(filter->poles[n],temp2);
        }
return(out);

}
```

```
void analysis_filter(in ,state, filter )
```

```
/* analisys filtering operation is done here.*/
/* no attempt is made to reduce computations.*/

short in ;                /* incoming signal*/
AN_STATUS *state;                 /* status of filter at entry time*/
FILTERBANK *filter;       /*  filterbank used */
{
int n,k;
float sum ,d ;
COMPLEX temp1 , temp2 ;

for(n=0;n<filter->size;n++)
        {
        temp1= mult_complex( filter->weight[n] , state->error);
        temp2= sum_complex(state->HK[n],temp1);
        state->HK[n]  = mult_complex( filter->poles[n], temp2);
        }
sum=0.0;
for(n=0;n<filter->size;n++)
        {
        sum += state->HK[n].re;    /* complex conjugate 1/2 FB  */
        }

/* error is real by def because of the structure of the filterbank;
 we compute half of it only.we use complex for computation convenience. */

state->error.ima = 0.0;
d=in;           /* desired signal; input*/
state->error.re =d - sum ;  /* error signal is real  by def */
}




/***********************************/
/* speech enhancement routine      */
/***********************************/


void Speech_Enhancement(in ,anstate, anfilter, systate,syfilter,bufsize, envelop

short *in ;               /* incoming speech signal*/
AN_STATUS *anstate;       /* status of analysis filter at entry time*/
FILTERBANK *anfilter;     /* analysis filterbank used */
SY_STATUS *systate;       /* status of synthesis filter at entry time*/
FILTERBANK *syfilter;     /* synthesis filterbank used */
int  bufsize ;            /* length of in */
float  *envelope;         /* estimated envelope*/
float exponent;           /* for noise trap*/
float learn;              /*   slowfast*/
float whitener;           /*   a la bbn*/
SYNFIL *syin;             /* to input to synthesizer.*/
{
int n,k,delayed;
short out;                /* enhanced speech sample*/
float acc;                /* accumulated output*/
POLAR  sypolar;           /* input to synthesis filter in polar form*/
POLAR  anpolar;           /* output of analysis filter in polar form*/
COMPLEX  sycomp;          /* input to synthesis filter in complex form*/

for(k=0;k<bufsize;k++) /* for all samples in buffer*/
        {
        analysis_filter(in[k] ,anstate, anfilter ); /* first filter*/
```

```
        slowfast(envelope,anstate,estimate, epsilon, anfilter,learn);

        delayed=current -delay;
        if (delayed<0) delayed=MAXDELAY + delayed;
        for(n=0;n<syfilter->size;n++)
                {
                delayline.tap[current].HK[n] = anstate->HK[n];
                }
        gdnt(envelope,anfilter,estimate,&delayline.tap[delayed].HK[n],epsilon,wh
        for(n=0;n<syfilter->size;n++)
                {
                anpolar = transform_polar(delayline.tap[delayed].HK[n]);
                 /* to get phase*/
                sypolar.r = envelope[n];
                sypolar.angle= anpolar.angle;
                sycomp=transform_com(sypolar);
                syin->fil[n]=sycomp;
                }
        current++;
        if (current==MAXDELAY) current =0;      /* modulo arith*/
        acc=synthesis_filter(syin ,systate, syfilter );
        /* sum of outputs of synthesis filterbank*/
        out= (short)( 0.5 * acc);
        if  ((acc> MAXSHORT) || (acc< MINSHORT) )
                        printf(" error in normalization %f \n",acc);
        fwrite(&out,sizeof(short),1,f2);
        }
}


/************************************************/
/* initialization routines                      */
/************************************************/



void init_s(state)

/* used to initialize a filter recursion (zeroize for t<0).*/
SY_STATUS *state;
{
int j;
for(j=0;j<SIZE;j++)
        {
        state->sum[j].re=0;
        state->sum[j].ima=0;

        }
}


void init_a(state)

/* used to initialize a filter recursion (zeroize for t<0).*/
AN_STATUS *state;
{
int j;
state->error.re=0;
state->error.ima=0;
for(j=0;j<SIZE;j++)
        {
        state->HK[j].re=0;
        state->HK[j].ima=0;
        }
}
```

```
void init_apriori(epsilon)

float *epsilon;
{
int j;
for(j=0;j<SIZE;j++)
        {
        epsilon[j]= 3.1 ; /* SNR =5 db */
        }
}


void init_delay(ddelay)

DELAYLINE *ddelay;
{
int j;
for(j=0;j<MAXDELAY;j++)
        {
        for(k=0;k<SIZE;k++)
                {
                 ddelay->tap[j].HK[k].re = 0.0 ;
                 ddelay->tap[j].HK[k].ima = 0.0 ;
                }
        }
}

void init_env(envelope)

float *envelope;
{
int j;
for(j=0;j<SIZE;j++)
        {
        envelope[j]= 0.0 ;
        }
}

/*****************************************/
/* MAIN                                  */
/*****************************************/


main (argc,argv)
int argc;
char *argv[];
{
if ((argc < 7) || (argc>13) )
        {
        printf("usage : gdnt_d  fin fout noise nsamples exponent  overestimate l
        exit(0);
        }                   /* check for number only*/
open_input(argv[1]);
open_output(argv[2]);
f3=fopen(argv[3],"r+b");          /* noise file */
fread(estimate,sizeof(float),SIZE,f3);
nsamples=atol(argv[4]);
exponent=atoi(argv[5]);
overestimate=atof(argv[6]);
learn=atof(argv[7]);
whitener=atof(argv[8]);
delay=atoi(argv[9]);

for (ii=0L;ii<SIZE;ii++)
        {
```

```
        estimate[ii]= overestimate * estimate[ii];
        }
niterations=nsamples/BUFF;
analyse = &analysis;
an= &an_state ;
synthese = &synthesis;
sy= &syn_state ;
syin= &syinput;
selection_DFT(analyse);  /* select dft filter*/
selection_DFT(synthese);  /* select dft filter*/
init_a(an);                                /* initialize*/
init_s(sy);                                /* initialize*/
init_apriori(epsilon);
init_env(envelope);              /* estimated envelope */
init_delay(&delayline);
for (ii=0L;ii<niterations;ii++)
        {
        readblock(in,BUFF);
        Speech_Enhancement(in ,an, analyse,sy,synthese, BUFF ,envelope,exponent
        }
rest=nsamples-(niterations * BUFF);
if (rest>0)
        {
        readblock(in,rest);
        Speech_Enhancement(in ,an, analyse,sy,synthese ,rest ,envelope,exponent
        }
}
```

```
/*********************************************************/
/* gdnt_noise.c
   Speech enhancement  experiment using an analysis and a
   synthesis filterbank.
   The filterbank are linearly spaced and have
   128 filters. The filterbank is an Allpass resonator FB
   as developped by McGee.
   Noise estimate is calculated in this program.
                                                         */
/*********************************************************/

#include <stdio.h>
#include <math.h>
#include "complex2.h"           /*  complex routine library.*/
#define SIZE            128     /* filter bank size*/
#define FS              16000   /* sampling frequency */
#define BUFF    100000  /*  max buffer to read from noise file*/

/* typedef and structures.*/


typedef struct
        {
        int index;
        float wre;
        float wima;
        } TAB;                  /* table containing weights of FB*/

/*  filterbank structure */
typedef struct
        {
        COMPLEX poles[SIZE];    /* poles of filterbank*/
        COMPLEX weight[SIZE];   /* weight of filters */
        float freqs[SIZE];      /* resonant freqs of filterbank*/
        float wdig[SIZE];       /* freqs in radians of filterbank*/
        int size;               /* size of filter bank.*/
        } FILTERBANK;

/* status of  analysis filterbank*/
typedef struct
        {
        COMPLEX  error;                 /* sum of filters */
        COMPLEX HK[SIZE];               /* output of filters*/
        } AN_STATUS;                    /* status of filters*/



typedef struct
        {
        COMPLEX fil[SIZE];
        } SYNFIL;               /* data passed to synthezizer.*/

/* Global variables and constants */
FILE * f1;              /* input  file*/
FILE * f2;              /* noise file*/
long   ii;              /* counter */
int   k;                /* counter */
long  rest;             /* rest of the input files*/
long nsamples,niterations ;     /* counter variables */
float pi=3.141592658;   /* float is used for computational speed reasons */
float estimate[SIZE];   /* noise estimate */
short in[BUFF];         /* input signal from file*/
FILTERBANK *analise;    /* ptr  to analisys filterbank used.*/
FILTERBANK analisys;    /*  analisys filterbank used.*/
AN_STATUS *an ;         /* ptr to status of filtbank 1.*/
```

```
AN_STATUS an_state ;     /* status of filtbank 1.*/
float acc[SIZE][BUFF];   /* sum accumulation of filter outputs.*/
double sum[SIZE];        /* accumulation on time.*/
float estimate[SIZE];    /* noise estimate written to disk*/
float factor;            /* to multiply  noise input if needed. .*/

/***************************/
/* end of global variable */
/***************************/



void open_input(fichier)
        char *fichier;
{
        f1= fopen(fichier,"r+b");
}


void open_output(fichier)
char *fichier;
{
        f2= fopen(fichier,"w+b");
}


void readblock(buffer, length)
short *buffer; int length;
{
        fread(buffer,2,length,f1);
}     /* reads integers of 16 bits compatible the A/D */


void writeblock(buffer,  length)
short *buffer; int length;
{
        fwrite(buffer,2,length,f2);
}           /* writes integers of 16 bits compatible with the D/A */




selection_DFT(filter)
/* selection of parameters for  DFT filterbank*/
FILTERBANK *filter;
{
int i;
int size=SIZE;
filter->size=size;
for(i=0;i<(size);i++)
        {
        filter->freqs[i] = (float)i/(float)(2*size); /* digital freqs*/
        filter->wdig[i]=2.0*pi*(float)i/(float)(2*size); /* digital freqs in rad
        }
for(i=0;i<(size);i++)
        {
        filter->poles[i].re=cos(filter->wdig[i]);
        filter->poles[i].ima=sin(filter->wdig[i]);
        }

for(i=0;i<(size);i++)
        {
        filter->weight[i].re=2.0/(float)(2*size);
        filter->weight[i].ima=0.0;
        }
```

