Proc. 1994 Digital Communications Conference

FAST CELP ALGORITHM AND IMPLEMENTATION FOR SPEECH COMPRESSION

A. Langi, VE4ARM, W. Grieder, VE4WSG, and W. Kinsner, VE4WK

Department of Electrical and Computer Engineering
and Telecommunications Research Laboratories
University of Manitoba
Winnipeg, Manitoba, Canada R3T 5V6
Tel.: (204) 474-6992; Fax: (204) 275-0261
eMail: kinsner@ee.umanitoba.ca

ABSTRACT

This paper describes a fast algorithm and implementation of *code excited linear predictive* (CELP) speech coding. It presents principles of the algorithm, including (i) fast conversion of *line spectrum pair* parameters to linear predictive coding parameters, and (ii) fast searches of the parameters of *adaptive* and *stochastic codebooks*. The algorithm can be readily used for speech compression applications, such as on (i) high quality low-bit rate speech transmission in point-to-point or store-and-forward (network based) mode, and (ii) efficient speech storage in speech recording or multimedia databases. The implementation performs in real-time and near real-time on various platforms, including an IBM-PC AT equipped with a TMS320C30 module, an IBM PC 486, a SUN Sparcstation 2, a SUN Sparcstation 5, and an IBM Power PC (Power 590).

1. INTRODUCTION

1.1. Why is CELP Useful ?

Obtaining efficient representation of speech at low bit rates for communication or storage has been a problem of considerable importance, because of technical as well as economical requirements. Telephone-quality digital speech in a *pulse code modulation* (PCM) form requires a 64 kbits/s rate which cannot be transmitted in real time through 6 kHz and 30 kHz channel capacities of HF and VHF bands, respectively. Voice mail and multimedia employ speech storage, demanding efficient ways of storing speech, since one minute of PCM speech already requires 480 kbytes of storage space. Even if the channel can accommodate real-time speech, speech compression allows more communication connections to share the precious channel. Similarly, speech compression allows more speech messages to be stored in the storage of the same size.

This paper describes a speech compression technique for those purposes, called *code-excited linear predictive* (CELP) coding [Atal86][JaJS93], which obtains bit rates of as low as 4.8 kbits/s, giving a compression ratio of up to 13:1 [CaTW90]. Although this rate is higher than a 2.4 kbits/s *linear predictive coding* (LPC), speech compressed by CELP has quality, naturalness, and speaker recognizability, which are missing from the LPC.

The importance of CELP goes beyond its quality vs. bit-rate performance, as it provides a generic structure for future generation of *perceptual* speech coders [JaJS93]. All speech compression techniques have been based on two intrinsic operations: removal of *redundancy* and removal of *irrelevancy.* The first operation uses *prediction* and/or *transforms* to remove redundant data, thus reducing the bit rates. The second operation further reduces the bit rates through quantization of (i) the time components of the prediction error or (ii) the transform coefficients, allowing mathematically non-zero but *imperceptible* reconstruction error or distortion.

If further compression is still required, the coder minimizes the error perceptibility by exploiting *masking* properties of human speech perception. To certain extent, the speech energy itself perceptually masks the distortion. Thus the same energy levels of distortion have different perceptual effect if applied to speech signals with different energy levels. This approach promises a new level of higher quality and lower bit rate speech compression [JaJS93]. Coders that minimize perceptual distortion (such as CELP) are called perceptual coders.

One novelty of CELP is in incorporating the masking property in a working, practical scheme. Such incorporation is non trivial because perceptual distortion measures lack tractable means that have often been available in the traditional distortion energy measure.

The CELP solution to this problem is by using an *analysis-by-synthesis* approach, where the perceptual distortion is literally measured. CELP then exploits the computational structure, resulting in a sophisticated, practical compression technique. Clearly, the computational cost is very high.

1.2. Conceptual CELP

As shown in Fig. 1, a conceptual CELP structure [ScAt85] consists of:

a. two predictors (pitch and spectral predictor filters) to remove redundancy caused by long and short term correlations among speech samples, respectively; and

b. a close-loop, perceptual vector quantizer utilizing a codebook to remove irrelevancy indirectly from the time components of the prediction error.

The codebook stores random (stochastic) signals as prototypes of excitation signals for the two predictor filters. Furthermore, a perceptual weighting filter ensures that mean-square error measurement reflects the perceptual error measurement.

The CELP compressed speech then consists of:

a. a set of spectral predictor parameters;

b. a set of pitch predictor parameters; and

c. codebook (entry and gain) parameters.

It is these CELP parameters than can be transmitted or stored at rates as low as 4.8 kbits/s.

The speech compression algorithm begins by obtaining the predictor parameters, and then searching for codebook parameters corresponding to excitation prototype that minimizes the perceptual error. The CELP decompressor uses the codebook parameters to produce the excitation signal, exciting the cascade of pitch and spectral filters, resulting with the decompressed speech.

The selection of the predictors and the quantizer is by no means arbitrary. They match elements of a model of human speech production system [Lang92]. The model consists of an excitation source and a vocal tract. During *voiced* speech articulations, the excitation source produces quasi periodic pulses which excite the vocal tract. The pulses are subjected to resonance and anti-resonance processes in the vocal tract according to the changes in the vocal tract shape over time, resulting in audible and meaningful speech. Similar processes take place during *stop* and *fricative* articulations. However, the excitation source should produce noise-like excitations instead. In matching the model, CELP uses the spectral predictor filter to perform vocal tract function. The pitch predictor filter (usually a one-tap, all-pole filter) ensures the quasi-periodicity of the spectral filter excitation. In this cascaded filter structure, it is known that voiced speech signals have excitations of Gaussian distribution. Thus the codebook members represent such excitations. It also accommodates excitation for stops and fricatives. The fact that the CELP structure serves both signal compression principles (i.e., redundancy and irrelevancy removals) and speech production model (i.e., an articulation source and vocal tract) is the reason for the CELP highly successful performance.

1.3. Implementation Problem

Despite its concept maturity, real-time CELP implementation is still a complex problem. The codebook searching is so computationally demanding that a direct implementation requires very long computation time, much more than real-time requirement. In the searching process, each prototype

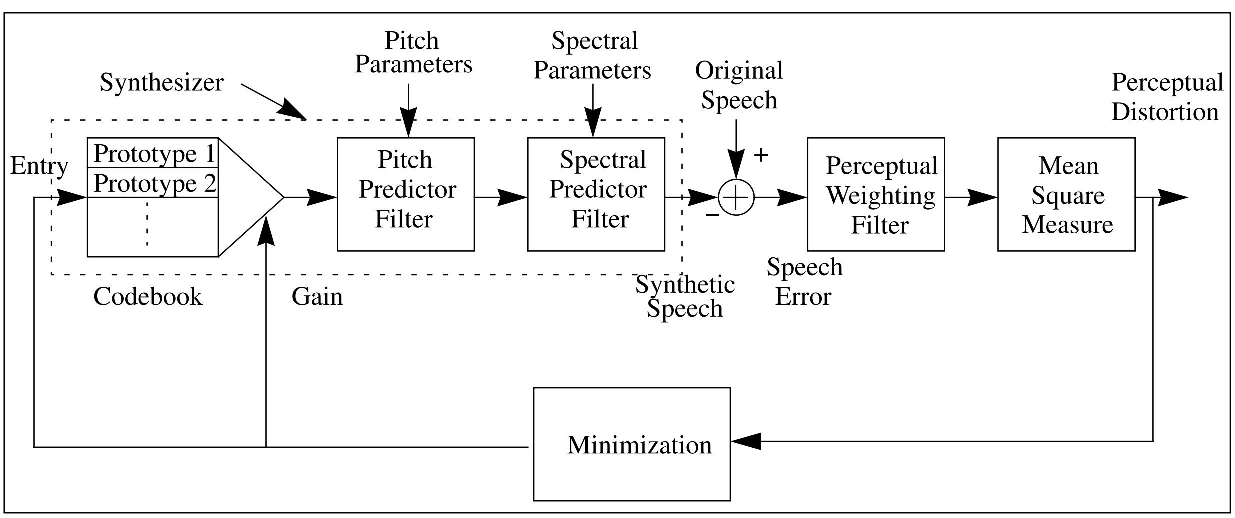

Fig. 1. Conceptual CELP analyzer.

