

ANALYSIS-BY-SYNTHESIS LINEAR PREDICTIVE SPEECH CODER

TECHNICAL FIELD

The present invention relates to an analysis-by-synthesis linear predictive speech coder. Such speech coders are used in e.g. cellular radio communication systems.

BACKGROUND OF THE INVENTION

An analysis-by-synthesis speech coder [1] consists of three main components in the synthesis part, namely a linear predictive coding (LPC) synthesis filter, an adaptive code book and some type of fixed excitation. The synthesis of the speech is done by filtering an excitation vector through the LPC synthesis filter to produce the synthetic speech signal. The excitation vector is formed by adding together scaled versions of vectors coming from the adaptive code book and the fixed excitation. The analysis part of an analysis-by-synthesis coder consists mainly of the LPC analysis and the excitation analysis. The excitation analysis is a search for the indices or other parameters for the excitation, e.g. indices for the code book, gain parameters for the excitation or the amplitudes and positions for excitation pulses.

The used excitation structure in an analysis-by-synthesis speech coder is essential for the quality of the reconstructed speech, the complexity of the search and the robustness to bit errors. To achieve high quality the excitation needs to be rich, i.e. contain both pulse-like and noise-like components. To achieve low complexity the excitation needs to be somewhat structured, due to the fact that the search for the excitation code tends to be of low complexity in a structured code book. To achieve high robustness in a mobile radio environment the bit error sensitivity for the unprotected bits of the excitation code must be low.

To achieve excitation richness so called mixed excitation procedures have been proposed [2-8]. The mix usually consists of

pulse and noise sequences. Pulse-like excitations are needed in onsets, plosive and voiced sections of the speech. Noise-like sequences are needed for unvoiced sounds.

To achieve low complexity structured excitation several methods have been proposed. Multi-pulse excitation (MPE) has been described in [9] and consists of pulses described by a position and an amplitude. Regular pulse excitation (RPE) has been described in [10] and consists of a sequence of regularly (equidistant) spaced pulses described by a grid (position of the first pulse) and pulse amplitudes. Transformed binary pulse excitation (TBPE) is described in [11-12] and consists of a binary sequence of pulses that are transformed by a shaping matrix to obtain a gaussian-like sequence of regularly spaced pulses. Vector sum excitation (VSE) is described in [13] and consists of a number of basis vectors that are combined into an output vector. The basis vectors are multiplied with either +1 or -1 and summed to form the excitation vector. Low complexity search methods exist for all these structured excitations.

To achieve robustness protection of the most significant bit [14], index assignment [15] and phase position coding [16] have been proposed.

SUMMARY OF THE INVENTION

An object of the present invention is an analysis-by-synthesis linear predictive speech coder that provides both high quality (excitation richness), low search complexity and high robustness in a mobile radio environment.

This problem is solved with a speech coder in accordance with claim 1.

BRIEF DESCRIPTION OF THE DRAWINGS

The invention, together with further objects and advantages

thereof, may best be understood by making reference to the following description taken together with the accompanying drawings, in which:

- 5 **FIGURE 1** is a block diagram of a typical analysis-by-synthesis linear predictive speech coder;
- FIGURE 2** illustrates the principles of multi-pulse excitation (MPE);
- FIGURE 3** illustrates a bit allocation scheme for a multi-pulse excitation;
- 10 **FIGURE 4** is a diagram illustrating the bit error sensitivity of the multi-pulse excitation defined in Figure 3;
- FIGURE 5** a-e illustrates the principles of phase position coded multi-pulse excitation;
- 15 **FIGURE 6a** illustrates the principles of transformed binary pulse excitation (TBPE);
- FIGURE 6b** illustrates TBPE for a special case of only two pulses;
- FIGURE 7** illustrates a bit allocation scheme for a transformed binary pulse excitation;
- 20 **FIGURE 8** is a diagram illustrating the bit error sensitivity of the transformed binary pulse excitation;
- FIGURE 9** illustrates a bit allocation scheme for a combined multi-pulse and transformed binary pulse excitation in accordance with a preferred embodiment of the present invention;
- 25 **FIGURE 10** is a diagram illustrating the bit error sensitivity

of the combined multi-pulse and transformed binary pulse excitation in accordance with a preferred embodiment of the present invention;

5 FIGURE 11 compares the bit error sensitivities illustrated in Figures 4, 8 and 10, sorted by bit error sensitivity; and

FIGURE 12 is a block diagram of a preferred embodiment of a speech coder in accordance with the present invention.

10 DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

The following description will refer to the European GSM system. However, it is appreciated that the principles of the present invention may be applied to other cellular systems as well.

15 Fig. 1 shows a block diagram of a typical analysis-by-synthesis linear predictive speech coder. The coder comprises a synthesis part to the left of the vertical dashed center line and an analysis part to the right of said line. The synthesis part essentially includes two sections, namely an excitation code generating section 10 and an LPC synthesis filter 12. The
20 excitation code generating section 10 comprises an adaptive code book 14, a fixed code book 16 and an adder 18. A chosen vector $a_i(n)$ from the adaptive code book 14 is multiplied by a gain factor g_i for forming a signal $p(n)$. In the same way an excitation vector from the fixed code book 16 is multiplied by a
25 gain factor g_j for forming a signal $f(n)$. The signals $p(n)$ and $f(n)$ are added in adder 18 for forming an excitation vector $ex(n)$, which excites the LPC synthesis filter 12 for forming an estimated speech signal vector $\hat{s}(n)$.