```
}
```

```
void analysis_filter(in ,state, filter,factor )

/* analisys filtering operation is done here.*/
/* no attempt is made to reduce computations.*/

short in ;                /* incoming signal*/
AN_STATUS *state;         /* status of filter at entry time*/
FILTERBANK *filter;       /*  filterbank used */
float factor;             /* to multiply incoming sequence.*/
{
int n,k;
float sum ,d ;
COMPLEX temp1 , temp2 ;

for(n=0;n<filter->size;n++)
        {
        temp1= mult_complex( filter->weight[n] , state->error);
        temp2= sum_complex(state->HK[n],temp1);
        state->HK[n]  = mult_complex( filter->poles[n], temp2);
        }
sum=0.0;
for(n=0;n<filter->size;n++)
        {
        sum += state->HK[n].re;   /* complex conjugate 1/2 FB  */
        }

/* error is real by def because of the structure of the filterbank;
 we compute half of it only.we use complex for computation convenience. */

state->error.ima = 0.0;
d= factor * in;                  /* desired signal; input*/
state->error.re =d - sum ;  /* error signal is real  by def */
}
```

```
void init_a(state)

/* used to initialize a filter recursion (zeroize for t<0).*/
AN_STATUS *state;
{
int j;
state->error.re=0;
state->error.ima=0;
for(j=0;j<SIZE;j++)
        {
        state->HK[j].re=0;
        state->HK[j].ima=0;
        }
}
```

```
main (argc,argv)
int argc;
char *argv[];
{
if (argc != 5)
        {
        printf("usage : qdnt_noise f1 f2 nsamples factor\n");
        exit(0);
        }
open_input(argv[1]);
open_output(argv[2]);
niterations=atol(argv[3]);
factor=atof(argv[4]);
analise = &analisys;
an= &an_state ;
selection_DFT(analise );  /* select log filter*/
init_a(an);                                /* initialize*/

readblock(in,niterations);
for(k=0;k<SIZE;k++)
        {
        sum[k]=0;
        }

for (ii=0L;ii<niterations;ii++)          /* for every time sample*/
        {
        analysis_filter(in[ii],an,analise,factor);
        for (k=0;k<SIZE;k++)             /* for every filter*/
                {
        acc[k][ii]=(an->HK[k].re * an->HK[k].re) + (an->HK[k].ima *an->HK[k].ima
                }
        }


for (ii=0L;ii<niterations;ii++)          /* for every time sample*/
        {
        for (k=0;k<SIZE;k++)             /* for every filter*/
                {
                sum[k] += acc[k][ii];    /* global variable with all the outputs*
                }
        }

for (k=0;k<SIZE;k++)
        estimate[k]=sum[k]/(double)niterations;
fwrite(estimate,sizeof(float),SIZE,f2);
}
```

```
/*************************************************************************
block.C
Short time spectral amplitude estimation
method for speech enhancement. A FFT overlap and add
method is used.


call:block  fin fout noise nsamples strategy overestimate parameters
Author : Luc Gagnon.
*************************************************************************/


#include <stdio.h>
#include <aplib.h>
#include <math.h>
#include "complex2.h"
#define HCINC   8               /* VA constants*/
#define VACINC  2
#define LENGTH  256             /* 512 * 1/16000hz = 32ms analysis frame*/
#define VSIZE   256
#define MIDSIZE 128
#define MAXSHORT  32767.0
#define MINSHORT  -32768.0
#define EXPLOSION       60      /*  used to prevent explosion of exp functions*/
#define MCCAULAY        1       /* McCaulay strategy ref: [1] */
#define POWER           2       /* Power subtraction method*/
#define WIENER          3       /* WIENER  filtering*/
#define EPHRAIM_III     4       /* EPHRAIM   method case III in paper */
#define EPHRAIM_V       7       /* EPHRAIM   method case V in paper */
#define MOORER          5       /* MOORER    method */
#define DNT             6       /* Non linear gain function*/
#define ID              8       /* to verify identity system*/
#define EGER            9       /* Eger non linear spectral gain*/
#define WHITEN          10      /* Wiener gain with whitener */
#define BBN             11      /* BBN method*/



/***********************************/
/* Global variables and constants */
/***********************************/


int     loglen, fftdst, fftsrc, ffttmp, coef, len; /* VA variables*/
FILE *f1;                       /* input  file*/
FILE *f2;                       /* output file*/
FILE *f3;                       /* noise file*/
int      k;                     /* counters for main */
COMPLEX fft[VSIZE];             /* dft of incoming block */
COMPLEX sig[VSIZE];
COMPLEX outidft[VSIZE];         /* output of inverse dft.*/
float   outfloat[VSIZE];        /* enhanced speech block*/
short   in[VSIZE]  ;            /* incoming speech*/
short   out[VSIZE];             /* enhanced speech block normalize and written t
float   win[VSIZE];             /* window coefficients.*/
short   slide_in[VSIZE];        /* overlapped data*/
float   in_win[VSIZE];          /* same after window.*/
float   phasef[VSIZE];          /* phase of incoming signal */
float   estimate[VSIZE];        /* estimate of noise power. */
float   fftpolar[VSIZE];        /*   abs value of first fft */
COMPLEX spectrum[VSIZE];
unsigned long   ndft;           /* number of dft */
unsigned long   nsamples;       /* number of samples*/
int     ii,n;                   /* counter */
float   out_1[VSIZE];           /* past block  of enhanced speech*/
float   out_2[VSIZE];           /* 2nd past block of  enhaned speech*/
```

```
float   outs[VSIZE] ;                   /* slided block written in float*/
float   pi = 3.141592658;
float   enhanp[VSIZE];                  /* estimated envelope of signal */
float   epsilon[VSIZE];                 /* apriori SNR for each channels*/


int strategy;                   /*  spectral modification strategy utilized.*/
float parameters[4];            /* function dependent*/
float beta;                     /* subtraction factor for power sub method*/
float alpha_m ;                 /* alpha for Moorer 's method*/
float r;                        /* exponent for "" */
float lambda_m ;                /*  threshold for ""*/
float qk ;                      /*  probability of speech absence :Ephraim case V */
float lambda;                   /* learning rate to recursive  estimation of a priori*/
float mc_aprio;                 /* a priori SNR fixed for McCaulay 's method */
float learn;                    /* learning rate for dnt method*/
float overestimate;             /* overestimate of noise variance factor*/
float nk[VSIZE];                /*  for Ephraim 5*/


/***************************/
/* end of global variable */
/***************************/


/***************************/
/* file functions          */
/***************************/

open_input(fichier)
char    *fichier;
{
        f1 = fopen(fichier, "r+b");
}


open_output(fichier)
char    *fichier;
{
        f2 = fopen(fichier, "w+b");
}


readblock(buffer, length)
short   *buffer;
int     length;
{
        fread(buffer, sizeof(short), length, f1);
}


writeblock(buffer, length)
short   *buffer;
int     length;
{
        fwrite(buffer, sizeof(short), length, f2);
}


/***************************/
/* window functions        */
/***************************/


make_hanning(length, win)
int     length;
```

```
float   *win;
/* makes coeff for Hanning window*/
{
        int     i;
        int     half;
        half = length / 2;
        for (i = 0; i <= length - 1; i++) {
                win[i] = 1 + cos(2 * pi * (i - half) / length);
        }
}


make_hamming( length, win)
int     length;
float   *win;
/* makes coeff for Hamming window*/
{
        int     i;
        int     half;
        half = length / 2;
        for (i = 0; i <= length - 1; i++) {
                win[i] = 0.54 + 0.46 * cos(2 * pi * (i - half) / length);
        }
}

make_rect( length, win)
/* trivial */
int     length;
float   *win;
/* makes coeff for  rectangular   window*/
{
        int     i;
        int     half;
        half = length / 2;
        for (i = 0; i <= length - 1; i++) {
                win[i] = 1.0;
        }
}



window_data(in, win, in_win, length)
short   *in;
float   *win;
float   *in_win;
int     length;
/* multiply window with data*/
{
        int     i;
        for (i = 0; i <= length - 1; i++) {
                in_win[i] =  (float)in[i] * win[i];
        }
}



/****************************************/
/* Complex arithmetic routines          */
/****************************************/


normalize( in, out, length )
float   *in;
short   *out;
int     length;
{
```

```
        float factor=1.0;
        int     i;
        float   temp;
        for (i = 0; i <= length - 1; i++) {
                if ( in[i] > MAXSHORT || in[i] < MINSHORT )
                        printf(" error in normalization of output \n\n");
                out[i] = (short) (in[i] * factor);
        }

}  /* normalize and transform a float array in a short array*/


phase(npoints, phasef,fft)
int     npoints;
float   *phasef;
COMPLEX *fft;
{
        float   pi = 3.141592658;
        int     i;
        for (i = 0; i <= npoints - 1; i++) {
                if (fft[i].re == 0 )
                        if (fft[i].ima < 0)
                                phasef[i] = 3.0 * pi / 2.0; /* no division by 0
                        else
                                phasef[i] = pi / 2;
                if (fft[i].ima == 0)
                        if (fft[i].re < 0)
                                phasef[i] = pi;
                        else
                                phasef[i] = 0;

                phasef[i] = atan2( fft[i].ima, fft[i].re);

        }

} /* calculate phase of DFT */


square(length, enhpower, enhamp)
int     length;
float   *enhpower;
float   *enhamp;
{
        int     i;
        for (i = 0; i <= length - 1; i++)
                enhamp[i] = sqrt(enhpower[i]);
}


abs_c(length, abso,fft)
int     length;
float   *abso;
COMPLEX fft[];
{
        int     i;
        float   poww;
        for (i = 0; i <= length - 1; i++) {
                poww = (fft[i].re * fft[i].re ) + ( fft[i].ima  * fft[i].ima);
                abso[i] = sqrt(poww);
        }
}


restorephase( length, enhamp,  phasef,spectrum)
```

```
int     length;
float   *enhamp;
float   *phasef;
COMPLEX *spectrum;
{
        int     i;
        for (i = 0; i <= (length - 1); i++) {
                spectrum[i].re = enhamp[i] * cos(phasef[i]);
                spectrum[i].ima = enhamp[i] * sin(phasef[i]);
        }
}


symmetry( size, enhamp)
int     size;
float   *enhamp;
{
        int     i;
        int midsize;
        midsize =size/2;
        for (i = midsize; i <= (size - 1); i++) {
                enhamp[i]= enhamp[size-i];/* odd symmetry*/
        }
}