must go through three filtering (the pitch, spectral, and perceptual filters) and one mean-square processes. It is easy to show that a brute-force approach would require a processor with more than 34 million MIPS, for a real-time CELP [Lang92]. An early 'practical' CELP implementation required 125s of Cray-1 computation time to process one second speech [ScAt85], while real-time procedure must process one second of speech in one second or less.

Thus, a practical CELP system must employ fast algorithms, which exploit the computational structure of a CELP scheme. In the process of developing practical CELP, the actual structure becomes significantly different from the conceptual one, while still performing the same functions (see [Lang92] for details on the transition). For example, the spectral parameters are quantized and represented now by a set of *line-spectrum pairs* (LSP) [SoJu84]. The pitch filter becomes another codebook, called *adaptive code book* (ACB). The codebook of the random signals is then called *stochastic code book* (SCB).

Unfortunately, the fast algorithm has significantly increased the implementation complexity as the optimization blurs the structure in favor of speed. The algorithm now combines the spectral predictor and the perceptual weighting filter into one filter. A joint optimization scheme searches for the suboptimal combination of codebook parameters, instead optimal combination through total exhaustive search of all combinations, as implied by the conceptual structure. The use of a special SCB results in a fast iterative search, in which the results of the perceptual distortion calculation from current prototype helps the calculation of that of the next prototype. It should be noted that although there is a proposed U.S. Federal Standard (FS) 1016 CELP [CaTW90] which describes each bit in the compressed speech, it does not specify how to obtain the compressed speech, leaving it to CELP implementors to develop one.

1.4. Paper Overview

The remaining part of this paper describes a practical, near real-time CELP algorithm, which reduces the computational power requirement by a factor of more than 175,000. Section 2 describes the procedures to compress and decompress speech. This paper focuses mainly on the description of algorithms compatible with the FS-1016 to enable communication with other FS-1016 systems. In Section 3, we briefly explain the actual computer implementation, resulting in performance ranging from 14 to 0.85 of real time, depending on the platform. The algorithm has been implemented on an IBM PC-AT equipped with a TMS320C30 (C30) evaluation module (EVM) [LaKi91],[Lang92]. The system is suitable for PC-based packet radio or speech recording systems. The algorithm has also been ported to the various UNIX platforms as well as MS Windows 3.1 platform for a voice mail development. Section 4 discusses performance of the various implementations, including their limitations. Finally, Section 5 provides conclusions.

2. Fast CELP Procedures

2.1. Input and Output

In practice, CELP is a block coding, in which a *frame* of 240 PCM speech samples $s[n]$ (with a total of 1.92 kbits) denoted as a vector $\mathbf{s}$ is converted to 144 bits of compressed data, called FS-1016 CELP parameters or *data stream*. The CELP parameters now consist of:

a. the *line spectrum pair* (LSP) parameters;

b. the *adaptive codebook* (ACB) parameters; and

c. the *stochastic codebook* (SCB) parameters.

All LSP, ACB, and SCB parameters are entries (indexes) of quantization tables and codebooks, namely LSP table, ACB, ACB gain table, SCB, and SCB gain table [LaKi90], [LaKi91]. They all require 138 bits only. The remaining 6 bits can be used for error correction, synchronization, and future expansion.

Naturally, the CELP procedures should perform a CELP compressor and decompressor system extracting CELP parameters from $s[n]$, and reconstructing $\mathbf{s}$ back from the FS-1016 data stream. Specifically, a CELP compressor (usually called *analyzer*) requires (i) LSP analysis procedure to obtain the LSP parameters, and (ii) codebook search procedure for both ACB and SCB parameters, while a CELP decompressor (usually called *synthesizer*) requires speech synthesis procedure. We describe the procedures as follow.

2.2. LSP Analysis

The CELP analyzer obtains the LSP parameters through the following three steps: (i) performing *linear predictive coding* (LPC) analysis on the PCM samples to represent spectral information [Pars86], [Proa83], (ii) converting the LPC parameters into LSP parameters [KaRa86], [HaHe90] for efficient representation, and (iii) ensuring LSP parameter stability.

2.2.1. LPC Analysis

The aim of LPC analysis is to obtain LPC parameters a_i (collectively denoted as $\mathbf{a}$) corresponding to the spectral filter. The spectral (or LPC) filter models a human vocal tract. One most common model is a 10-order all-pole digital filter $H(z)$ with ten coefficients a_i, as follows

$$H(z) = \frac{1}{1 + \sum_{i=1}^{10} a_i z^{-i}} = \frac{1}{A(z)} \qquad (1)$$

Let the input (excitation) of this filter be a zero-mean signal **t**. The output of this filter is then $\hat{\mathbf{s}}$, according to (in z-domain notations)

$$\hat{S}(z) = H(z)\,T(z)\,. \qquad (2)$$

For a given **s**, the LPC analysis finds **a** that minimizes $\|\mathbf{s} - \hat{\mathbf{s}}\|$. The elements ($a_i$) of such a vector **a** are LPC parameters, which are the solutions of a linear equation system

$$0 = \sum_{i=0}^{10} a_i r_{i-j} \qquad a_0 = 1 \qquad j = 1, \ldots, 10 \quad (3)$$

where r_i are autocorrelation terms defined as

$$r_i = \sum_{n=i}^{N-1} s[n]\,s[n-i] \qquad i = 0, \ldots, 10 \qquad (4)$$

2.2.2. LPC to LSP Parameter Conversion

The system must quantize **a** using the LSP analysis since a_i are 10 real numbers which require too many bits (10×16=160 bits) for representation. On the other hand, the LSP parameters (we call them LSP_j) are more efficient (only 34 bits) because they are ten integers ranging form 0 to 8 (or to 16), corresponding to the entries of a suitable LSP table.

To show the conversion, we first show that **a** can be represented by z_i, which are zeros of two polynomials $p(z)$ and $q(z)$ related through

$$\begin{aligned} A(z) &= \frac{1}{2}(p(z) + q(z)) \\ p(z) &= A(z) + z^{-11} A(z^{-1}) \\ q(z) &= A(z) - z^{-11} A(z^{-1}) \end{aligned} \qquad (5)$$

Clearly, polynomials $p(z)$ and $q(z)$ represent $H(z)$. In other words, zeros z_i of $p(z)$ and $q(z)$ (eleven each) can represent **a**.