30 In the analysis part the estimated vector $\hat{s}(n)$ is subtracted from the actual speech signal vector $s(n)$ in an adder 20 for forming an error signal $e(n)$. This error signal is forwarded to a

weighting filter 22 for forming a weighted error vector $e_w(n)$. The components of this weighted error vector are squared and summed in a unit 24 for forming a measure of the energy of the weighted error vector.

5 A minimization unit 26 minimizes this weighted error vector by choosing that combination of gain g_i and vector from the adaptive code book 14 and that gain g_j and vector from the fixed code book 16 that gives the smallest energy value, that is which after filtering in filter 12 best approximates the speech signal vector $s(n)$. This optimization is divided into two steps. In the first
10 step it is assumed that $f(n)=0$ and the best vector from the adaptive code book 14 and the corresponding g_i are determined. An algorithm for determining these parameters is given in the enclosed APPENDIX. When these parameters have been determined a
15 vector and corresponding gain g_j are chosen from the fixed code book 16 in accordance with a similar algorithm. In this case the determined parameters of the adaptive code book 14 are locked to their determined values.

The filter parameters of filter 12 are updated for each speech
20 signal frame (160 samples) by analyzing the speech signal frame in a LPC analyzer 28. This updating has been marked by the dashed connection between analyzer 28 and filter 12. Furthermore, there is a delay element 30 between the output of adder 18 and the adaptive code book 14. In this way the adaptive code book 14 is
25 updated by the finally chosen excitation vector $ex(n)$. This is done on a subframe basis, where each frame is divided into four subframes (40 samples).

As has been noted above the used excitation structure of the
fixed code book is essential for the quality of the reconstructed
30 speech, the complexity of the search and the robustness to bit errors. To achieve high quality the excitation needs to be rich, i.e. contain both pulse-like and noise-like components. To achieve low complexity the excitation needs to be somewhat structured. The search for the excitation code tends to be of

relatively low complexity in a structured code book. To achieve high robustness in a mobile radio environment the bit error sensitivity for the unprotected bits of the excitation code must be low. This is not as important for the protected (channel coded) bits of the excitation code. Thus, the bit error sensitivity in the excitation code should differ between protected and unprotected bits. Usually the unprotected class of bits will limit the performance in high BER channels.

As mentioned above high robustness may be achieved by channel coding protection, but bandwidth constraints usually limits this to 60-80% overhead for redundant channel coding of bits. Since in general a coding rate about $\frac{1}{2}$ or more is needed for high performance, not all bits may be protected. Some of the bits need to be very robust to bit errors to be sent without channel protection. Thus, the bits of the speech coding need to have strongly unequal error sensitivity. To achieve very high performance special attention has to be given to the fact that the unprotected bits usually limit the performance.

Multi-pulse excitation, which is illustrated in Fig. 2, is known to provide high quality at higher bit rates. For example 6-8 pulses per 40 samples (or 5 milliseconds) is known to give good quality. Fig. 2 illustrates 6 pulses distributed over a subframe. The excitation vector may be described by the positions of these pulses (positions 7, 9, 14, 25, 29, 37 in the example) and the amplitudes of the pulses (AMP1-AMP6 in the example). Methods for finding these parameters are described in [9]. Usually the amplitudes only represent the shape of the excitation vector. Therefore a block gain is used to represent the amplification of this basic vector shape. Fig. 3 shows an example of the format of the bit distribution of a typical multi-pulse excitation consisting of six pulses. In this example five bits are used for a scalar quantized block gain (scaling of the pulses), one bit is used for each pulse sign, two bits for the scalar quantization of each pulse amplitude and $(40 \text{ over } 6) = 22$ bits for pulse position coding using a combinatorial position coding scheme (see [1] p.

360 and Appendix) . This adds up to $5+6+12+22=45$ bits/5 ms=9 kb/s.

The bit error sensitivity of the multi-pulse excitation is known to be relatively high for some of the bits. This is illustrated in Fig. 4. The figure illustrates the signal-to-noise ratio of reconstructed speech for 100% BER in each bit position of the excitation. Thus, each bit position in the format of Fig. 3 is individually set to the wrong value, while all other bit positions are correct. The reconstructed signal is compared to the original signal and the signal-to-noise ratio is computed. Thus, the length of each line in Fig. 4 represents the sensitivity of the reconstructed speech to an error in that bit position. In the figure high SNR indicates low bit error sensitivity.