/************************************/
/* Va FFT routines                  */
/************************************/


cfft(signal, fft,  len)
COMPLEX *signal;
COMPLEX *fft;
int     len;
{
        /*   Copy the input array, signal, from the host   */
        maplodcfv(signal, HCINC, fftsrc, VACINC, len);
        mapsyncmath(-1, VA0);

        /*   Perform a complex FFT on the input data  */
        mapfftnc(fftsrc, VACINC, coef, VACINC, ffttmp, VACINC, len);
        mapsyncdma(-1, VA0);

        /*   Determine the location of the results of  last step */
        fftdst = (loglen & 1) ? ffttmp : fftsrc;

        /*   Hold up host while result is being Stored  */
        mapstrcfv(fftdst, VACINC, fft, HCINC, len);
        mapbwaitdma(VA0);


        return(0);
}


icfft(spectrum, outidft,  len)
COMPLEX *spectrum;
COMPLEX *outidft;
int     len;
{
        ;*   Copy the input array, spectrum, from the host   */
        maplodcfv(spectrum, HCINC, fftsrc, VACINC, len);
        mapsyncmath(-1, VA0);


        /*   Take inverse transform & sync on math completion */
```

```
        mapifftnc(fftsrc, VACINC, coef, VACINC, ffttmp, VACINC, len);
        mapsyncdma(-1, VA0);


        /*   Determine the location of the results of last step */
        fftdst = (loglen & 1) ? ffttmp : fftsrc;

        /*   Hold up host while result is being Stored  */
        mapstrcfv(fftdst, VACINC, outidft, HCINC, len);
        mapbwaitdma(VA0);


        return(0);
}



init_VA()
/* initialize Vector Accelerator */

{
        len = VSIZE;
        mapinitva(1, 1, 0);
        /*      Test the exponent power of 2  */
        loglen = mapilog2(len);

        /*    Check the Validity of len   */
        if (len > 2048 || len != 1 << loglen) {
                printf("Length error!\n");
                return(-1);
        }

        /* Allocate vectors in VA memory */
        fftsrc = mapmalloc(len * 2, len * VACINC, VA0);
        ffttmp = mapmalloc(len * 2, 0, VA0);
        coef = mapmalloc(len, 0, VA0);
        /*   Generate the Coefficient Table  */
        mapffttab(coef, loglen);
}



/*************************************************/
/* initialization routines                       */
/*************************************************/


void init_apriori(epsilon)

float *epsilon;
{
int j;
for(j=0;j<VSIZE;j++)
        {
        epsilon[j]= 1.1 ; /* SNR =0 db */
        }
}

void init_env(envelope)

float *envelope;
{
int j;
for(j=0;j<VSIZE;j++)
        {
        envelope[j]= 0.0 ;
```

```
        }
}


/*************************************************/
/* spectral amplitude  estimation routines     */
/*************************************************/


bbn(enhanp,midsize,noiseestimate,fftpolar,alpha,beta)
/* bbn method for spectral subtraction*/
float   *enhanp ;
int midsize;
float  *noiseestimate;
float *fftpolar ;
float   alpha , beta;

{
        int k;
        float   D[MIDSIZE+1];
        float   temp;
        for (k = 0; k <= midsize; k++) {
                D[k] = (fftpolar[k]* fftpolar[k]) - (alpha * noiseestimate[k]);
                temp = (beta * noiseestimate[k]) ;
                if (D[k] > temp )
                        enhanp[k] = D[k];
                else
                        enhanp[k] = temp;
                enhanp[k] = sqrt(enhanp[k]);
        }

}




float bessi1(x)
float x;
{
        float ax,ans;
        double y;

        if (x>EXPLOSION) x=EXPLOSION;           /* would explode otherwise*/
        if ((ax=fabs(x)) < 3.75) {
                y=x/3.75;
                y*=y;
                ans=ax*(0.5+y*(0.87890594+y*(0.51498869+y*(0.15084934
                        +y*(0.2658733e-1+y*(0.301532e-2+y*0.32411e-3))))));
        } else {
                y=3.75/ax;
                ans=0.2282967e-1+y*(-0.2895312e-1+y*(0.1787654e-1
                        -y*0.420059e-2));
                ans=0.39894228+y*(-0.3988024e-1+y*(-0.362018e-2
                        +y*(0.163801e-2+y*(-0.1031555e-1+y*ans))));
                ans *= (exp(ax)/sqrt(ax));
        }
        return x < 0.0 ? -ans : ans;
}

float bessi0(x)
float x;
{
        float ax,ans;
        double y;
```

```
        if (x>EXPLOSION) x=EXPLOSION;           /* would explode otherwise*/
        if ((ax=fabs(x)) < 3.75) {
                y=x/3.75;
                y*=y;
                ans=1.0+y*(3.5156229+y*(3.0899424+y*(1.2067492
                        +y*(0.2659732+y*(0.360768e-1+y*0.45813e-2)))));
        } else {
                y=3.75/ax;
                ans=(exp(ax)/sqrt(ax))*(0.39894228+y*(0.1328592e-1
                        +y*(0.225319e-2+y*(-0.157565e-2+y*(0.916281e-2
                        +y*(-0.2057706e-1+y*(0.2635537e-1+y*(-0.1647633e-1
                        +y*0.392377e-2)))))));
        }
        return ans;
}

/*************************/
/* McCaulay functions    */
/*************************/

float fp(priori,posteriori)

/* see Mathcad simulation for explanation of this function.*/
float priori;
float posteriori;
{
float ans,a;
a=  2* sqrt( priori* posteriori) ;
ans=(exp(-1.0 *priori) ) * bessi0( a );

return(ans);
}

float prob1(priori,posteriori)
/* for two-state model*/
float priori;
float posteriori;
{
float ans,f;
f=fp(priori,posteriori);
ans=f/(f+1.0); /* prob( H1|V) */
return(ans);
}


mccaulay(envelope,size,estimate, epsilon ,fftpolar )
/*  estimate the envelope of the signal according to McCaulay's
strategy */
/* McCaulay method ref : [1] */
float *envelope ;        /* envelope to estimate*/
int size;               /* size*/
float *estimate;        /* estimate of noise*/
float *epsilon;         /* a priori SNR factor*/
float *fftpolar;        /* state of filter*/

{
int i;
float V_2 ;     /* instantaneous power of channel */
float a ;       /* temporary variable*/
float max_likely; /* maximum likelyhood estimator*/
float ph1_v; /* probability of speech signal */
for(i=0;i<=size;i++)
/* middle bin has to be estimated/ 0-> MIDSIZE inclusively*/
        {     /* for all channels*/
        V_2=(fftpolar[i] * fftpolar[i]);    /* instant energy*/
```

```
        /* maximum likelyhood estimator is computed..*/
        if ( ( a = V_2 - estimate[i]) > 0 )
                max_likely= 0.5 * (fftpolar[i] + sqrt(a) );
        else    max_likely= 0.5 * fftpolar[i];

        /* probability is computed..*/
        ph1_v=prob1(epsilon[i],V_2/estimate[i]);

        envelope[i] = max_likely * ph1_v;       /* estimated envelope*/
        }
}


/*********************************/
/* end of McCaulay functions   */
/*********************************/


/*********************************/
/*  Ephraim functions          */
/*********************************/


float GMMSE(priori,posteriori)
/* MMSE gain as defined by Ephraim */

float priori;
float posteriori;

{
float vk,a,Y,b,ans;
vk=  (posteriori * priori/ (priori +1.0));
if (vk >= EXPLOSION)
        {
        ans= priori /(priori + 1.0);
        }/* when SNR >>> Wiener gain*/
else
        {

        a = sqrt(vk);
        Y= sqrt(pi) / 2.0;
        b= ((1.0 + vk ) * bessi0(vk/2.0) ) +  ( vk * bessi1(vk/2.0) );
        if (posteriori == 0 ) posteriori =  0.0001;
        ans= Y * ( a / posteriori) * exp( -1.0 * vk /2.0 ) * b;
        }
return(ans);

}


Ephraim_III(envelope,size,estimate, epsilon ,fftpolar )

/*  estimate the envelope of the signal according to Epraim's
strategy  : case III see ref.*/
float *envelope ;        /* envelope to estimate*/
int size;
float *estimate;        /* estimate of noise*/
float *epsilon;         /* a priori SNR factor*/
float *fftpolar;        /* state of filter*/

{
int i;
float R2 ;    /* instantaneous power of channel */
float posteriori ;
for(i=0;i<=size;i++)
```

```
        {
        R2=(fftpolar[i] * fftpolar[i]);    /* instant energy*/
        posteriori= R2 / estimate[i];
        envelope[i]= fftpolar[i] * GMMSE(epsilon[i],posteriori);
                        /* estimated envelope*/
        }
}


Ephraim_V(envelope,size,estimate, nk ,fftpolar,qk )

/*  estimate the envelope of the signal according to Epraim's
strategy  : case V see ref.*/
float *envelope ;         /* envelope to estimate*/
int size;
float *estimate;          /* estimate of noise*/
float *nk;                /* a priori SNR factor multiplied by 1-qk*/
float *fftpolar;          /* state of filter*/
float qk;                 /* prob of speech signal absence*/

{
int i;
float e;        /* a priori SNR*/
float uk;
float vk;
float ff;
float likelyhood;
float posteriori;
float R2 ;    /* instantaneous power of channel */
for(i=0;i<=size;i++)
        {    /* for all channels*/
        R2=(fftpolar[i] * fftpolar[i]);    /* instant energy*/
        posteriori= R2 / estimate[i];
        uk= (1.0 -qk) / qk;
        e=  nk[i] * (1.0/ (1.0-qk) );
        vk=  (posteriori * e/ (e +1.0));
        if (vk >EXPLOSION )
                likelyhood=1.0 ;        /* would explode otherwise*/
        else
                {
                ff= uk * exp(vk ) / (1.0 + e);
                likelyhood= ff/ (1.0 + ff);
                }
        envelope[i] = fftpolar[i] * likelyhood  * GMMSE(e,posteriori);
                        /* estimated envelope*/
        }
}



update_epsilon(envelope,fftpolar,estimate, epsilon, size,learning_rate)
/* decision oriented evaluation of a priori SNR*/


float *envelope;          /* estimated envelope of signal*/
float *fftpolar;
float *estimate;          /* noise estimates*/
float *epsilon;           /* a priori SNR's*/
int size;
float learning_rate;      /* for recursive estimation*/
{
int i;
float a,b,v2,P;
```