Furthermore, z_i can be represented fully by ω_i, as

$$\omega_i = \arg(z_i)\,; \qquad i = 0, \ldots, 9 \qquad (6)$$

where arg(•) is the argument of a complex variable. The proof relies on the fact that $z = 1$ and $z = -1$ are always the zeros of $p(z)$ and $q(z)$, respectively. Thus the 20 remaining zeros are sufficient to represent $p(z)$ and $q(z)$. Furthermore, all z_i are symmetric about the real axis, and lie on the unit circle in the z-plane. Thus, 10 zeros (below the real line) are actually redundant, leaving us with the remaining 10 significant zeros, which uniquely corresponding to 10 values of ω_i through Eq. (6). Furthermore, it can be shown that ω_i with even and odd i correspond to $p(z)$ and $q(z)$, respectively. We then conclude that these 10 values of ω_j can reconstruct all the zeros of $p(z)$ and $q(z)$, thus representing **a**. Equivalently, for a given **a**, we can always derive such ω_j.

Having obtained ω_j, we can efficiently represent them through quantization. Although we can directly quantize a_j, the dynamic range of a_j is high (i.e. there are many significant values of a_j), requiring many quantization steps to achieve low quantization error. On the other hand, each ω_j has a much limited dynamic range, since the ranges of ω_j are disjoint subintervals S_j, in a real-number interval of 0 to π, i.e.,

$$\begin{aligned} &\omega_j \in S_j; \qquad 0 \le \bigcup_j S_j < \pi; \\ &i \ne j \Rightarrow (S_j \cap S_i = \varnothing)\,; \qquad j = 0, \ldots, 9 \end{aligned} \qquad (7)$$

Thus, fewer quantization steps for ω_j can achieve the same quantization error.

We then use the FS-1016 LSP table to quantize ω_j. For each ω_j, FS-1016 sets a *list* of 8 possible quantized values of ω_j (or 16 if j is 2 to 5), covering S_j and its neighborhood. Thus, there are 10 lists, namely list j, j = 0 to 9, collectively called the FS-1016 *LSP table*. Let *LSPTable*[j,i] be a particular quantized value indexed by i in list j, where i is from 0 to 7, or to 15. We quantize ω_j by selecting i such that *LSPTable*[j,i] is the closest value to ω_j in list j. Now, assigning such an i to LSP_j and performing similar steps for all j, we have LSP_j as a representation of the quantized ω_j. We have called those LSP_j as LSP parameters, which can now represent **a**. This representation is efficient because we only need 3+4+4+4+4+3+3+3+3+3 = 34 bits for each **a**, instead of 160 bits in the original floating-point form.

One advantage of using the FS-1016 LSP table is that we can derive a fast LSP conversion algorithm, by searching the table without actually knowing the exact zeros. There are numerical methods such as Newton-Rhapson and Jenkins-Traub [PTVF92] for finding the zeros of $p(z)$ and $q(z)$, but they are tedious. Furthermore, the exact ω_j must later be quantized anyway.

A different and faster approach is by checking zero-crossing of a new pair of polynomials $\tilde{p}(x)$ and $\tilde{q}(x)$. These polynomials are related to $p(z)$ and $q(z)$ in the fact that their zeros, x_i, are

$$x_i = \cos \omega_i \tag{8}$$

Such $\tilde{p}(x)$ and $\tilde{q}(x)$ must then take a form of

$$\begin{aligned} \tilde{p}(x) &= \sum_{i=0}^{5} b_i x^i \\ \tilde{q}(x) &= \sum_{i=0}^{5} c_i x^i \end{aligned} \tag{9}$$

where the coefficients **b** and **c** are

$$\begin{aligned} b_5 &= 32 \\ b_4 &= 16p_1 \\ b_3 &= 8(p_2 - 5) \\ b_2 &= 4(p_3 - 4p_1) \\ b_1 &= 2(p_4 - 3p_2 + 5) \\ b_0 &= p_5 - 2p_3 + 2p_1 \end{aligned} \tag{10}$$

and

$$\begin{aligned} c_5 &= 32 \\ c_4 &= 16q_1 \\ c_3 &= 8(q_2 - 5) \\ c_2 &= 4(q_3 - 4q_1) \\ c_1 &= 2(q_4 - 3q_2 + 5) \\ c_0 &= q_5 - 2q_3 + 2q_1 \end{aligned} \tag{11}$$

Here, p_i and q_i are coefficients of $p(z)$ and $q(z)$, respectively, where i refers to a polynomial term containing z^i. The p_0 and q_0 are always equal to one. For a given **a**, it is easy to show using Eq. (7a) that the remaining p_i and q_i can be obtained recursively through a loop of i from 1 to 5 of

$$\begin{aligned} p_i &= a_i + a_{11-i} - p_{i-1} \\ q_i &= a_i + a_{11-i} - q_{i-1} \end{aligned} \tag{12}$$

The fast LSP conversion then uses the fact that each x associated with a zero of $p(z)$ or $q(z)$ causes $p(x)$ or $q(x)$ to be zero, respectively. Thus, the scheme applies values of x corresponding to ω in the LSP table (i.e., *LSPTable*[j,i]) to the polynomials $\tilde{p}(x)$ and $\tilde{q}(x)$, and observes for zero crossings. As before, j even and odd correspond to $\tilde{p}(x)$ and $\tilde{q}(x)$, respectively. For each j, the scheme then assigns certain i to LSP_j, such that x=*LSPTable*[j,i] is the closest x within the same j that causes a zero crossing of $\tilde{p}(x)$ or $\tilde{q}(x)$.

2.2.3. Ensuring LSP Stability.

We must have a scheme for robust representation of the LPC parameters, because they are very sensitive and the conversion to LSP parameters increases the sensitivity. Since $H(z)$ is a recursive filter, a distortion in **a** can easily move the poles of $H(z)$ to outside the unit circle of the z-plane, resulting in an unstable $H(z)$. The conversion to LSP further introduces more distortion due to quantization errors.

Fortunately, if the ordered values of ω_j are monotonically increasing (from 0 to π), the LSP method guarantees the stability of $H(z)$ [SoJu84]. Thus, before transmitting the LSP_j, the scheme verifies the ordered values of ω_j corresponding to LSP_j. If the ordered values violate the monotonicity, the scheme replaces it with a stable set of LSP_j form previous frame.

Sometimes, the pre-defined quantization steps can also create a stability problem. There are cases when some adjacent ω_j are too close together, so that for the given resolution, the table fails to distinguish them. Or, the ω_j may lie beyond the table coverage. In this situation, the fast LSP conversion usually gives incorrect, unstable LSP_j. An effort to avoid such cases is by expanding the bandwidth of **a** prior to LSP conversion process. Thus, instead of using **a**, the scheme use **c**, defined as

$$c_i = a_i \gamma^i \tag{13}$$

where γ is the expanding factor (typically set to 0.994), and i is an index from 1 to 10.

2.3. Codebook Parameter Searching

2.3.1. Searching Problem

To obtain the codebook parameters, the analysis searches for codebook parameters minimizing perceptual distortion

$$\|\mathbf{e}\|^2 = \|\mathbf{s} - \hat{\mathbf{s}}\|_W^2 = \left\| P_W[\mathbf{s} - \hat{\mathbf{s}}] \right\|^2 \tag{14}$$

where $\|\bullet\|$ denotes a norm (or magnitude) of a vector, and P_w represents a perceptual weighting filter defined as

$$P_W(z) = \frac{H\left(\frac{z}{\gamma}\right)}{H(z)} \qquad 0 \le \gamma \le 1 \tag{15}$$

A typical γ is 0.8. (Such a $P_W(z)$ makes Eq. (2) a perceptual spectral-masking based measure rather than simply a pure Euclidean measure of waveform closeness). We call **e** the perceptual error vector.