From Fig. 4 it can be seen that the most significant bits of the block gain (bits 3-5) are very sensitive to bit errors, while the least significant bits of the block gain) bits 1-2 are less sensitive. Furthermore, the signs of the pulses (bits 28, 31, 34, 37, 40 and 43) are also very sensitive to bit errors. The amplitude bits (bits 29, 30, 32, 33, 35, 36, 38, 39, 41, 42, 44 and 45) are less sensitive to bit errors. Depending on the position coding scheme used the pulse position bits (bits 6-27) are more or less sensitive to bit errors. For a combinatorial scheme, as in Figs. 3 and 4, all pulse positions are jointly coded into one code word. Bit errors in that code word will move all the pulse positions around, making many of the bits (bits 11-27) sensitive to bit errors.

One way to reduce the bit error sensitivity of the pulse position coding is to restrict the pulse positions. One coding scheme of this type is phase position coding [16]. This pulse position coding scheme has higher coding efficiency than a combinatorial scheme, but the trade off is somewhat lower speech quality. The principles of phase position coding are illustrated in Figs. 5a-e. In phase position coding the total number of positions are divided into a number of sub-blocks, 4 sub-blocks in the figure. Each sub-block contains a number of phases, ten phases in the

figure. A restriction is imposed on the allowable pulse position. There is only one pulse allowed in each phase. This means that the positions can be coded by describing the phase positions and sub-block positions of the pulses. The phase positions are coded using a combinatorial scheme. The most significant bits of the sub-block positions will have high bit error sensitivity. On the other hand, the least significant bits of the phase position code words will have lower bit error sensitivity.

In Fig. 5a-e it is assumed that the pulses are generated by the same signal as the pulses in Fig. 2. In the first step the position of the strongest pulse is determined. This corresponds to the pulse in position 7 of fig. 2. This pulse has been indicated in Fig. 5a. Since pulse position 7 corresponds to phase 7, phase 7 of all the other sub-blocks has been crossed out as a forbidden pulse position for the remaining pulses. In fig. 5b the second strongest pulse is determined in position 14, which corresponds to sub-block 2 and phase 4, which means that phase 4 is forbidden for the remaining pulses. In Fig. 5c and 5d the pulses in positions 25 and 29 are determined in a similar way. The next pulse to be determined is the pulse corresponding to the pulse in position 9 of fig. 2. However, phase 9 is now forbidden. Therefore the pulse has to be positioned in one of the phase positions that are still allowed. The position chosen is that which gives the best approximation of the target excitation. In the example the pulse is positioned in phase 8 of sub-block 1. Note that since the pulse has been shifted relative to the corresponding pulse (AMP2) in fig. 2, the amplitude may also have changed. Finally, the remaining pulse corresponding to the pulse in position 37 in fig. 2 is determined. This phase (7) is also forbidden. Instead a pulse is generated in phase position 6 of sub-block 4. This pulse has been indicated by a dashed line in fig. 5e.

One major problem with multi-pulse excitation is that the decoder at the receiving end does not know which of the pulses that are most important. The most important pulses are also the pulses

that are most sensitive to bit errors. The most important pulses are usually found first in the sequential search in the coder and usually have the largest amplitudes. However, due to the position coding the most sensitive information is spread out over the bits. This increases the level of sensitivity for all bits instead of giving an unequal bit error sensitivity, as would be desirable. One solution to this would be to split the pulses into two groups. The first group would consist of the first found pulses. This would make the first group more sensitive to bit errors. Furthermore, to split the excitation coding into two parts and using phase position coding will make the bits more unequal in bit error sensitivity. A drawback of the splitting method is that the coding efficiency of the second group is lower. Thus, a more efficient coding of the second group of the excitation is needed. Low error sensitivity is also needed, since these bits are candidates for being sent unprotected.

A stochastic code book excitation is known to provide high quality at lower bit rates than a multi-pulse excitation. However, the complexity to search a stochastic code book is high, making implementation difficult, if not impossible. Techniques to lower the complexity exist, e.g. shifted sparse code books. However, even with these techniques the complexity is still too high for higher bit rates. Another drawback is the bit error sensitivity. A single bit error will make the decoder use a totally different stochastic sequence from the code book.

The transformed binary pulse excitation (TBPE) is known to provide close to stochastic excitation efficiency at equivalent bit rates. The structure of such a code book makes the search highly efficient. The storage requirement in ROM is also low. The transformation matrices are used to make the excitation more gaussian-like. The inherent structure with regular spacing of the pulses make the excitation sparse. The main drawback of this method is that the quality drops when the low complexity search methods are kept while the code book size is increased. The regular spacing limits the increase in performance when the bit

rate is increased. TBPE is described in detail in [11-12] and is further described below with reference to Figs. 6a-b.

Fig. 6a illustrates the principles behind transformed binary pulse excitation. The binary pulse code book may comprise of
5 vectors containing for example 10 components. Each vector component points either up (+1) or down (-1) as illustrated in Fig. 6a. The binary pulse code book contains all possible combinations of such vectors. The vectors of this code book may be considered as the set of all vectors that point to the
10 "corners" of a 10-dimensional "cube". Thus, the vector tips are uniformly distributed over the surface of a 10-dimensional sphere.