```
for(i=0;i<= size;i++)
        {
        v2 = fftpolar[i] * fftpolar[i];
        P= (v2/estimate[i]) - 1.0;
        if (P<= 0) P=0.0;
        a= (1.0 - learning_rate) * P;   /* present measure contribution*/
        b= (learning_rate) * (envelope[i] * envelope[i] / estimate[i] );
        epsilon[i]= a+b;
        } /* decision oriented a priori estimation ref:[2]*/

}


/*********************************/
/* end of Ephraim functions     */
/*********************************/



Moorer_est(envelope,size,estimate,fftpolar,alpha,r,kk)
/*  estimate the envelope of the signal according to MOORER
strategy */
float *envelope ;         /* envelope to estimate*/
int size;
float *estimate;         /* estimate of noise*/
float *fftpolar;
float alpha;             /* see reference []*/
float r;                 /*exponent of nonlinear function*/
float kk;                /*  multiplicative factor to get threshold from noise
                           standard deviation*/

{
int i;
float P,F ;      /* temporary variables */
float gain;      /* non linear gain*/
for(i=0;i<=size;i++)
        {    /* for all channels*/
        P= fftpolar[i] /(fftpolar[i] + (kk* sqrt(estimate[i]) ) );
        F= pow(P,2.0 * r);
        gain= alpha + (1.0 -alpha) *F;
        /* non-linear gain is computed..*/

        envelope[i] = fftpolar[i] * gain;       /* estimated envelope*/
        }
}



Eger(envelope,size,estimate,fftpolar,N,kk)
/*  estimate the envelope of the signal according to Eger
strategy */
float *envelope ;         /* envelope to estimate*/
int size;
float *estimate;         /* estimate of noise*/
float *fftpolar;
float N;                 /*exponent of nonlinear function*/
float kk;                /*  multiplicative factor to get threshold from noise
                           standard deviation*/

{
int i;
float P,F ;      /* temporary variables */
float gain;      /* non linear gain*/
for(i=0;i<=size;i++)
        {    /* for all channels*/
        P= fftpolar[i] /((kk* sqrt(estimate[i]) ) );
```

```
        F= pow(P,N);
        gain= F/(1.0+F);
        envelope[i]=fftpolar[i] * gain;
        }
}


id(envelope,size,estimate,fftpolar)
/*  id system verification */


float *envelope ;         /* envelope to estimate*/
int size;
float *estimate;         /* estimate of noise*/
float *fftpolar;
{
int i;
float gain;      /* non linear gain*/
for(i=0;i<=size;i++)
        {    /* for all channels*/
        gain= 1.0;
        envelope[i]=fftpolar[i] * 1.0 ;
        }
}
```

```
power_subtraction(envelope,size,estimate,fftpolar,beta)
/* power subtraction method for speech enhancement */

float *envelope ;         /* envelope to estimate*/
int size;
float *estimate;         /* estimate of noise*/
float *fftpolar;
float beta;              /* factor to subtract overestimate of noisepower.*/
{
int i;
float a,b ;      /* temporary variables */
float gain;      /* channel gain*/
for(i=0;i<=size;i++)
        {    /* for all channels*/
        a= (fftpolar[i] * fftpolar[i]) / (  estimate[i]);
        if (a==0) a=.0000001;
        b=  1.0 - (beta * (1.0/a));
        if (b <=0)
                gain =0.0;
        else
                gain =sqrt( b);
        /* power_subtraction gain is computed..*/

        envelope[i] = fftpolar[i] * gain;       /* estimated envelope*/
        }
}


wiener_est(envelope,size,estimate,fftpolar,epsilon)
/* Wiener filtering approach to speech enhancement */

float *envelope ;         /* envelope to estimate*/
int size;
```

```
float *estimate;        /* estimate of noise*/
float *fftpolar;
float *epsilon;         /* a priori SNR 's */
{
int i;
float gain;     /* channel gain*/
for(i=0;i<=size;i++)
        {    /* for all channels*/
        gain = epsilon[i] / ( epsilon[i] +1.0 );
        /* gain is computed..*/

        envelope[i] = fftpolar[i] * gain;       /* estimated envelope*/
        }
}


whiten(envelope,size,estimate,fftpolar,epsilon,whitener)
/* Wiener filtering approach to speech enhancement */

float *envelope ;        /* envelope to estimate*/
int size;
float *estimate;        /* estimate of noise*/
float *fftpolar;
float *epsilon;         /* a priori SNR 's */
float whitener;
{
int i;
float gain,a;   /* channel gain*/
for(i=0;i<=size;i++)
        {    /* for all channels*/
        gain =  epsilon[i] / ( epsilon[i] +1.0 ) ;
        /* gain is computed..*/
        envelope[i] = fftpolar[i] * gain;       /* estimated envelope*/
        if (envelope[i]<= (a=sqrt(whitener * estimate[i])) )
                envelope[i]=a;
        }
}




slowfast(envelope,fftpolar,estimate, epsilon, size,learning_rate)
/* evaluation of a priori SNR with the dnt recursion*/


float *envelope;        /* estimated envelope of signal*/
float *fftpolar;
float *estimate;        /* estimate of noise*/
float *epsilon;         /* a priori SNR's*/
int size;
float learning_rate;    /* for recursive estimation*/
{
int i;
float a,v2,P,T;
for(i=0;i<= size;i++)
        {
        v2 = fftpolar[i] * fftpolar[i];
        P= (v2/estimate[i])-1  ; /* try sn-1/lamb*/
        a= (1.0 - learning_rate) * (P-epsilon[i]) + epsilon[i];
        if(  a<= 0) a=0.0000001;
        epsilon[i]= a;
        }
}
```

```
dnt(envelope,size,estimate,fftpolar, priori)
/* dynamic noise trap  filtering approach to speech enhancement */

float *envelope ;         /* envelope to estimate*/
int size;
float *estimate;         /* estimate of noise*/
float *fftpolar;
float *priori;           /* a priori SNR 's */
{
int i;
float gain;     /* channel gain*/
for(i=0;i<=size;i++)
        {    /* for all channels*/
        gain = priori[i] / ( priori[i] +1.0 );
        envelope[i] = fftpolar[i] * gain;       /* estimated envelope*/
        }


}




main (argc, argv)
int     argc;
char    *argv[];
{
if ((argc < 7) || (argc>13) )
        {
        printf(" usage :block  fin fout noise nsamples strategy overestimate par
        exit(0);
        }
open_input(argv[1]);
open_output(argv[2]);
f3 = fopen(argv[3], "r+b");
fread(estimate, sizeof(float), VSIZE, f3);
/* noise estimate is produced in another program*/

nsamples=atol(argv[4]);
strategy=atoi(argv[5]);
overestimate=atof(argv[6]);
parameters[0]=atof(argv[7]);
parameters[1]=atof(argv[8]);
parameters[2]=atof(argv[9]);    /* method dependent parameters*/

for (ii=0L;ii<VSIZE;ii++)
        {
        estimate[ii]= overestimate * estimate[ii];
        }
init_apriori(epsilon);
init_apriori(nk);                /* for EPHRIAM case V*/
init_env(enhanp);                /* estimated envelope */
ndft = nsamples / MIDSIZE;
make_hanning(VSIZE, win) ;     /* make window coeffi. once for all*/
init_VA();
for(ii=0;ii<VSIZE;ii++)
        epsilon[ii]=1.0;


readblock(in, VSIZE);   /* initial fill */
for (ii = 1L; ii <= ndft; ii++)
```

```
{
if (ii == 1L)
        {
        for (k = 0; k <= (MIDSIZE - 1); k++)
                {
                out_1[k] = 0;
                out_2[k] = 0;
                }
        }
if ((ii % 2L) == 0)  /* when even  window 50 % overlapp  */
        {
        readblock(in, VSIZE);
        for (k = 0; k <= (MIDSIZE - 1); k++)
                        slide_in[k] = slide_in[k+MIDSIZE];

        for (k = (MIDSIZE); k <= (VSIZE - 1); k++)
                        slide_in[k] = in[k-(MIDSIZE)];

        }
else
        for (k = 0; k <= (VSIZE - 1); k++)
                        slide_in[k] = in[k];


window_data(slide_in, win, in_win, VSIZE);
for (k = 0; k <= (VSIZE - 1); k++)
        {
        sig[k].re = in_win[k];
        sig[k].ima = 0.0;                /* cfft wants a complex array*/
        }
cfft(sig, fft, VSIZE);          /* fft taken on windowed data*/
abs_c(VSIZE, fftpolar,fft);     /* absolute value taken on data to get
                                   envelope */
phase( VSIZE, phasef,fft);      /* phase is kept for further use*/



switch (strategy) {
case WIENER :
        lambda=parameters[0];   /* lambda=0.98  */
        update_epsilon(enhanp,fftpolar,estimate,epsilon,MIDSIZE,lambda);
        wiener_est(enhanp,MIDSIZE,estimate,fftpolar,epsilon);
        break;
case WHITEN :
        lambda=parameters[0];   /* lambda=0.98  */
        alpha_m= parameters[1]; /* minimum gain : whitening factor*/
        update_epsilon(enhanp,fftpolar,estimate,epsilon,MIDSIZE,lambda);
        whiten(enhanp,MIDSIZE,estimate,fftpolar,epsilon,alpha_m);
        break;
case POWER :
        beta = parameters[0];
        power_subtraction(enhanp,MIDSIZE,estimate,fftpolar,beta);
        break;
case EPHRAIM_III :
        lambda=parameters[0];   /* lambda=0.98  */
        update_epsilon(enhanp,fftpolar,estimate,epsilon,MIDSIZE,lambda);
        Ephraim_III(enhanp,MIDSIZE,estimate, epsilon ,fftpolar ) ;
        break;
case EPHRAIM_V :
        lambda=parameters[0];   /* lambda=0.98  */
        qk= parameters[1];
        update_epsilon(enhanp,fftpolar,estimate,nk,MIDSIZE,lambda);
        Ephraim_V(enhanp,MIDSIZE,estimate, nk ,fftpolar,qk );
        break;
case MOORER :
```