The codebook parameters affect perceptual distortion in Eq. (14) through the excitation **t** and then $\hat{\mathbf{s}}$. A codebook consists of prototypes or *codewords* **b**, which are arrays of impulses $b[n]$. Each codeword is indexed by a codebook *entry* called *CBEntry*. For each codebook, there is a gain table containing *gain factors*, which are real numbers. Each gain factor is indexed by a gain table entry called *GainEntry*. Thus for the ACB and SCB there are *ACBEntry* and *SCBEntry*, respectively, while for the ACB and SCB gain table entry there are *ACBGainEntry* and *SCBGainEntry*, respectively.

A set of those entries produces **t** according to

$$\begin{aligned} \mathbf{t} &= \mathbf{b}_{(a)}(ACBEntry)g_{(a)}(ACBGainEntry) + \\ &\quad \mathbf{b}_{(s)}(SCBEntry)g_{(s)}(SCBGainEntry) \end{aligned} \tag{16}$$

The $\mathbf{b}_{(a)}(ACBEntry)$ and $\mathbf{b}_{(s)}(SCBEntry)$ are the ACB and SCB codewords pointed by *ACBEntry* and *SCBEntry*, respectively, while $g_{(a)}(ACBGainEntry)$ and $g_{(s)}(SCBGainEntry)$ are the ACB and SCB gain factors pointed by *ACBGainEntry* and *SCBGainEntry*, respectively. For a given **s**, the **t** produces $\hat{\mathbf{s}}$ and then **e** according to Eq. (2) and Eq. (14), respectively. Thus the search problem becomes: for a given *searching target* **s**, find *ACBEntry*, *SCBEntry*, *ACBGainEntry*, and *SCBGainEntry* corresponding to **e** that minimizes Eq. (16).

To solve the searching problem, there are several techniques such as those described in [KlKK90]. However, not all off them can be combined. We describe here fast searching algorithms that we actually use. Some are mandatory (implied by FS-1016), while some are our choice. We also discuss their consequences in the scheme.

2.3.2. Breaking the Frames into Subframes

One obvious way to reduce the computational cost for searching is by reducing the size of the codebooks, i.e., reducing the number of prototypes in the codebook. However, this approach increases the vector quantization error. To reduce the quantization error, one should reduce the length (i.e., dimension) of the prototype. However, this increases the bit requirement because we need more prototype to represent a segment of **t**. FS-1016 solves this delicate balance by using a prototype length of 60 samples. This means, the searching target in one frame is split into four **s** in four *subframes*, and the scheme performs four searching processes to complete encoding of one frame, resulting in four sets of codebook entries. The SCB size can then be reduced to as low as 512 while preserving natural speech quality.

It should be noted that since ACB is a codebook that actually represents a one adaptive tap, all pole pitch filter [Lang92], its size is not determined this way. The ACB size determines the range of pitch frequency it can cover. For an excitation $x[n]$, the filter produces

$$y[n] = gy[n-d] + x[n] \tag{17}$$

with g as the filter coefficient (equivalent with ACB gain) and d is the tap position (equivalent with ACB entry). Varying d changes the pitch frequency (in Hz) according to

$$\text{Pitch Frequency} = \frac{\text{Sampling Frequency}}{d} \tag{18}$$

FS-1016 covers pitch frequency between 54 Hz to 400 Hz, requiring d to be between 20 to 147. Thus, we use an ACB size of 128. FS-1016 actually provides a size option of 256 to improve the pitch resolution in high frequency (associated with woman speakers). It is clear from Eq. (18) that the pitch resolution at higher frequency is coarser. The additional ACB entries are then added to improve the high frequency resolution. To reduce the computational cost, we did not use this option.

The subframe search approach also enable a smoother transition of LSP parameters through interpolation. Thus for each subframe $i = 1, \ldots, 4$, the scheme uses different $H(z)$ coming from interpolated LSP parameters defined as

$$\omega_j = \frac{9-2i}{8}\text{Previous}\,\omega_j + \frac{2i-1}{8}\text{Present}\,\omega_j \tag{19}$$

Thus the system must always keep the LSP parameters from the previous frame.

2.3.3. Combining Perceptual and Spectral Filters

We can reduce the computation cost by reducing the number of filters used during the search. To compute the perceptual distortion in Eq. (14), each prototype must pass through the LPC filter and the perceptual weighting filter. In the z-domain, the perceptual distortion vector is

$$\begin{aligned} E(z) &= P_w(z)\{S(z) - \hat{S}(z)\} \\ &= P_w(z)S(z) - P_w(z)H(z)T(z) \\ &= Y(z) - W(z)T(z) \\ &= Y(z) - X(z) \end{aligned} \tag{20}$$

where

$$Y(z) = P_w(z)S(z) \quad (21)$$

$$W(z) = P_w(z)H(z) = \frac{H\left(\frac{z}{\gamma}\right)}{H(z)}H(z) = H\left(\frac{z}{\gamma}\right) \quad (22)$$

$$X(z) = W(z)\,T(z) \quad (23)$$

Observe that there is only one filtering $W(z)$ required now (i.e., Eq. (23)) for every prototype. As a new searching target, $Y(z)$ is calculated once only using Eq. (21), and then the search minimizes (in vectorial notations)

$$\|\mathbf{e}\|^2 = \|\mathbf{y} - \mathbf{x}\|^2 \quad (24)$$

There is a slight problem of this approach if we calculate Eq. (23) in vector and matrix operations. In a matrix form, filter $W(z)$ is approximated by a 60×60 matrix W defined as

$$W = \begin{bmatrix} w[0] & 0 & \dots & 0 \\ w[1] & w[0] & \dots & 0 \\ \dots & \dots & \dots & \dots \\ w[59] & w[58] & \dots & w[0] \end{bmatrix} \quad (25)$$

where $w[i]$ are the impulse responses of $W(z)$, such that

$$\mathbf{x} = W\mathbf{t} \quad (26)$$

Unfortunately, the search results are good only if the CELP synthesizer also uses $H(z)$ in a matrix form, which is not the case. Let $\mathbf{z}$ be the zero response of $H(z)$ at the synthesizer, i.e., $z[n]$ are the output of the $H(z)$ when its input is zero for all subframe. In practice, $\mathbf{z}$ is not zero due to the non-zero contents of the $H(z)$ delay elements, resulting from the previous excitation. Thus, the actual output of the synthesizer is

$$\hat{\mathbf{s}} = H\mathbf{t} + \mathbf{z} \quad (27)$$

The analyzer must then introduce a compensation scheme such that we minimize Eq. (14) but still use combined filter W with Eq. (26). From Eq. (27) we have

$$P_w\hat{\mathbf{s}} = P_w H\mathbf{t} + P_w\mathbf{z} = \mathbf{x} + P_w\mathbf{z} \quad (28)$$

Using the derivation in Eq. (20), we have

$$\begin{aligned} \|\mathbf{e}\|^2 &= \left\|P_w\mathbf{s} - \mathbf{x} - P_w\mathbf{z}\right\|^2 \\ &= \left\|P_w(\mathbf{s} - \mathbf{z}) - \mathbf{x}\right\|^2 \\ &= \|\tilde{\mathbf{y}} - \mathbf{x}\|^2 \end{aligned} \quad (29)$$

Now, $\tilde{\mathbf{y}}$ is the new searching target, defined as

$$\tilde{\mathbf{y}} = P_w(\mathbf{s} - \mathbf{z}) \quad (30)$$

Let $\mathbf{e}(CBEntry, GainEntry)$ be the perceptual error vector corresponding to a codebook entry *CBEntry* and a gain entry *GainEntry*. Clearly minimizing

$$\begin{aligned} &\|\mathbf{e}(CBEntry, GainEntry)\|^2 = \\ &\|\tilde{\mathbf{y}} - \mathbf{x}(CBEntry, GainEntry)\|^2 \end{aligned} \quad (31)$$

is equivalent to minimizing Eq. (20), with $\mathbf{z}$ has been taking into account. Figure 2 shows the new structure.