Furthermore, TBPE contains one or several transformation matrices (MATRIX 1 and MATRIX 2 in Fig. 6a). These are precalculated
15 matrices stored in ROM. These matrices operate on the vectors stored in the binary pulse code book to produce a set of transformed vectors. Finally, the transformed vectors are distributed on a set of excitation pulse grids. The result is four different versions of regularly spaced "stochastic" code
20 books for each matrix. A vector from one of these code books (based on grid 2) is shown as a final result in Fig. 6a. The object of the search procedure is to find the binary pulse code book index of the binary code book, the transformation matrix and the excitation pulse grid that together give the smallest
25 weighted error.

The matrix transformation step is further illustrated in Fig. 6b. In this case the binary pulse code book is assumed to consist of only two positions (this is an unrealistic assumption, but it helps to illustrate the principles behind the transformation
30 step). All the possible binary vectors of the binary pulse code book are illustrated in the left part of Fig. 6b. These vectors may be considered as being equivalent to vectors pointing to the corners of a 2-dimensional "cube", which is a square, that has been indicated by dotted lines in the left part of Fig. 6b. These

vectors are now transformed by a matrix. This matrix may for example be an orthogonal matrix, which rotates the entire "cube". The transformed binary vectors comprise the projections of the individual transformed vectors on the X- and Y-axes, respectively. The resulting transformed binary code is illustrated in the right part of Fig. 6b. After transformation the transformed vectors are distributed on a set of grids, as explained with reference to Fig. 6a.

Fig. 7 shows the bit allocation format of a typical TBPE excitation. In this example a two stage TBPE code book is used, in which TBPE code book 1 is a 40 sample code book and the second stage is divided into two 20 sample TBPE code books 2A, 2B. Code book 1 uses ten bits for the binary pulse code book index, two bits for the grids of code book 1, one bit for the matrices of code book 1 and four bits for the gain of code book 1. Code books 2A, 2B use 2x6 bits for binary pulse code book indices, 2x2 bits for code book grids, 2x2 bits for code book matrices and 2x4 bits for code book gains. This adds up to 45 bits/5 ms= 9 kb/s.

The bit error sensitivity for the transformed binary pulse excitation defined in Fig. 7 is shown in Fig. 8. The inherent structure of TBPE gives a gray-coded index in the binary pulse code books. This means that code words close in hamming distance are also close in excitation vector distance. A single bit error will only change the sign of one of the regular pulses. Therefore the bit positions in the index have roughly equal sensitivity in Fig. 8 (bits 1-10 for binary pulse code book 1, bits 18-23 for binary pulse code book 2A and bits 32-37 for binary pulse code book 2B). The first code book including index, grid and matrix (bits 1-10, 11-12, 13) has higher sensitivity. The matrix bit (bit 13) shows a very high sensitivity in this example. Furthermore, the code book gain of the first code book (bits 14-17) shows higher sensitivity than the second code book gains (bits 28-31, 42-45). One problem is that the sensitivity is spread out over the bits. The sensitivity is generally lower than for multi-pulse excitation bits, but there is only a weakly unequal error

sensitivity. However, the structure combines inherent index assignment and low complexity. This makes TBPE a strong candidate for replacing the second part of the multi-pulse excitation discussed above.

5 The structure proposed in the present invention is a mixed excitation using a few multi-pulses and a TBPE code book. The positions of the pulses are preferably coded with a restricted position coding scheme, such as phase position coding described above. The mixed excitation using pulses and transformed binary pulse (noise) sequences improve quality. The MPE and TBPE
10 searches are low complexity schemes. The mix of multi-pulse bits and TBPE shows strongly unequal error sensitivity, which fits into an unequal error protection scheme with some bits unprotected.

15 Fig. 9 illustrates an example of the format of the bit allocation in a preferred embodiment of the present invention. In this example there are three multi-pulses and one 13 bit index (13 binary pulses) TBPE code book with four grids and two matrices. Phase position coding is performed using ten sub-blocks and four
20 phases. This gives $3 \times 2 \log(10) = 10$ bits for the sub-block positions and $(4 \text{ over } 3) = 2$ bits for the phase code words, 3×1 bits for the pulse signs, 3×2 bits for the pulse amplitudes, four bits for the block gain, 13 bits for the binary pulse code book index, 2 bits for the grid, 1 bit for the matrix and four bits
25 for the code book gain. This all adds up to $10+2+3+6+4+13+2+1+4 = 45$ bits/5 ms=9 kb/s.

Fig. 10 illustrates the bit sensitivity of the mixed excitation in accordance with the preferred embodiment of the invention. From Fig. 10 it is apparent that the few multi-pulses (bits 1-21)
30 are more sensitive to bit errors than the TBPE code book index (bits 26-41). The phase position coding makes some of the bits for the pulse positioning less sensitive to bit errors (bits 1-3 of the sub-block positions and bits 11-12 of the phase code words). The amplitudes of the pulses (bits 14-15, 17-18, 20-21)

are less sensitive than the signs (bits 13, 16, 19). The bits in the TBPE index (bits 26-38) are equal in sensitivity and the sensitivity is very low compared to the pulse signs and positions. Some of the bits of the multi-pulse block gain (bits 24-25) are more sensitive. The bit for the transformation matrix (bit 41) is also sensitive.