```
            alpha_m= parameters[0]; /* minimum gain */
            r=parameters[1] ;
            lambda_m = parameters[2]; /*  factor to multiply estimate*/
            Moorer_est(enhanp,MIDSIZE,estimate,fftpolar,alpha_m,r,lambda_m);
            break;
      case EGER :
            r=parameters[0] ;
            lambda_m = parameters[1]; /* threshold factor*/
            Eger(enhanp,MIDSIZE,estimate,fftpolar,r,lambda_m);
            break;
      case MCCAULAY :
            mc_aprio=parameters[0];          /* =31 */
            if (k==0) for(n=0 ;n<MIDSIZE; n++) epsilon[n]=mc_aprio;
            mccaulay(enhanp,MIDSIZE,estimate,epsilon,fftpolar);
            break;
      case DNT :
            learn=parameters[0];
            slowfast(enhanp,fftpolar,estimate, epsilon, MIDSIZE,learn);
            dnt(enhanp,MIDSIZE,estimate,fftpolar, epsilon);
            break;
      case ID :
            id(enhanp,MIDSIZE,estimate,fftpolar);
            break;
      case BBN :
            beta = parameters[0]; /* ovwer estimate*/
            alpha_m= parameters[1]; /*  whitening factor*/
            bbn(enhanp,MIDSIZE,estimate,fftpolar,beta,alpha_m);
            break;
      default :       break;
                      } /* switch*/

      symmetry(VSIZE,enhanp);

      restorephase(VSIZE, enhanp , phasef,spectrum );
                  /* result in variable spectrum*/
      icfft(spectrum, outidft, VSIZE);
      for (k = 0; k < VSIZE; k++)
            {
            outfloat[k] = (1.0 / ( 2.0 * (float)VSIZE)) *  outidft[k].re;
            }/* factor 2 because of hanning or hamming window and overlapp*/

         /* overlap and add */
      if ((ii % 2) == 0) /* exit window is cumulated at even counts*/
            {
            for (k = 0; k <= (MIDSIZE - 1); k++)
                  {
                  outs[k] = out_2[k+MIDSIZE] + out_1[k];
                  }
            for (k = MIDSIZE; k <= (VSIZE - 1); k++)
                  {
                  outs[k] = out_1[k] + outfloat[k-MIDSIZE];
                  }
            normalize(outs, out, VSIZE);
            writeblock(out, VSIZE);
            }
      for (k = 0; k <= VSIZE - 1; k++)        /* for any count*/
            {
            out_2[k] = out_1[k];
            out_1[k] = outfloat[k];
            }

      }
mapfreeva(VA0);
}
```

```
/*******************************************************
noise_stsa.C
noise spectrum estimation program for the  STSA program.


 Author : Luc Gagnon.
*******************************************************/


#include <stdio.h>
#include <apiib.h>
#include <math.h>
#define HCINC   8
#define VACINC  2
#define LENGTH  256
#define PI      3.1415926
#define VSIZE   256
#define MIDSIZE 128
#define BIG     2000



/* typedef and structures.*/

typedef struct
        {
        float  re,
        ima;
        } COMPLEX;

typedef struct
        {
        float  r,
        angle;
        } POLAR;

/* Global variables and constants */


int loglen, fftdst,fftsrc,ffttmp,coef,len; /* VA variables*/
FILE * f1;            /* input  file*/
FILE * f2;            /* output file*/
int j,k;                 /* counters for main */
COMPLEX fft[VSIZE];      /* dft of incoming block */
COMPLEX sig[VSIZE];
short in[VSIZE]  ;        /* incoming speech*/
float win[VSIZE];        /* window coefficients.*/
short slide_in[VSIZE];     /* overlapped data*/
float in_win[VSIZE];   /* same after window.*/
float noiseestimate[VSIZE];      /* estimate of noise power. */
double nois[BIG][VSIZE];                 /*  noise power of every windows */
float fftpolar[VSIZE];           /*   abs value of  fft */
COMPLEX spectrum[VSIZE];
unsigned long ndft;              /* number of dft */
unsigned long nsamples;
int  ii;                  /* counter */
float pi=3.141592658;
float y;         /*  exponent factors*/
float py[VSIZE];         /* power */
float kk;
float sum=0.0;
/*************************/
/* end of global variable */
/*************************/
```

```
open_input(fichier)
char *fichier;
{
        f1= fopen(fichier,"r+b");
}


open_output(fichier)
char *fichier;
{
        f2= fopen(fichier,"w+b");
}


readblock(buffer, length)
short *buffer; int length;
{
        fread(buffer,sizeof(short),length,f1);
}

writeblock(buffer, length)
short *buffer; int length;
{
        fwrite(buffer,sizeof(short),length,f2);
}

POLAR com_polar(x)
COMPLEX x;

{
POLAR res;
res.r=sqrt(( x.re * x.re) + ( x.ima * x.ima ) );
if (x.re==0)
        {
        if (x.ima ==0) res.angle=0;
        else
                if (x.ima < 0 ) res.angle= 3.0 *pi/2.0;
                else res.angle=pi/2.0;
        }

else    res.angle= atan2(x.ima,x.re);
return(res);
}


COMPLEX polar_com(x)
POLAR x;
{
COMPLEX res;
res.re= x.r * cos(x.angle);
res.ima= x.r * sin(x.angle);
return(res);
}

POLAR exp_com(x,n)
POLAR x; float n;
{
POLAR res;
res.angle= x.angle * n ;
res.r= pow(x.r,n);

return(res);
}
```

```
COMPLEX conjugate(x)

COMPLEX x;

{
COMPLEX y;
y.re=x.re;
y.ima= -1 * x.ima;
return(y);
}


make_hanning(length, win)
int length; float *win;
/* makes coeff for Hanning window*/
{
int i;
int half;
half=length/2;
for(i=0;i<=length-1;i++)
        {
        win[i]=1+cos(2*pi*(i-half)/length);
        }
}

make_rect(length, win)
int length; float *win;
/* makes coeff for trivial rectangular window*/
{
int i;
int half;
half=length/2;
for(i=0;i<=length-1;i++)
        {
        win[i]=1.0 ;
        }
}



make_hamming( length, win)
int length; float *win;
/* makes coeff for Hamming window*/
{
int i;
int half;
half=length/2;
for(i=0;i<=length-1;i++)
        {
        win[i]=0.54 + 0.46 * cos(2*pi*(i-half)/length);
        }
}


window_data(in,win,in_win, length)
short *in;float *win;float *in_win;int length;
/* multiply window with data*/
{
int i;
for(i=0;i<=length-1;i++)
        {
        in_win[i]=  (float)in[i] * win[i];
        }
}
```

```
abs_c(length, abso)
int length;  float *abso;
{
 int i;
 float poww;
 for(i=0;i<=length-1;i++)
        {
        poww=(fft[i].re * fft[i].re ) +( fft[i].ima  * fft[i].ima);
        abso[i]=sqrt(poww);
        }
}


cfft(signal,fft,  len)
     COMPLEX *signal;
     COMPLEX *fft;
     int len;
{
    /*   Copy the input array, signal, from the host   */
     maplodcfv(signal, HCINC, fftsrc, VACINC, len);
     mapsyncmath(-1,VA0);

    /*   Perform a complex FFT on the input data  */
     mapfftnc(fftsrc, VACINC, coef, VACINC, ffttmp, VACINC, len);
     mapsyncdma(-1,VA0);

    /*   Determine the location of the results of step [6] */
     fftdst = (loglen & 1) ? ffttmp : fftsrc;

    /*   Hold up host while result is being Stored  */
     mapstrcfv(fftdst, VACINC, fft, HCINC, len);
     mapbwaitdma(VA0);


     return(0);
}







main ( argc,argv)
int argc;
char *argv[];

{
if (argc != 5) { printf("usage: block_noise filein fileout nsamples k \n\n"); ex
open_input(argv[1]);
open_output(argv[2]);
y=2.0;  /* variance is estimated.*/
len=VSIZE;
/* initialize VA*/
mapinitva(1,1,0);
/*      Test the exponent power of 2  */
loglen = mapilog2(len);
```

```
/*    Check the Validity of len    */
if (len > 2048 || len != 1 << loglen) {
       printf("Length error!\n");
          return(-1);
     }

/* Allocate vectors in VA memory */
fftsrc = mapmalloc(len*2,len*VACINC,VA0);
ffttmp = mapmalloc(len*2,0,VA0);
coef = mapmalloc(len,0,VA0);
/*   Generate the Coefficient Table  */
mapffttab(coef, loglen);


nsamples=atoi(argv[3]);
kk=atof(argv[4]);                    /* to weight the power */
ndft=nsamples/MIDSIZE;
make_hanning(VSIZE,win) ;      /* make window coeffi. once for all*/
readblock(in,VSIZE);   /* initial fill */
for (ii=1L;ii<=ndft;ii++)
        {
        if ((ii%2L)==0)  /* when even */
        /* window 50 % overlapp  */
                {
                readblock(in,VSIZE);
                for (k=0;k<=(MIDSIZE-1);k++)
                        {
                        slide_in[k]=slide_in[k+MIDSIZE];
                        }
                for (k=(MIDSIZE);k<=(VSIZE-1);k++)
                        {
                        slide_in[k]=in[k-(MIDSIZE)];
                        }
                }
        else
                for (k=0;k<=(VSIZE-1);k++) { slide_in[k]=in[k];}

        window_data(slide_in,win,in_win,VSIZE);
        for (k=0;k<=(VSIZE-1);k++)
                {
                sig[k].re=in_win[k];
                sig[k].ima=0;              /* cfft wants a complex array*/
                }
        cfft(sig,fft,VSIZE);               /* fft taken on windowed data*/
        abs_c(VSIZE,fftpolar);             /* absolute value taken on data*/
        for (k=0;k<=(VSIZE-1);k++)
                {
                py[k]=pow(fftpolar[k],y);  /* y =2 */
                nois[ii][k]=py[k]*(kk * kk); /* kk is a factor to utilize the
                                                program addfiles /. */
                }
        }

 sum=0;
 for (k=0;k<=(VSIZE-1);k++)
        {
        noiseestimate[k]=0;
        for (ii=1;ii<=ndft;ii++)
                {
                noiseestimate[k] += nois[ii][k];
                }
        noiseestimate[k]= noiseestimate[k]/ndft;
        /*printf(" %u  %f \n\n",k,noiseestimate[k]);*/
        sum += noiseestimate[k];
```

```
        }/*avg of var*/
printf(" sum is = %f \n ",sum);
fwrite(noiseestimate,sizeof(float),VSIZE,f2);
mapfreeva(VA0);
}
```

```
/*****************************************************
subceps.c
Pseudo-Cepstral subtraction method for speech enhancement.
The method is a version of the INTEL method developed
for the RADC .
call : subceps filein fileout noisefile samples  alpha .

        samples = number of samples to process.
        alpha = subtraction factor

All the  other parameters are set to correspond to the SEU
(speech Enhancement Unit) given the information available.
Therefore, rootcompression =0.5 for all bins
and  .75 for bin [0].
Window length is about 64 ms.