2.3.4. Serial Search

To further reduce the computational cost, the scheme serially searches the ACB parameters before the SCB parameters. The system uses 512 and 128 entries for SCB and ACB, respectively, and 16 entries for each gain table. If the scheme has to search all codebooks simultaneously, it has to search through 512×128×16×16 = 16,777,216 entries. On the other hand, serial search works on 512×16 + 128×16 = 10,240 trials only.

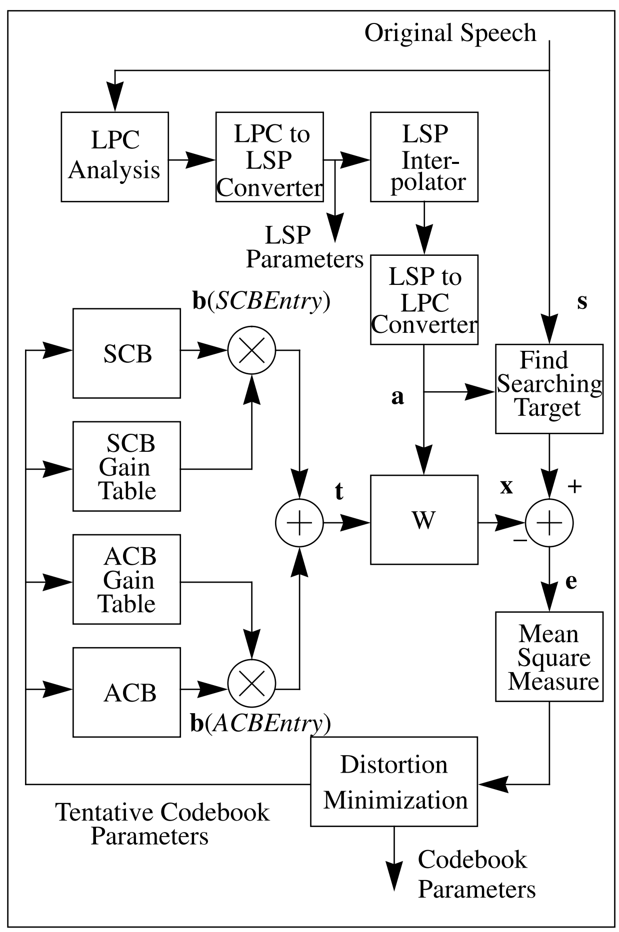

Fig. 2. Practical CELP analyzer.

Consequently, ACB and SCB searches differ in the searching targets. The searching target of ACB is $\tilde{\mathbf{y}}$ as defined in Eq. (30). The resulting ACB parameters alone can produce $\mathbf{x}$ according to Eq. (26), but they result in a high $\|\mathbf{e}\|^2$. The SCB parameters must then generate a signal that 'fills the gap' between $\tilde{\mathbf{y}}$ and such an $\mathbf{x}$. Thus, $\tilde{\mathbf{y}} - W\mathbf{t}$ becomes the SCB searching target, where $\mathbf{t}$ is obtained from Eq. (16) using newly obtain ACB parameters but without SCB parameters.

2.3.5. Joint Optimization Search

A *joint optimization* scheme suboptimally searches for codebook and gain entries in one process, thus further reducing the number of prototype trials. In minimizing $\|\mathbf{e}(CBEntry, GainEntry)\|^2$, the system should search through all combinations of *CBEntry* and *GainEntry*. However, the joint optimization scheme assigns an optimal *GainEntry* for each *CBEntry*, so that the scheme effectively searches for *CBEntry* only. In other words, instead of searching through 10,240 entries, the scheme only needs to search through 512+128 = 640 entries. This suboptimal solution saves computation in an order of magnitude. The basic approach is as follows.

1. For every codebook entry called *CBEntry*, compute $\mathbf{v}$ (sometime called the *normalized* $\mathbf{x}$, i.e. the $\mathbf{x}$ obtained with unit gain, according to
$$\mathbf{v} = W\,\mathbf{b}[CBEntry] \tag{32}$$
Here, the $\mathbf{b}[CBEntry]$ is a prototype in the codebook pointed by the *CBEntry*. This process is often called *convolution*.
2. For every *CBEntry*, compute *GainEntry* associated with the *CBEntry*. Suppose g is the gain value which scales $\mathbf{b}$ to become $\mathbf{t}$. Clearly,
$$\mathbf{x} = g\mathbf{v} \tag{33}$$
One way to minimize Eq. (31) is to maximize a *Peak* value defined in *inner-product* terms as
$$Peak = \langle \tilde{\mathbf{y}}, \mathbf{x} \rangle - \langle \mathbf{x}, \mathbf{x} \rangle = g\langle \tilde{\mathbf{y}}, \mathbf{v} \rangle - g^2 \langle \mathbf{v}, \mathbf{v} \rangle \tag{34}$$
To find the best g to maximize Eq. (34), we take a derivative of Eq. (34) with respect to g, and find its root. The root, which is the best g, turns out to be
$$g = \frac{\langle \tilde{\mathbf{y}}, \mathbf{v} \rangle}{\langle \mathbf{v}, \mathbf{v} \rangle} \tag{35}$$
Furthermore, *GainEntry* is now the index whose value in the gain table is the closest value to this g.
3. For every *CBEntry*, compute also the *Peak* value using Eq. (33).
4. Find the *CBEntry* that has the closest distance, that is one with the highest *Peak* value. This *CBEntry* and its associated *GainEntry* become the desired codebook parameters.

Notice that there are three main computational processes: the convolution to obtain $\mathbf{v}$ and the two inner products $\langle \tilde{\mathbf{y}}, \mathbf{v} \rangle$ and $\langle \mathbf{v}, \mathbf{v} \rangle$. They are called many times, as many as the codebook size. Consequently, they are the bottleneck of the system.

2.3.6. Fast Convolution with Special Codebooks

The search scheme employs a fast convolution algorithm for the convolution in Eq. (32) by exploiting the *overlapping* property of the codebook elements [KlKK90]. As a result, some of the convolution results of an entry can be used to compute convolution of the next entry. Let us design an SCB such that all the prototypes' elements come from an array $\mathbf{r}$ having 1082 elements. Suppose the elements of a prototype pointed by *CBEntry* (i.e., $\mathbf{b}(CBEntry)$) are $b_{CBEntry}[i]$ with i = 0, ..., 59. Then we force the elements to be
$$b_{CBEntry}[i] = r[2(511 - CBEntry) + i] \tag{36}$$
It can be verified that the prototypes are overlapping, i.e., most elements of a prototype are also elements of another prototype in its neighborhood.