The three schemes discussed in this application and illustrated in Figs. 4, 8 and 10 are compared with respect to error sensitivity in Fig. 11. In Fig. 11 the bits of each scheme have been sorted in bit error sensitivity order from highest to lowest sensitivity. From Fig. 11 it can be seen that the multi-pulse excitation (MPE) and the mixed excitation (MPE & TBPE) have the strongest unequal error sensitivity. The TBPE excitation has the most even sensitivity, and this sensitivity is generally lower than for the MPE excitation. The mixed excitation generally has lower sensitivity than the multi-pulse excitation, which makes the mixed excitation more robust. The mixed excitation also has some very sensitive bits (bits 1-12) and the some insensitive bits (bits 25-45), which makes this excitation perfect for unequal error protection. Since the number of unsensitive bits is larger for the mixed excitation than for the multi-pulse excitation, the performance of the unprotected class of bits will be better in low quality channels.

Fig. 12 illustrates a preferred embodiment of a speech coder in accordance with the present invention. The essential difference between the speech coder of Fig. 1 and the speech coder of Fig. 12 is that the fixed code book 16 of Fig. 1 has been replaced by a mixed excitation generator 32 comprising the multi-pulse excitation (MPE) generator 34 and a transformed binary pulse excitation (TBPE) generator 36. The corresponding block gains have been denoted g_m and g_r , respectively, in Fig. 12. The excitations from generators 34, 36 are added in an adder 38, and the mixed excitation is added to the adaptive code book excitation in adder 18.

An example of an algorithm used in the mixed excitation coder structure in accordance with the present invention is shown below. The algorithm contains all parts that are relevant in a speech encoder. The algorithm consists of six main sections. The MPE and TBPE sections, which constitute the mixed excitation are expanded to show the contents of the mixed excitation structure analysis. One frame based section, e.g. for each 160 sample frame, is the LPC analysis section, which calculates and quantizes the short-term synthesis filter. The remaining five sections are sub-frame based, e.g. they are performed for each 40 sample sub-frame. The first of these is the sub-frame preprocessing, i.e. parameter extraction; the second is the long-term analysis or adaptive code book analysis; the third is the MPE analysis; the fourth is the TBPE analysis; and the fifth is the state update.

EXAMPLE ALGORITHM**LPC analysis****For each subframe (1-4) do****Subframe preprocessing**5 **LTP analysis (adaptive code book search)****Multi-pulse excitation (MPE)****Calculate impulse response of weighting filter****Calculate autocorrelation function of impulse response****Calculate cross correlation function between impulse**10 **response and weighted residual after LTP analysis****Search MPE positions and amplitudes****Quantize amplitudes and block gain****Make MPE innovation vector****Form position code words**15 **Form new weighted residual after MPE analysis****Transformed binary pulse excitation (TBPE)****Calculate impulse response of weighting filter****Calculate cross correlation function between impulse****response and weighted residual after MPE analysis**20 **For each matrix do****For each grid do****Calculate matrix cross correlation function****Approximate pulses with sign of cross
correlation function**25 **Form weighted TBPE innovation and compare****Form TBPE code words****Quantize TBPE gain****Form TBPE innovation vector****State update**30 **A detailed description of this algorithm may be found in the
enclosed C++ program listing.**

It will be understood by those skilled in the art that various
modifications and changes may be made to the present invention
without departure from the spirit and scope thereof, which is
35 defined by the appended claims.

APPENDIX

This APPENDIX summarizes an algorithm for determining the best adaptive code book index i and the corresponding gain g_i in an exhaustive search. The signals are also shown in Fig. 1.

5	$ex(n) = p(n)$	Excitation vector ($f(n) = 0$)
	$p(n) = g_i \cdot a_i(n)$	Scaled adaptive code book vector
	$\hat{s}(n) = h(n) * p(n)$	Synthetic speech (* = convolution)
10	$e(n) = s(n) - \hat{s}(n)$	Error vector
	$e_w(n) = w(n) * (s(n) - \hat{s}(n))$	Weighted error
	$E = \sum [e_w(n)]^2 \quad n=0..N-1$	Squared weighted error
	$N = 40$ (for example)	Vector length
	$s_w(n) = w(n) * s(n)$	Weighted speech
15	$h_w(n) = w(n) * h(n)$	Weighted impulse response for synthesis filter
	$\min E_i = \min \sum_{n=0}^{N-1} [e_{w,i}(n)]^2$	Search optimal index in the adaptive code book

$$\frac{\partial E_i}{\partial g_i} = 0 \quad \rightarrow \quad g_i = \frac{\sum_{n=0}^{N-1} s_w(n) \cdot a_i(n) * h_w(n)}{\sum_{n=0}^{N-1} [a_i(n) * h_w(n)]^2} \quad \text{Gain for index } i$$