 Author : Luc Gagnon.
*****************************************************/


#include <stdio.h>
#include <aplib.h>
#include <math.h>
#include "complex2.h"

#define HCINC           8
#define VACINC          2
#define LENGTH          1024
#define VSIZE           1024
#define MIDSIZE         512
#define ROOTCOMP        0.5
#define BIN0COMP        0.75
#define ROOTDECOMP      2.0
#define BIN0DECOMP      1.3333333
#define MAXSHORT        32767.0



/***********************************/
/* Global variables and constants */
/***********************************/


int loglen, fftdst,fftsrc,ffttmp,coef,len; /* VA variables*/
FILE * f1;              /* input  file*/
FILE * f2;              /* output file*/
FILE * f3;              /* noise file*/
int j,k;                /* counters for main */
COMPLEX fft[VSIZE];     /* dft of incoming block */
COMPLEX sig[VSIZE];     /* complex array for FFT processing.*/
COMPLEX outidft[VSIZE]; /* output of inverse dft.*/
float outfloat[VSIZE];  /* enhanced speech block*/
float max;              /* max value used for normalise*/
short in[VSIZE]  ;      /* incoming speech*/
short out[VSIZE];       /* enhanced speech block normalised and written to disk*
float win[VSIZE];       /* window coefficients.*/
short slide_in[VSIZE];  /* overlapped data*/
float in_win[VSIZE];    /* same after window.*/

float phasef[VSIZE];            /* phase of incoming signal */
float fftpolar[VSIZE];          /*   abs value of first fft */
POLAR fftpolar2[VSIZE];         /*   POLAR value of second fft */
COMPLEX spectrum[VSIZE];        /* spectrum of incoming sequence*/
COMPLEX cepstrum[VSIZE];        /* pseudocepstrum*/
COMPLEX encepstrum[VSIZE];      /* pseudocepstrum*/
```

```
unsigned long ndft;                    /* number of dft */
unsigned long nsamples;
int  ii;                               /* counter */
float out_1[VSIZE];                    /* past block  of enhanced speech*/
float out_2[VSIZE];                    /* 2nd past block of  enhaned speech*/
float outs[VSIZE] ;                    /* slided block written in float*/
float pi=3.141592658;
float enabs[VSIZE];                    /* enhanced  cepstrum*/
COMPLEX estimate[VSIZE];               /* noise cepstrum envelope estimate*/
float overestimate;                    /* factor to multiply estimate*/


/***************************/
/* end of global variable */
/***************************/



open_input(fichier)
char *fichier;
{
        f1= fopen(fichier,"r+b");
}


open_output(fichier)
char *fichier;
{
        f2= fopen(fichier,"w+b");
}


readblock(buffer, length)
short *buffer; int length;
{
        fread(buffer,sizeof(short),length,f1);
}

writeblock(buffer, length)
short *buffer; int length;
{
        fwrite(buffer,sizeof(short),length,f2);
}



/***************************/
/* window functions        */
/***************************/


make_hanning(length, win)
int     length;
float   *win;
/* makes coeff for Hanning window*/
{
        int     i;
        int     half;
        half = length / 2;
        for (i = 0; i <= length - 1; i++) {
                win[i] = 1 + cos(2 * pi * (i - half) / length);
        }
}


make_hamming( length, win)
```

```
int     length;
float   *win;
/* makes coeff for Hamming window*/
{
        int     i;
        int     half;
        half = length / 2;
        for (i = 0; i <= length - 1; i++) {
                win[i] = 0.54 + 0.46 * cos(2 * pi * (i - half) / length);
        }
}


make_Bartlett( length, win)
/* makes coeff for Bartlett window*/

int length;
float *win;
{
int i;
int half;
half=length/2;
for(i=half;i<=length-1;i++)
        {
        win[i]=2.0 -(4.0 * (i-half)) /  length;
        }
win[0]=0;
for(i=1;i<half;i++)
        {
        win[i]=win[length-i];;
        }
}

make_rect( length, win)
/* trivial */
int     length;
float   *win;
/* makes coeff for  rectangular   window*/
{
        int     i;
        int     half;
        half = length / 2;
        for (i = 0; i <= length - 1; i++) {
                win[i] = 1.0;
        }
}



window_data(in, win, in_win, length)
short   *in;
float   *win;
float   *in_win;
int     length;
/* multiply window with data*/
{
        int     i;
        for (i = 0; i <= length - 1; i++) {
                in_win[i] =  (float)in[i] * win[i];
        }
}


/***************************/
/* Normalization functions */
/***************************/
```

```
float max_seq(in,length)
float *in; int length;

{
 float max,f,val;
 int i;
 max=0.0;
 for(i=0;i<=length-1;i++)
   {
   val=fabs(in[i]);
   if (val>=max) max=val;
   }
 return(max);
}



short maxx(in,length)
short *in; int length;

{
 short max,f,val;
 int i;
 max=0;
 for(i=0;i<=length-1;i++)
   {
   val=abs(in[i]);
   if (val>=max) max=val;
   }
 return(max);
}



normalise( in, out, length )
 float *in;
 short *out;
 int length;
{
 float max,f;
 int i;
 for(i=0;i<=length-1;i++)
        {
        if (in[i] > MAXSHORT ) printf("erreur nomalisation \n\n ");
        out[i] = (  in[i]) ;
        }
 }  /* transform a float array in a short array*/


/*********************************/
/* Complex arithmetic routines */
/*********************************/

phase(  npoints,fft, phasef)

int npoints;
COMPLEX *fft;
float *phasef;
{
  float pi=3.141592658;
  int i;
  for(i=0;i<=npoints-1;i++)
        {
        if (fft[i].re==0 )
                if (fft[i].ima<0)
```

```
                        phasef[i]= 3.0 * pi/2.0; /* no division by 0 */
                else
                        phasef[i]=pi/2;
        if (fft[i].ima==0)
                if (fft[i].re<0)
                        phasef[i]= pi;
                else
                        phasef[i]=0;

                phasef[i] = atan2( fft[i].ima,fft[i].re);

        }

} /* calculate phase of DFT */


square(length, enhpower,enhamp)
int length; float *enhpower; float *enhamp;
{
 int i;
 for(i=0;i<=length-1;i++)  enhamp[i]=sqrt(enhpower[i]);
}

abs_c(length,fft,  abso)

int length;
COMPLEX *fft;
float *abso;
{
int i;
float poww;
 for(i=0;i<=length-1;i++)
        {
        poww=(fft[i].re * fft[i].re ) +( fft[i].ima  * fft[i].ima);
        abso[i]=sqrt(poww);
        }
}


POLAR com_polar(x)
COMPLEX x;

{
POLAR res;
res.r=sqrt(( x.re * x.re) + ( x.ima * x.ima ) );
if (x.re==0)
        {
        if (x.ima ==0) res.angle=0;
        else
                if (x.ima < 0 ) res.angle= 3.0 *pi/2.0;
                else res.angle=pi/2.0;
        }

else    res.angle= atan2(x.ima,x.re);
return(res);
}


COMPLEX polar_com(x)
POLAR x;
{
COMPLEX res;
res.re= x.r * cos(x.angle);
res.ima= x.r * sin(x.angle);
return(res);
```

```
}

POLAR exp_com(x,n)
POLAR x; float n;
{
POLAR res;
res.angle= x.angle * n ;
res.r= pow(x.r,n);

return(res);
}


COMPLEX conjugate(x)

COMPLEX x;

{
COMPLEX y;
y.re=x.re;
y.ima= -1 * x.ima;
return(y);
}



restorephase( length, enhamp,spectrum,  phasef)

int length;
float *enhamp;
COMPLEX *spectrum;
float *phasef;
{
 int i;
 for(i=0;i<=(length-1);i++)
        {
          spectrum[i].re=enhamp[i] * cos(phasef[i]);
          spectrum[i].ima=enhamp[i] * sin(phasef[i]);
        }
}

subtract(cepstrum, encepstrum,estimate,size)
COMPLEX *cepstrum;       /* complex cepstrum to enhance */
COMPLEX *encepstrum;     /* enhanced cepstrum           */
float *estimate;         /* envelope cepstrum of noise*/
int size;
{
int i;
float cphase[VSIZE];
float cenv[VSIZE];
float cenenv[VSIZE];
phase(  size,cepstrum, cphase);
abs_c( size,cepstrum,cenv);
for(i=0;i<size ;i++)
        {
        cenenv[i]=cenv[i]-(estimate[i]) ; /* performed on all bins*/
        }
restorephase( size, cenenv,encepstrum,  cphase);
}



subtract_com(cepstrum, encepstrum,estimate,size)
/* subtraction done on complex cepstrum.*/
COMPLEX *cepstrum;       /* complex cepstrum to enhance */
COMPLEX *encepstrum;     /* enhanced cepstrum           */
```

```
COMPLEX *estimate;      /* complex cepstrum of noise*/
int size;
{
int i;
for(i=0;i<size ;i++)
        {
        encepstrum[i].re=cepstrum[i].re-(estimate[i].re) ; /* performed on all b
        encepstrum[i].ima=cepstrum[i].ima-(estimate[i].ima) ; /* performed on al
        }
}
```