With this special SCB, we can obtain $v_{CBEntry}[i]$ using Eq. (32) as follows
$$v_{CBEntry}[i] = \sum_{j=0}^{59} w[j-i]\, b_{CBEntry}[j] \tag{37}$$
To simplify the notation, define $u(CBEntry,i,j)$ as
$$\begin{aligned} u(CBEntry, i, j) &= w[j-i]\, b_{CBEntry}[j] \\ &= w[j-i]\, r[2(511 - CBEntry) + j] \end{aligned} \tag{38}$$
We then have
$$\begin{aligned} v_{CBEntry}[i] = {} & \sum_{j=0}^{1} u(CBEntry, i, j) + \\ & \sum_{j=2}^{61} u(CBEntry, i, j) - \\ & \sum_{j=60}^{61} u(CBEntry, i, j) \\ = {} & \text{head term} + \text{middle term} - \text{tail term} \end{aligned} \tag{39}$$
It can be verified easily that the middle term is exactly $v_{CBEntry-1}[i]$ because of the overlapping property.

This remarkable fact leads to a fast iteration for convolution. Now, instead of performing 60 terms of multiply and accumulate (MAC) operations as implied by Eq. (37), the scheme calculates a $v_{CBEntry}[i]$ in 4 MAC only to obtain the head and tail terms, and uses the previously calculated $v_{CBEntry-1}[i]$ as the middle term. The computational cost reduction is by a factor of 15.

We can even avoid having to compute the tail term if we can afford having a long array **v**' and a short array **v**" of length 1082 and 60, respectively, as shown in the following modified joint-optimization algorithm.

1. We start with computing $v_0[i]$ using the old method (Eq. (37)) as a starting point for iteration. Store the results into an empty **v**' according to

$$v'[i] = v_0[i]; \qquad i = 0, \ldots, 59 \tag{40}$$

2. Calculate *Peak* and *GainEntry* as in the joint optimization, and store them in *BestPeak* and *BestGainEntry*, respectively. Store also *CBEntry* (in this case is 0) into *BestCBEntry*.

   Then for every *CBEntry* = 1, ..., 511, perform:
3. Calculate the 60 head terms and store it in **v**".
4. Update the array **v**' according to

$$v'[i+2CBEntry] \leftarrow v'[i+2CBEntry] + v''[i] \tag{41}$$

5. Calculate *Peak* and *GainEntry* as in the joint optimization. However, get **v** from **v**' according to

$$v[i] = v'[i+2CBEntry]; \qquad i = 0, \ldots, 59 \tag{42}$$

6. Compare *Peak* with a variable *BestPeak* (predefined as zero). If current *Peak* is larger than *BestPeak*, the scheme updates *BestPeak* with *Peak*, and stores *CBEntry* and *GainEntry* in *BestCBEntry* and *BestGainEntry*, respectively.

After performing those steps for all entries, the desired parameters are available in *BestCBEntry* and *BestGainEntry*.

Further cost reduction is due to the fact that FS-1016 SCB uses $b_{CBEntry}[i]$ that is not only overlapping but also sparse (77% of the elements are 0) and ternary (i.e., the elements takes values –1, 0, and 1 only). Thus before calculating the head terms in the Step 3 above, the scheme checks if $b_{CBEntry}[j]$ is zero. In such cases, 60 computations of the term using this $b_{CBEntry}[j]$ in Step 3 are skipped. The scheme should have 77% of such cases.

With ternary $b_{CBEntry}[j]$, multiplications in computing the head terms are not necessary anymore because multiplication by 1 and -1 are equivalent with changing sign only.

Although the above example is derived for the SCB search, the ACB search can also use fast convolution. Since ACB is actually a one-tap, all-pole filter, the overlapping property is inherent in the ACB. However, the ACB elements are not ternary nor sparse, thus both calculation of the head terms and multiplications cannot be omitted. But, the calculation is fast already, because the number of MAC in its head term is one only (except in some special cases at the lower entries), instead of two as in the SCB. Furthermore, the size of the ACB we use is 128 as opposed to 512 of the SCB.

It should be clear that this fast convolution works only if we use $W(z)$ in a matrix form, otherwise we cannot have Eq. (37) and the rest of its derivations.

2.3.7. Delta Coding for ACB Parameters

Further computation reduction is possible for ACB search. Here we utilize the fact that human pitch does not suddenly change within two subframes (15 ms). This means we expect that the difference between selected codebook entries of consecutive subframes can be less than 64 entries. Thus we can employ *delta coding* that codes the entry difference only. Such coding needs a reference point. The FS-1016 uses ACB entries of odd subframes as the references and delta codes the even subframes, i.e., the entry of the second or the fourth subframe is represented by the difference between the actual entry and the previous-subframe entry. This scheme reduces the computation because the even search routine operates on a subset of the ACB only (64 entries instead of 128 entries). This scheme also reduces the bit rate since the number of bits to represent the difference is less than that to represent the actual entry.

2.4. Speech Synthesis

2.4.1. Synthesis Process

A CELP synthesizer reconstructs 240 samples of **s** from a set CELP parameters. In principle, the synthesizer must first construct the filter $H(z)$ using the interpolated LSP parameters. The synthesizer then computes the excitation impulses **t** for one subframe using the codebook parameters according Eq. (17). Finally, it applies the excitation impulses **t** to the filter $H(z)$ to synthesize 60-element speech $\hat{\mathbf{s}}$ using Eq. (2). Repeating the process three more times results in a complete 240 elements of $\hat{\mathbf{s}}$.

Since most of the steps have been explained, we just describe here the conversion of LSP to LPC parameters.

2.4.2. LSP to LPC Conversion

We want to reconstruct **a** from the interpolated LSPs ω_j. Let us define **xp** and **xq** according to

$$\left.\begin{aligned} xp_i &= \omega_{2i} \\ xq_i &= \omega_{2i+1} \end{aligned}\right\} \qquad i = 0, \ldots, 4 \tag{43}$$

The following steps then convert the LSP:

1. Recover the array **b** as in Eq. (10) according to the following equations

$$\begin{aligned}
b_5 &= 32 \\
b_4 &= -b_5 \sum_{i=0}^{4} xp_{i+1} \\
b_3 &= b_5 \sum_{i=1}^{4} \sum_{j=i+1}^{5} xp_i xp_j \\
b_2 &= -b_5 \sum_{i=0}^{3} \sum_{j=i+1}^{4} \sum_{n=j+1}^{5} xp_i xp_j xp_m \\
b_1 &= b_5 \sum_{i=1}^{2} \sum_{j=i+1}^{3} \sum_{m=j+1}^{4} \sum_{n=m+1}^{5} xp_i xp_j xp_m xp_n \\
b_0 &= -b_5 \left(xp_1 xp_2 xp_3 xp_4 xp_5\right)
\end{aligned} \tag{44}$$

2. Recover the coefficients of $p(z)$ according the following equations

$$\begin{aligned}
p_1 &= \frac{b_4}{16} \\
p_2 &= \frac{b_3 + 40}{8} \\
p_3 &= \frac{b_2 + 16p_1}{4} \\
p_4 &= \frac{b_1 + 6p_2 - 10}{2} \\
p_5 &= b_0 + 2p_3 - 2p_1
\end{aligned} \tag{45}$$