C++ PROGRAM LISTINGS

F_SpeMain.cc

```

/*
 * class F_SpeMain
 *
 * main class for speech encoder
 *
 * COPYRIGHT (C) 1995 ERICSSON RADIO SYSTEMS AB
 */
#include "F_SpeMain.hh"
#include <iostream.h>

F_SpeMain::F_SpeMain(const FloatVec& inTemp) :
    F_hugeSpeechFrame(F_hugeFrameLength),
    F_lspPrev(F_nrCoeff),
    F_ltpHistory(F_historyLength),
    F_weightFilterRingState(F_nrCoeff)
{
    for (int i=0; i<F_frameLength; i++)
        F_hugeSpeechFrame[i] = 0.0;

    /*
     * insert first 'delay' samples to be compatible with prestudy
     * coder
     */
    for (i=F_frameLength; i<F_hugeFrameLength; i++)
        F_hugeSpeechFrame[i] = inTemp[i-F_frameLength];

    for (i = 0; i < F_nrCoeff; i++)
        F_lspPrev[i] = F_lspInit[i];

    for (i=0; i<F_historyLength; i++)
        F_ltpHistory[i] = 0.0;

    for (i=0; i<F_nrCoeff; i++)
        F_weightFilterRingState[i] = 0.0;
}

void F_SpeMain::main(const FloatVec& F_speechFrame,
                    ShortVec& F_analysisData)
{
    /* local variables */
    FloatVec F_lspCurr(F_nrCoeff);
    ShortVec F_lspVQCodes(F_nLspTables);
    Float F_energy;
    Shortint F_energyCode;
    ShortVec F_ltpLagCodes(F_nrOfSubframes);
    ShortVec F_ltpGainCodes(F_nrOfSubframes);
    ShortVec F_mpeBlockMaxCodes(F_nrOfSubframes);
    ShortVec F_mpeAmpCodes(F_nrOfSubframes);
    ShortVec F_mpeSignCodes(F_nrOfSubframes);
    ShortVec F_mpePositionCodes(F_nrOfSubframes);
    ShortVec F_tbpeGainCodes(F_nrOfSubframes);

```

```

ShortVec F_tbpeGridCodes(F_nrofSubframes);
ShortVec F_tbpeMatrixCodes(F_nrofSubframes);
ShortVec F_tbpeIndexCodes(F_nrofSubframes);

F_speFrame.main(F_speechFrame,           /* in */
               F_lspPrev,               /* in */
               F_hugeSpeechFrame,       /* in/out */
               F_lspCurr,               /* out */
               F_lspVQCodes,            /* out */
               F_energy,                 /* out */
               F_energyCode);           /* out */

for (int F_subframeNr=0; F_subframeNr<F_nrofSubframes;
     F_subframeNr++) {

    /* subframe local variables */
    Float F_excNormFactor;
    FloatVec F_wCoeff(F_nrcoeff);
    FloatVec F_wSpeechSubframe(F_subframeLength);

    FloatVec F_ltpExcitation(F_subframeLength);
    FloatVec F_wLtpResidual(F_subframeLength);

    FloatVec F_mpeInnovation(F_subframeLength);
    FloatVec F_wMpeResidual(F_subframeLength);

    FloatVec F_tbpeInnovation(F_subframeLength);

    F_speSubPre.main(F_hugeSpeechFrame,   /* in */
                    F_subframeNr,        /* in */
                    F_lspCurr,           /* in */
                    F_lspPrev,           /* in */
                    F_energy,            /* in */
                    F_weightFilterRingState, /* in */
                    F_excNormFactor,     /* out */
                    F_wCoeff,            /* out */
                    F_wSpeechSubframe);  /* out */

    F_speSubLtp.main(F_wSpeechSubframe,   /* in */
                    F_wCoeff,            /* in */
                    F_ltpHistory,         /* in */
                    F_wLtpResidual,       /* out */
                    F_ltpExcitation,      /* out */
                    F_ltpLagCodes[F_subframeNr], /* out */
                    F_ltpGainCodes[F_subframeNr]); /* out */

    F_speSubMpe.main(F_wCoeff,           /* in */
                    F_excNormFactor,     /* in */
                    F_wLtpResidual,       /* in */
                    F_mpeInnovation,      /* out */
                    F_mpePositionCodes[F_subframeNr], /* out */
                    F_mpeAmpCodes[F_subframeNr], /* out */
                    F_mpeSignCodes[F_subframeNr], /* out */
                    F_mpeBlockMaxCodes[F_subframeNr], /* out */
                    F_wMpeResidual);     /* out */
}

```

```
F_speSubTbpe.main(  
    F_wMpeResidual,          /* in */  
    F_wCoeff,                /* in */  
    F_excNormFactor,         /* in */  
    F_tbpeInnovation,        /* out */  
    F_tbpeGainCodes[F_subframeNr], /* out */  
    F_tbpeIndexCodes[F_subframeNr], /* out */  
    F_tbpeGridCodes[F_subframeNr], /* out */  
    F_tbpeMatrixCodes[F_subframeNr]); /* out */  
  
F_speSubPost.main(F_ltpExcitation, /* in */  
    F_tbpeInnovation, /* in */  
    F_mpeInnovation, /* in */  
    F_wCoeff, /* in */  
    F_ltpHistory, /* in/out */  
    F_weightFilterRingState); /* out */  
}  
  
F_spePost.main(F_lspCurr, /* in */  
    F_energyCode, /* in */  
    F_lspVQCodes, /* in */  
    F_ltpGainCodes, /* in */  
    F_ltpLagCodes, /* in */  
    F_mpeBlockMaxCodes, /* in */  
    F_mpeAmpCodes, /* in */  
    F_mpeSignCodes, /* in */  
    F_mpePositionCodes, /* in */  
    F_tbpeGainCodes, /* in */  
    F_tbpeIndexCodes, /* in */  
    F_tbpeMatrixCodes, /* in */  
    F_tbpeGridCodes, /* in */  
    F_lspPrev, /* out */  
    F_analysisData); /* out */  
}
```