```
/************************************/
/* VA routines                      */
/************************************/

init_VA()
/* initialize Vector Accelerator */

{
        len = VSIZE;
        mapinitva(1, 1, 0);
        /*      Test the exponent power of 2  */
        loglen = mapilog2(len);

        /*    Check the Validity of len   */
        if (len > 2048 || len != 1 << loglen) {
                printf("Length error!\n");
                return(-1);
        }

        /* Allocate vectors in VA memory */
        fftsrc = mapmalloc(len * 2, len * VACINC, VA0);
        ffttmp = mapmalloc(len * 2, 0, VA0);
        coef = mapmalloc(len, 0, VA0);
        /*   Generate the Coefficient Table  */
        mapffttab(coef, loglen);
}

cfft(signal,fft,  len)
     COMPLEX *signal;
     COMPLEX *fft;
     int len;
{
    /*   Copy the input array, signal, from the host   */
     maplodcfv(signal, HCINC, fftsrc, VACINC, len);
     mapsyncmath(-1,VA0);

    /*   Perform a complex FFT on the input data  */
     mapfftnc(fftsrc, VACINC, coef, VACINC, ffttmp, VACINC, len);
     mapsyncdma(-1,VA0);

    /*   Determine the location of the results of step [6] */
     fftdst = (loglen & 1) ? ffttmp : fftsrc;

    /*   Hold up host while result is being Stored  */
     mapstrcfv(fftdst, VACINC, fft, HCINC, len);
     mapbwaitdma(VA0);
```

```
    return(0);
}

icfft(spectrum, outidft,  len)
    COMPLEX *spectrum;
    COMPLEX *outidft;
    int len;
{
        int index;
    /*   Copy the input array, spectrum, from the host   */
     maplodcfv(spectrum, HCINC, fftsrc, VACINC, len);
     mapsyncmath(-1,VA0);


    /*   Take inverse transform & sync on math completion */
    mapifftnc(fftsrc, VACINC, coef, VACINC, ffttmp, VACINC, len);
     mapsyncdma(-1,VA0);


    /*   Determine the location of the results of step [6] */
     fftdst = (loglen & 1) ? ffttmp : fftsrc;

    /*   Hold up host while result is being Stored  */
     mapstrcfv(fftdst, VACINC, outidft, HCINC, len);
     mapbwaitdma(VA0);


     return(0);
}



main (argc,argv)

int argc;
char *argv[];
{

if (argc!=6)
        {
        printf(" error usage: subceps filein fileout noise  samples overest\n");
        exit(0);
        }
open_input(argv[1]);
open_output(argv[2]);
f3 = fopen(argv[3], "r+b");
fread(estimate, sizeof(COMPLEX), VSIZE, f3);
/* noise cepstrum estimate is produced in another program*/

nsamples=atol(argv[4]);
overestimate=atof(argv[5]);
for (ii=0L;ii<VSIZE;ii++)
        {
        estimate[ii].re= overestimate * estimate[ii].re;
        estimate[ii].ima= overestimate * estimate[ii].ima;
        }
init_VA();
ndft=nsamples/MIDSIZE;
make_Bartlett(VSIZE,win) ;/* make window coeffi. once for all*/
readblock(in,VSIZE);   /* initial fill */
for (ii=1L;ii<=ndft;ii++)
        {
        if (ii==1L)
                {
                for (k=0;k<=(MIDSIZE-1);k++) { out_1[k]=0; out_2[k]=0;}
```

```
        }
if ((ii%2L)==0)  /* when even */
/* window 50 % overlapp  */
        {
        readblock(in,VSIZE);
        for (k=0;k<=(MIDSIZE-1);k++)
                {
                slide_in[k]=slide_in[k+MIDSIZE];
                }
        for (k=(MIDSIZE);k<=(VSIZE-1);k++)
                {
                slide_in[k]=in[k-(MIDSIZE)];
                }
        }
else
        for (k=0;k<=(VSIZE-1);k++) { slide_in[k]=in[k];}

window_data(slide_in,win,in_win,VSIZE);
for (k=0;k<=(VSIZE-1);k++)
        {
        sig[k].re=in_win[k];
        sig[k].ima=0;           /* cfft wants a complex array*/
        }
cfft(sig,fft,VSIZE);            /* fft taken on windowed data*/
abs_c(VSIZE,fft,fftpolar);      /* absolute value taken on data*/
phase( VSIZE,fft,phasef);       /* phase is retained for further use*/
for (k=(MIDSIZE+1);k<=(VSIZE-1);k++)/* upper half is zeroized*/
        {
        fftpolar[k]=0;
        }
for (k=1;k<=(MIDSIZE);k++)      /*  root compression*/
        {
        fftpolar[k]=pow(fftpolar[k],ROOTCOMP);
        }
fftpolar[0]=pow(fftpolar[0],BIN0COMP);  /* bin0*/
for (k=0;k<=(VSIZE-1);k++)
        {
        sig[k].re=fftpolar[k];
        sig[k].ima=0;           /* cfft wants a complex array*/
        }

cfft(sig,cepstrum,VSIZE);  /* pseudo cepstrum taken on windowed data*/
subtract_com(cepstrum, encepstrum,estimate,VSIZE);
icfft(encepstrum,outidft,VSIZE);
for (k=0;k<=(VSIZE-1);k++)
        {
        outidft[k].re=outidft[k].re * 1/(VSIZE);
        outidft[k].ima=outidft[k].ima * 1/(VSIZE);
        }

for(k=1;k<=(MIDSIZE);k++)       /*  root decompression*/
        {
        fftpolar2[k]=com_polar(outidft[k]);
        fftpolar2[k]=exp_com(fftpolar2[k],ROOTDECOMP);
        } /* note this  is done on a polar*/
fftpolar2[0]=com_polar(outidft[0]);
fftpolar2[0]=exp_com(fftpolar2[0],BIN0DECOMP);
for(k=MIDSIZE;k<=(VSIZE-1);k++)
        {
        fftpolar2[k]=  fftpolar2[(-1*k)+(VSIZE)] ;
        /* note [1] =[1023]*/
        }       /*  even symmetry is restored here.*/
for(k=0;k<=(VSIZE-1);k++)
        {
        enabs[k]=fftpolar2[k].r;
        }       /* restore phase wants a float*/
```

```
        restorephase(VSIZE,enabs ,spectrum,phasef);     /* result in spectrum*/
        icfft(spectrum,outidft,VSIZE);
        for(k = 0; k < VSIZE; k++)
                {
                outfloat[k] = (1.0/(float)(2*VSIZE)) *  outidft[k].re;
                }
        if ((ii%2)==0)    /* exit window is cumulated at even counts*/
                {
                for (k=0;k<=(MIDSIZE-1);k++)
                        { outs[k]=out_2[k+MIDSIZE]+out_1[k];}
                for (k=MIDSIZE;k<=(VSIZE-1);k++)
                        {outs[k]=out_1[k]+ outfloat[k-MIDSIZE];}
                normalise(outs,out,VSIZE);
                writeblock(out,VSIZE);
                }
        for (k=0;k<=VSIZE-1;k++)        /* for any count*/
                        {
                        out_2[k]=out_1[k];
                        out_1[k]=outfloat[k];
                        }
        }
mapfreeva(VA0);
}
```

```
/*****************************************************
noise_comceps.c
Pseudo-Cepstral subtraction method for speech enhancement.
The method is a version of the INTEL method developed
for the RADC .
Noise estimation in the pseudocepstral complex domain

 Author : Luc Gagnon.
*****************************************************/


#include <stdio.h>
#include <aplib.h>
#include <math.h>
#include "complex2.h"

#define HCINC           8
#define VACINC          2
#define LENGTH          1024
#define VSIZE           1024
#define MIDSIZE         512
#define ROOTCOMP        0.5
#define BIN0COMP        0.75
#define BIG     200



/***********************************/
/* Global variables and constants */
/***********************************/


int loglen, fftdst,fftsrc,ffttmp,coef,len; /* VA variables*/
FILE * f1;              /* input  file*/
FILE * f2;              /* output file*/
FILE * f3;              /* noise file*/
int j,k;                /* counters for main */
COMPLEX fft[VSIZE];     /* dft of incoming block */
COMPLEX sig[VSIZE];     /* complex array for FFT processing.*/
short in[VSIZE]  ;      /* incoming speech*/
float inf[VSIZE]  ;      /* incoming speech multiplied by factor kk*/
float win[VSIZE];       /* window coefficients.*/
short slide_in[VSIZE];  /* overlapped data*/
float in_win[VSIZE];    /* same after window.*/

float phasef[VSIZE];            /* phase of incoming signal */
float fftpolar[VSIZE];          /*   abs value of first fft */
POLAR fftpolar2[VSIZE];         /*   POLAR value of second fft */
COMPLEX spectrum[VSIZE];        /* spectrum of incoming sequence*/
COMPLEX cepstrum[VSIZE];        /* pseudocepstrum*/
COMPLEX encepstrum[VSIZE];      /* pseudocepstrum*/
unsigned long ndft;             /* number of dft */
unsigned long nsamples;
int  ii;                        /* counter */
float out_1[VSIZE];             /* past block  of enhanced speech*/
float out_2[VSIZE];             /* 2nd past block of  enhaned speech*/
float outs[VSIZE] ;             /* slided block written in float*/
float pi=3.141592658;
float enabs[VSIZE];             /* enhanced  cepstrum*/
 COMPLEX estimate[VSIZE];               /* noise cepstrum envelope estimate*/
float overestimate;             /* factor to multiply estimate*/

COMPLEX nois[BIG][VSIZE];               /*  noise power of every windows */
float kk;
/***************************/
/* end of global variable */
```

```
/**************************/


open_input(fichier)
char *fichier;
{
        f1= fopen(fichier,"r+b");
}


open_output(fichier)
char *fichier;
{
        f2= fopen(fichier,"w+b");
}


readblock(buffer, length)
short *buffer; int length;
{
        fread(buffer,sizeof(short),length,f1);
}

writeblock(buffer, length)
short *buffer; int length;
{
        fwrite(buffer,sizeof(short),length,f2);
}