3. Recover array **c** as in Eq. (11) according the following equations

$$\begin{aligned}
c_5 &= 32 \\
c_4 &= -c_5 \sum_{i=0}^{4} xq_{i+1} \\
c_3 &= c_5 \sum_{i=1}^{4} \sum_{j=i+1}^{5} xq_i xq_j \\
c_2 &= -c_5 \sum_{i=0}^{3} \sum_{j=i+1}^{4} \sum_{n=j+1}^{5} xq_i xq_j xq_m \\
c_1 &= c_5 \sum_{i=1}^{2} \sum_{j=i+1}^{3} \sum_{m=j+1}^{4} \sum_{n=m+1}^{5} xq_i xq_j xq_m xq_n \\
c_0 &= -c_5 \left(xq_1 xq_2 xq_3 xq_4 xq_5\right)
\end{aligned} \tag{46}$$

4. Obtain set of $q(z)$ coefficients, according the following equations

$$\begin{aligned}
q_1 &= \frac{c_4}{16} \\
q_2 &= \frac{c_3 + 40}{8} \\
q_3 &= \frac{c_2 + 16q_1}{4} \\
q_4 &= \frac{c_1 + 6q_2 - 10}{2} \\
q_5 &= c_0 + 2q_3 - 2q_1
\end{aligned} \tag{47}$$

5. Finally, use **p** and **q** to construct **a** by inverting Eq. (12), as follows

$$\begin{aligned}
a_0 &= 1 \\
p_0 &= q_0 = 1 \\
a_i &= \frac{p_{i-1} + p_i + q_i - q_{i-1}}{2}; \quad i = 1, \ldots, 5 \\
a_{11-i} &= \frac{p_{i-1} + p_i + q_i - q_i}{2}; \quad i = 1, \ldots, 5
\end{aligned} \tag{48}$$

3. Computer Implementation

We can now translate the above algorithm to a computer implementation. We have coded the procedures in ANSI C routines. We briefly describe the actual program to show how a CELP system actually uses the procedures. Details of routines for codebook searching are presented in [GrLK93].

3.1. LSP Analysis

First, a routine `PCMtoFloat` converts the speech samples **s** into a floating point form, since **s** usually comes from an analog-to-digital converter with integer data format, while floating-point computation is preferred to reduce the distortion caused by finite-length registers. A routine `AnalyzeLPC` then extracts **a** from **s**, explained in Section 2.2.1. Prior to converting **a** to LSPs, the scheme calls a routine `ExpandBandwidth` to expand the bandwidth of a_i using Eq. (13) with an expanding factor γ of 0.994. This procedure ensures that **a** are within the range of the LSP table. The scheme calls `ConvertLPCToLSP` routine to obtain LSP_j from **a**, according to Section 2.2.2. Finally, a routine `CheckLSPStability` verifies the monotonicity of the LSP_j before allowing them to be used (see section 2.2.3).

3.2. Codebook Searching

A computer routine called `CodebookSearching` finds the ACB and SCB parameters. First, we must construct a 60×60 matrix W representing $W(z)$ (see Eq. (25)). The scheme starts with obtaining **a**. An `InterpolateLSP` routine provides LSPs for each individual subframe by interpolation using Eq. (19). The filter W practically requires **a** instead of LSPs, thus the scheme calls `ConvertLSPtoLPC` routine for the conversion (see Section 2.4.2). An `ExpandBandwidth` routine then performs Eq. (13) to generate **c**, with an expanding factor γ of 0.8. It is easy to show using Eq. (22) that $W(z)$ is equivalent with $H(z)$ with **c** replaces **a**. Furthermore, to represent the filter $W(z)$, W must contain the impulse responses, w_i, of the filter, as shown in Eq. (22). A `FindImpulseResponse` routine provides such elements.

Having constructed W, the scheme prepares for ACB parameter searching. The `FindACBSearchingTarget` routine determines the ACB searching target $\tilde{\mathbf{y}}$ for the current subframe using Eq. (30).

If the subframe is odd, i.e., the first or third subframe, the scheme calls the `ACBSearchingOdd` routine to get the ACB parameters, otherwise the `ACBSearchingEven` performs that function. The scheme then computes the searching target of SCB searching, by calling `FindSCBSearchingTarget`. The `SCBSearching` obtains the SCB parameters and stores them in an output buffer. The routine now has a complete set of the codebook parameters.

Before the loop proceeds for the next subframe, it must prepare and update the system states. First, an `UpdateACB` routine updates the contents of the ACB with new values from the excitation impulses to imitate the effects of delay elements in an all-pole, one-tap pitch filter. Second, the delay elements of $H(z)$ must also be updated according to those of the CELP synthesizer. At this phase, the synthesizer has stored values in its delay elements which has an additive effect to the synthetic speech produced later in the next subframe. The `GetDelayElements` tracks those values, which are later used by the next-subframe `FindACBSearchingTarget` to compensate the additive effect represented by the zero response, as discussed in Section 2.3.3. Finally, if the subframe is even, i.e., the second or fourth subframe, the `DeltaEncoding ACB` routine encodes the ACB entry using a delta coder.

Having obtained the LSPs and all entries for ACB and SCB for one frame, the scheme collects them in an FS-1016 data stream for transmission, by calling `ConvertToDataStream`. A routine `UpdatePreviousLSP` updates the contents of previous LSPs with the newly obtained LSPs to be used for interpolation (using Eq. (19)) and stability checking in the next frame.

3.3. Speech Synthesis

A synthesis program converts each FS-1016 data stream into a frame of speech. First, a routine `ConvertFromStream` unpacks the LSPs and the entries of ACB and SCB from the data stream. Since two of the ACB entries are delta coded, a routine `DeltaDecoding` obtains the actual entries.

As in the case of codebook searching, the synthesis performs a loop for four consecutive subframes. The loop starts with `InterpolateLSP` to obtain smooth transition of the LPC filter $H(z)$, using Eq. (19). A routine `ConvertLSPtoLSP` provides **a** from the LSPs to construct $H(z)$ (see Section 2.4.2). To get the excitation impulses **t**, the loop calls `UpdateACB`, which computes **t** using the ACB and SCB entries, and also updates ACB using the resulting **t**. Finally, a routine `GetDelayElements` applies the **t** to $H(z)$ to produce the speech, and at the same time, updates the delay elements of $H(z)$ to be used later for the next $H(z)$.

Before the process continues to the next frame, a routine `UpdatePreviousLSP` updates the contents of previous LSPs with the current LSPs.

4. PERFORMANCE

The algorithm presented here is fast enough for practical uses, such as store-and-forward communication, voice-mail, and multimedia. We have ported the computer program for various platforms, including TMS C30, IBM PC, SUN workstations, and IBM PowerPC based workstation. It has also been ported as a dynamic link library (DLL) for Windows 3.1, ready to be used for various speech applications.

Figure 3 shows a simple CELP compression application as an example of accessing the DLL.

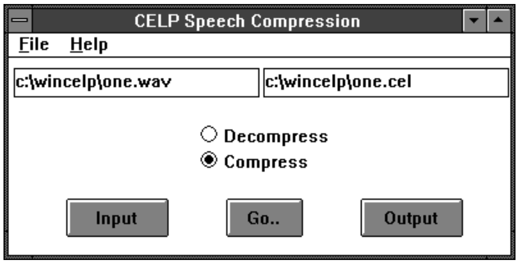

Fig. 3. A simple Windows 3.1 CELP system utilizing the CELP dynamic link library.

Table 1 shows that the execution time is within a reasonable range. On the IBM Power PC workstation, the algorithm run faster than real-time (0.85 real-time for both analysis and synthesis). The execution time of the C30 implementation is approximately two to three times real-time requirement. On the SUN Sparcstations, the routines require between five times to twice real-time requirement. Finally, a 33 Mhz IBM-PC 486DX requires approximately 14 times real-time.