F_SpeSubMpe.cc

```
/*
 * class F_SpeSubMpe
 *
 * Multipulse innovation analysis
 *
 * COPYRIGHT (C) 1995 ERICSSON RADIO SYSTEMS AB
 */
#include "F_SpeSubMpe.hh"
#include "ShortVec.hh"
#include <iostream.h>
#include <stdlib.h>
#include <math.h>

F_SpeSubMpe::F_SpeSubMpe()
{

void F_SpeSubMpe::main(const FloatVec& F_wCoeff,
                      const Float F_excNormFactor,
                      const FloatVec& F_wLtpResidual,
                      FloatVec& F_mpeInnovation,
                      Shortint& F_mpePositionCode,
                      Shortint& F_mpeAmpCode,
                      Shortint& F_mpeSignCode,
                      Shortint& F_mpeBlockMaxCode,
                      FloatVec& F_wMpeResidual)
{
    /* temporary variables */
    FloatVec F_impResp(F_mpeTruncLen);
    FloatVec F_autoCorr(F_subframeLength);
    FloatVec F_crossCorr(F_subframeLength);
    FloatVec F_crossCorrUpd(F_subframeLength);
    FloatVec F_pulseAmp(F_nMpePulses);
    ShortVec F_posTaken(F_subframeLength);
    ShortVec F_mpePosVector(F_nMpePulses);
    ShortVec F_mpeAmpVector(F_nMpePulses);
    ShortVec F_mpeSignVector(F_nMpePulses);

    /* calculate impulse response */
    F_calcImpResp(    F_wCoeff,
                    F_impResp);

    /* calculate autocorrelation */
    F_autoCorrelate(    F_impResp,
                      F_autoCorr);

    /* calculate cross correlation */
    F_crossCorrelate(    F_impResp,
                      F_wLtpResidual,
                      F_crossCorr);

    /* initialize and search first pulse */
    F_searchInit(    F_crossCorr,
                    F_autoCorr,
```

```

        F_crossCorrUpd,
        F_mpePosVector,
        F_pulseAmp,
        F_postTaken);

/* search rest of pulses */
F_searchRest(    F_autoCorr,
                 F_crossCorr,
                 F_crossCorrUpd,
                 F_mpePosVector,
                 F_pulseAmp,
                 F_postTaken);

/* quantize blockmax and pulse amplitudes */
F_openLoopQuantize( F_excNormFactor,
                   F_pulseAmp,
                   F_mpeAmpVector,
                   F_mpeSignVector,
                   F_mpeBlockMaxCode);

/* make innovation vector */
F_makeInnVector(  F_pulseAmp,
                  F_mpePosVector,
                  F_mpeInnovation);

/* order pulse position */
F_orderPositions( F_mpePosVector,
                  F_mpeAmpVector,
                  F_mpeSignVector);

/* make codewords position */
F_makeCodeWords( F_mpePosVector,
                 F_mpePositionCode,
                 F_mpeAmpVector,
                 F_mpeAmpCode,
                 F_mpeSignVector,
                 F_mpeSignCode);

/* make new weighed residual */
F_makeMpeResidual( F_mpeInnovation,
                  F_wCoeff,
                  F_wLtpResidual,
                  F_wMpeResidual);
}

Shortint F_SpeSubMpe::F_maxMagIndex(const FloatVec& F_corrVec,
                                   const ShortVec& F_postTaken)
{
    /* find index for maximum mag of vector
       excluding used positions */

    /* temporary variables */

```

```

Float max;
Float temp;
int maxI;

/* move to possible position */
for (int i = 0; i < F_subframeLength && F_postTaken[i]; i++) ;

max = fabs(F_corrVec[i]);
maxI = i;

while (i < F_subframeLength) {
    temp = fabs(F_corrVec[i]);
    if (!F_postTaken[i] && temp > max) {
        max = temp;
        maxI = i;
    }
    i++;
}

return maxI;
}

void F_SpeSubMpe::F_solveNewAmps(const FloatVec& F_a,
                                const FloatVec& F_c,
                                const Shortint F_nPulse,
                                FloatVec& F_b)
{
    /* Temporary variables */
    Float den;

    switch (F_nPulse) {
    case 1:
        /* This switch is obsolete in this implementation */
        cerr << "F_SpeSubMpe::F_solveNewAmps case 1 should never occur" << endl;
        exit(-1);
    case 2:
        den = F_a[0]*F_a[0]-F_a[1]*F_a[1];
        if (den == 0.0)
        {
            cerr << "MPE singular matrix" << endl;
            break;
        }
        Float denInv = 1.0/den;
        F_b[0] = (F_c[0]*F_a[0] - F_c[1]*F_a[1]) * denInv;
        F_b[1] = (F_c[1]*F_a[0] - F_c[0]*F_a[1]) * denInv;
        break;
    case 3:
        /* Kramers rule */
        den = F_a[0]*F_a[0]*F_a[0]+F_a[1]*F_a[3]*F_a[2]+
            F_a[2]*F_a[1]*F_a[3]-F_a[1]*F_a[1]*F_a[0]-
            F_a[0]*F_a[3]*F_a[3]-F_a[2]*F_a[0]*F_a[2];
        if (den == 0.0)
        {