/**************************/
/* window functions       */
/**************************/


make_hanning(length, win)
int     length;
float   *win;
/* makes coeff for Hanning window*/
{
        int     i;
        int     half;
        half = length / 2;
        for (i = 0; i <= length - 1; i++) {
                win[i] = 1 + cos(2 * pi * (i - half) / length);
        }
}


make_hamming( length, win)
int     length;
float   *win;
/* makes coeff for Hamming window*/
{
        int     i;
        int     half;
        half = length / 2;
        for (i = 0; i <= length - 1; i++) {
                win[i] = 0.54 + 0.46 * cos(2 * pi * (i - half) / length);
        }
}


make_Bartlett( length, win)
```

```
/* makes coeff for Bartlett window*/

int length;
float *win;
{
int i;
int half;
half=length/2;
for(i=half;i<=length-1;i++)
        {
        win[i]=2.0 -(4.0 * (i-half)) /  length;
        }
win[0]=0;
for(i=1;i<half;i++)
        {
        win[i]=win[length-i];;
        }
}



make_rect( length, win)
/* trivial */
int     length;
float   *win;
/* makes coeff for  rectangular   window*/
{
        int     i;
        int     half;
        half = length / 2;
        for (i = 0; i <= length - 1; i++) {
                win[i] = 1.0;
        }
}



window_data(in, win, in_win, length)
short   *in;
float   *win;
float   *in_win;
int     length;
/* multiply window with data*/
{
        int     i;
        for (i = 0; i <= length - 1; i++) {
                in_win[i] =  (float)in[i] * win[i];
        }
}




/*********************************/
/* Complex arithmetic routines */
/*********************************/

phase(  npoints,fft, phasef)

int npoints;
COMPLEX *fft;
float *phasef;
{
  float pi=3.141592658;
  int i;
```

```
 for(i=0;i<=npoints-1;i++)
        {
        if (fft[i].re==0 )
                if (fft[i].ima<0)
                        phasef[i]= 3.0 * pi/2.0; /* no division by 0 */
                else
                        phasef[i]=pi/2;
        if (fft[i].ima==0)
                if (fft[i].re<0)
                        phasef[i]= pi;
                else
                        phasef[i]=0;

                phasef[i] = atan2( fft[i].ima,fft[i].re);

        }
} /* calculate phase of DFT */


square(length, enhpower,enhamp)
int length; float *enhpower; float *enhamp;
{
 int i;
 for(i=0;i<=length-1;i++)  enhamp[i]=sqrt(enhpower[i]);
}

abs_c(length,fft,  abso)

int length;
COMPLEX *fft;
float *abso;
{
int i;
float poww;
 for(i=0;i<=length-1;i++)
        {
        poww=(fft[i].re * fft[i].re ) +( fft[i].ima  * fft[i].ima);
        abso[i]=sqrt(poww);
        }
}



POLAR com_polar(x)
COMPLEX x;

{
POLAR res;
res.r=sqrt(( x.re * x.re) + ( x.ima * x.ima ) );
if (x.re==0)
        {
        if (x.ima ==0) res.angle=0;
        else
                if (x.ima < 0 ) res.angle= 3.0 *pi/2.0;
                else res.angle=pi/2.0;
        }

else    res.angle= atan2(x.ima,x.re);
return(res);
}


COMPLEX polar_com(x)
POLAR x;
{
```

```
COMPLEX res;
res.re= x.r * cos(x.angle);
res.ima= x.r * sin(x.angle);
return(res);
}

POLAR exp_com(x,n)
POLAR x; float n;
{
POLAR res;
res.angle= x.angle * n ;
res.r= pow(x.r,n);

return(res);
}


COMPLEX conjugate(x)

COMPLEX x;

{
COMPLEX y;
y.re=x.re;
y.ima= -1 * x.ima;
return(y);
}



restorephase( length, enhamp,spectrum,  phasef)

int length;
float *enhamp;
COMPLEX *spectrum;
float *phasef;
{
 int i;
 for(i=0;i<=(length-1);i++)
        {
          spectrum[i].re=enhamp[i] * cos(phasef[i]);
          spectrum[i].ima=enhamp[i] * sin(phasef[i]);
        }
}

subtract(cepstrum, encepstrum,estimate,size)


COMPLEX *cepstrum;      /* complex cepstrum to enhance */
COMPLEX *encepstrum;    /* enhanced cepstrum           */
float *estimate;        /* envelope cepstrum of noise*/
int size;
{
int i;
float cphase[VSIZE];
float cenv[VSIZE];
float cenenv[VSIZE];
phase(  size,cepstrum, cphase);
abs_c( size,cepstrum,cenv);
for(i=0;i<size ;i++)
        {
        cenenv[i]=cenv[i]-(estimate[i]) ; /* performed on all bins*/
        }

restorephase( size, cenenv,encepstrum,  cphase);
```

```
}



/**********************************/
/* VA routines                    */
/**********************************/

init_VA()
/* initialize Vector Accelerator */

{
        len = VSIZE;
        mapinitva(1, 1, 0);
        /*      Test the exponent power of 2  */
        loglen = mapilog2(len);

        /*    Check the Validity of len   */
        if (len > 2048 || len != 1 << loglen) {
                printf("Length error!\n");
                return(-1);
        }

        /* Allocate vectors in VA memory */
        fftsrc = mapmalloc(len * 2, len * VACINC, VA0);
        ffttmp = mapmalloc(len * 2, 0, VA0);
        coef = mapmalloc(len, 0, VA0);
        /*   Generate the Coefficient Table  */
        mapffttab(coef, loglen);
}

cfft(signal,fft,  len)
     COMPLEX *signal;
     COMPLEX *fft;
     int len;
{
    /*   Copy the input array, signal, from the host   */
     maplodcfv(signal, HCINC, fftsrc, VACINC, len);
     mapsyncmath(-1,VA0);

    /*   Perform a complex FFT on the input data  */
     mapfftnc(fftsrc, VACINC, coef, VACINC, ffttmp, VACINC, len);
     mapsyncdma(-1,VA0);

    /*   Determine the location of the results of step [6] */
     fftdst = (loglen & 1) ? ffttmp : fftsrc;

    /*   Hold up host while result is being Stored  */
     mapstrcfv(fftdst, VACINC, fft, HCINC, len);
     mapbwaitdma(VA0);


     return(0);
}

icfft(spectrum, outidft,  len)
     COMPLEX *spectrum;
     COMPLEX *outidft;
     int len;
{
        int index;
```

```
    /*   Copy the input array, spectrum, from the host   */
     maplodcfv(spectrum, HCINC, fftsrc, VACINC, len);
     mapsyncmath(-1,VA0);


    /*   Take inverse transform & sync on math completion */
    mapifftnc(fftsrc, VACINC, coef, VACINC, ffttmp, VACINC, len);
     mapsyncdma(-1,VA0);


    /*   Determine the location of the results of step [6] */
     fftdst = (loglen & 1) ? ffttmp : fftsrc;

    /*   Hold up host while result is being Stored  */
     mapstrcfv(fftdst, VACINC, outidft, HCINC, len);
     mapbwaitdma(VA0);


     return(0);
}



main (argc,argv)

int argc;
char *argv[];
{

if (argc!=5)
        {
        printf(" error usage: noise_ceps filein fileout samples k\n");
        exit(0);
        }
open_input(argv[1]);
f3 = fopen(argv[2], "w+b");

nsamples=atol(argv[3]);
kk=atof(argv[4]);
init_VA();
ndft=nsamples/MIDSIZE;
make_Bartlett(VSIZE,win) ;/* make window coeffi. once for all*/
readblock(in,VSIZE);   /* initial fill */
for (k=0;k<=(VSIZE-1);k++)
        inf[k]= kk* (float) in[k];
for (ii=1L;ii<=ndft;ii++)
        {
        if ((ii%2L)==0)  /* when even */
        /* window 50 % overlapp  */
                {
                readblock(in,VSIZE);
                for (k=0;k<=(VSIZE-1);k++)
                                inf[k]= kk* (float)in[k];
                for (k=0;k<=(MIDSIZE-1);k++)
                        {
                        slide_in[k]=slide_in[k+MIDSIZE];
                        }
                for (k=(MIDSIZE);k<=(VSIZE-1);k++)
                        {
                        slide_in[k]= inf[k-(MIDSIZE)];
                        }
                }
        else
                for (k=0;k<=(VSIZE-1);k++) { slide_in[k]=  inf[k];}
```

```
        window_data(slide_in,win,in_win,VSIZE);
        for (k=0;k<=(VSIZE-1);k++)
                {
                sig[k].re=in_win[k];
                sig[k].ima=0;                   /* cfft wants a complex array*/
                }
        cfft(sig,fft,VSIZE);                    /* fft taken on windowed data*/
        abs_c(VSIZE,fft,fftpolar);              /* absolute value taken on data*/
        for (k=(MIDSIZE+1);k<=(VSIZE-1);k++)/* upper half is zeroized*/
                {
                fftpolar[k]=0;
                }
        for (k=1;k<=(MIDSIZE);k++)              /*  root compression*/
                {
                fftpolar[k]=pow(fftpolar[k],ROOTCOMP);
                }
        fftpolar[0]=pow(fftpolar[0],BIN0COMP);  /* bin0*/
        for (k=0;k<=(VSIZE-1);k++)
                {
                sig[k].re=fftpolar[k];
                sig[k].ima=0;                   /* cfft wants a complex array*/
                }

        cfft(sig,cepstrum,VSIZE);  /* pseudo cepstrum taken on windowed data*/

        for (k=0;k<=(VSIZE-1);k++)
                {
                nois[ii][k].re=cepstrum[k].re;  /*  */
                nois[ii][k].ima=cepstrum[k].ima;  /*  */
                }
        }

for (k=0;k<=(VSIZE-1);k++)
        {
        estimate[k].re=0;
        estimate[k].ima=0;
        for (ii=1;ii<=ndft;ii++)
                {
                estimate[k].re += nois[ii][k].re;
                estimate[k].ima += nois[ii][k].ima;
                }
        estimate[k].re= estimate[k].re/ndft;
        estimate[k].ima= estimate[k].ima/ndft;
        /*printf(" %u  %f \n\n",k,estimate[k]);*/

        }/*avg of var*/
fwrite(estimate,sizeof(COMPLEX),VSIZE,f3);
mapfreeva(VA0);
}
```