Despite the use of the suboptimal algorithm to speed up the codebook searching, the synthesized speech still has high intelligibility and natural quality [Lang92]. The results from a Fairbanks rhyme test show an intelligibility score of more than 95% word correct identification. Furthermore, subjective and objective tests using male spoken Harvard sentences result in a mean opinion score (MOS) of 3.21 and a segmental signal-to-noise ratio (SEGSNR) of 10.10 dB, respectively.

Table 1. Execution time of the algorithm on various platforms, in terms of % real-time.

Platforms	Analysis Time (% real-time)	Synthesis Time (% real-time)
PC 486DX/33	1304	23
SUN Sparc 2	445	12
SUN Sparc 5	221	5
PC-AT/ TMS C30	220	5
IBM PowerPC (Power 590)	83	2

Furthermore, the size of the executable file is small. The C30 program and data require less than 11 Kwords of memory. The size of the SUN version executable file is 64 Kbytes. The algorithm can be coded modularly in C to enable tailoring it to another application.

However, the fast algorithms is quite complex, i.e., it involves many processes, loops, and variables. The efforts in reducing the computation time results in increasing the memory requirement to hold look-up tables and codebooks. The algorithm also reduces the overhead in data transfers by fixing the locations of arrays and globally using them. This increases the complexity, because data may be altered by several different processes, which means there are many processes that should be considered simultaneously.

In most platforms, a real-time application still requires faster processors. Our observation on the C30 program reveals that the codebook searching consumes 218% of real-time requirement, i.e., 2.18 seconds of codebook searching are required for every second of speech. As shown in Table 2, this results from the inner products inside the joint optimization scheme, which consumes 111% of the real-time requirement. This part should become the main attention to improve the execution speed.

However, it should be noted that the synthesis part requires only 2 to 23% of the real-time requirement of execution time in all platforms. This means the system can easily perform real-time playback. This asymmetric type of systems (i.e. systems with easy playback) has found a wide range of applications, such as in broadcasting, database, library, and CD-ROM based multimedia.

Table 2. Computation time requirements of some most demanding routines in TMSC30.

Processes	% Real Time
Inner Products	111
Joint Optimization	148
Codebook Search	218
SCB Searching Target	6

5. Discussion

This paper has described an efficient algorithm and its implementation of the CELP speech processing system. Near real-time implementation is possible using fast extraction of LSP parameters, fast searches of ACB and SCB parameters, and CELP synthesis. The codebook searches employ the joint optimization scheme, which consumes the largest block of the codebook searching computation due to a combination of the complexity of this routine and the large number of times it is called by the ACB and SCB searching algorithms. The algorithm allows high quality speech to be achieved with a bit rate of as low as 4.8 kHz. The algorithm can be readily used for CELP implementations, such as on (i) high quality low-bit rate speech transmission in point-to-point or store-and-forward (network based) mode, and (ii) efficient speech storage in speech recording or multimedia databases.

We are currently seeking hardware implementation to reduce not only the execution time, but also the physical size of the actual implementation. We are studying the algorithm for the purpose of casting some of its parts to silicon. At this stage, a full hardware implementation is premature since the optimality is not clear. However, we should focus on casting the inner products and convolution processes that have become the algorithm bottleneck. Implementing the inner product process in dedicated hardware is attractive, because it has a simple computational structure, i.e., a regular multiply and accumulate process of 60 terms. The convolution of SCB elements in Eq. (39) is also attractive for hardware implementation because the SCB elements are predefined. As explained in Section 2.3.6, the ternary property simplifies the convolution into an addition/ subtraction process with a branch controlled by SCB elements, making it easier to implement in hardware.

ACKNOWLEDGMENTS

This work was supported in part by the Natural Sciences and Engineering Research Council (NSERC) of Canada, the Manitoba Telephone System (MTS), and now by the Telecommunication Research Laboratories (TRLabs). One of the authors (AL) wishes to thank IUC-Microelectronics ITB, Laboratory of Signals and Systems ITB, and PT INTI Persero, Bandung, Indonesia, for their support in this research work.

REFERENCES

[Atal86] B. S. Atal, “High quality speech at low bit-rates: Multi-pulse and stochastically excited linear predictive coders”, in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing*, (Tokyo, Japan), IEEE CH2243-4/86, pp. 1681-1684, 1986.

[CaTW90] J. P. Campbell, Jr., T. E. Tremain, and V. C. Welch, “The proposed Federal Standard 1016 4800 bps voice coder: CELP”, *Speech Technology*, pp. 58-64, Apr./May 1990.

[GrLK93] W. Grieder, A. Langi, and W. Kinsner, “Codebook searching for 4.8 kbps CELP speech coder”, in *Proc. IEEE Wescanex 93*, pp. 397-406.

[Hahe90] R. Ragen, and P. Hedelin, “Low bit-rate spectral coding in CELP, a new LSP method”, in *Proc IEEE Int. Conf. Acoust., Speech, Signal Processing*, IEEE CH2847-2/90, pp. 189-192, 1990.

[JaJS93] N. Jayant, J. Johnston, and R. Safranek, “Signal compression based on models of human perception”, *Proceeding of IEEE*, vol. 81, no. 10, pp. 1385-1422, October 1993.

[KaRa86] P. Kabal, and R. P. Ramachandran, “The computation of line spectral frequencies using Chebyshev polynomials”, *IEEE Trans. ASSP*, vol. ASSP-34, no. 6, pp. 1419-1426, December 1986.

[KlKK90] W. B. Kleijn, D. J. Krasinski, and R. H. Ketchum, “Fast Methods for the CELP speech coding algorithm”, *IEEE Trans. ASSP*, vol 38, no. 8., pp.1330-1342, August 1990.

[LaKi90] A. Langi and W. Kinsner, “CELP High-quality speech processing for packet radio transmission and networking”, *ARRL 9th Computer Networking Conf.*, pp. 164-169, 1991.

[LaKi91] A. Langi and W. Kinsner, “Design and Implementation of CELP Speech Processing System using TMS320C30”, *ARRL 10th Computer Networking Conf.*, pp. 87-93, 1991.

[Lang92] A. Langi, “Code-Excited Linear Predictive Coding for High-Quality and Low Bit-Rate Speech”, *M.Sc. Thesis*, The University of Manitoba, Winnipeg, MB, Canada, 138 pp., 1992.

[Pars86] T. W. Parsons, *Voice and Speech Processing*. New York: McGraw-Hill, 402 pp., 1986.

[Proa83] J. G. Proakis, *Digital Communication*. New York: McGraw-Hill, 608 pp., 1983.

[PTVF92] W.H. Press, S.A. Teukolsky, W.T. Veterling, and B.P. Flannery. *Numerical Recipes in C*. (2nd ed) New York, NY: Cambridge University Press, 1992.

[ScAt85]M. R. Schroeder, and B. S. Atal, “Code excited linear prediction (CELP): high quality speech at very low bit rate,” in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing*, IEEE CH2118-8/85, vol 1, pp. 937-940, 1985.

[SoJu84] F. K. Soong, and B. -H. Juang, “Line Spectrum Pair (LSP) and speech data compression”, in *Proc IEEE Int. Conf. Acoust., Speech, Signal Processing*, IEEE CH1945-5/84, pp. 1.10.1-1.10.4, 1984.