```

```

        cerr << "MPE singular matrix" << endl;
        break;
    }
    denInv = 1.0/den;
    F_b[0] = (F_c[0]*F_a[0]*F_a[0]+F_c[1]*F_a[3]*F_a[2]+
             F_c[2]*F_a[1]*F_a[3]-F_c[1]*F_a[1]*F_a[0]-
             F_c[0]*F_a[3]*F_a[3]-F_c[2]*F_a[0]*F_a[2])*
             denInv;
    F_b[1] = (F_a[0]*F_c[1]*F_a[0]+F_a[1]*F_c[2]*F_a[2]+
             F_a[2]*F_c[0]*F_a[3]-F_a[1]*F_c[0]*F_a[0]-
             F_a[0]*F_c[2]*F_a[3]-F_a[2]*F_c[1]*F_a[2])*
             denInv;
    F_b[2] = (F_a[0]*F_a[0]*F_c[2]+F_a[1]*F_a[3]*F_c[0]+
             F_a[2]*F_a[1]*F_c[1]-F_a[1]*F_a[1]*F_c[2]-
             F_a[0]*F_a[3]*F_c[1]-F_a[2]*F_a[0]*F_c[0])*
             denInv;
    break;
}
}

void F_SpeSubMpe::F_updateCrossCorr(const FloatVec& F_autoCorr,
                                     const Shortint F_pos,
                                     const Float F_gain,
                                     FloatVec& F_crossCorrUpd)
{
    /* temporary variables */
    int i;
    int temp;

    /* update crosscorrelation vector */
    temp = -F_mpeTruncLen + F_pos + 1;
    if (temp < 0)
        temp = 0;
    for (i = temp; i < F_pos; i++)
        F_crossCorrUpd[i] = F_crossCorrUpd[i] -
                             F_gain*F_autoCorr[F_pos-i];

    temp = F_pos+F_mpeTruncLen;
    if (temp > F_subframeLength)
        temp = F_subframeLength;
    for (i = F_pos; i < temp; i++)
        F_crossCorrUpd[i] = F_crossCorrUpd[i] -
                             F_gain*F_autoCorr[i-F_pos];
}

void F_SpeSubMpe::F_calcImpResp(const FloatVec& F_wCoeff,
                                 FloatVec& F_impResp)
{
    /* temporary variables */
    FloatVec state(F_nrCoeff);
    int i, m;
    Float signal;

    /* calculate impulse response */
    for (i=0; i < F_nrCoeff; i++)

```

```

    state[i] = 0;

    signal = 1.0;
    for (i=0; i < F_mpeTruncLen; i++) {
        for (m=F_nrCoeff-1; m>0; m--) {
            signal -= F_wCoeff[m]*state[m];
            state[m] = state[m-1];
        }
        signal -= F_wCoeff[0]*state[0];
        state[0] = signal;
        F_impResp[i] = signal;
        signal = 0;
    }
}

```

```

void F_SpeSubMpe::F_autoCorrelate(const FloatVec& F_impResp,
                                   FloatVec& F_autoCorr)
{
    /*temporary variables */
    int i, j;

    /* calculate autocorrelation vector */
    for (i=0; i < F_mpeTruncLen; i++) {
        F_autoCorr[i] = 0.0;
        for (j=i; j < F_mpeTruncLen; j++)
            F_autoCorr[i] = F_autoCorr[i] +
                F_impResp[j]*F_impResp[j-i];
    };
    for (i=F_mpeTruncLen; i < F_subframeLength; i++)
        F_autoCorr[i] = 0.0;
}

```

```

void F_SpeSubMpe::F_crossCorrelate(
                                   const FloatVec& F_impResp,
                                   const FloatVec& F_wSpeechSubframe,
                                   FloatVec& F_crossCorr)
{
    /* temporary variables */
    int i, j, lastpos;

    /* calculate crosscorrelation vector */
    for (i=0; i < F_subframeLength; i++){
        F_crossCorr[i] = 0.0;
        lastpos = i+F_mpeTruncLen;
        if (lastpos > F_subframeLength)
            lastpos = F_subframeLength;
        for (j=i; j < lastpos; j++)
            F_crossCorr[i] =
                F_crossCorr[i] + F_wSpeechSubframe[j]*F_impResp[j-i];
    }
}

```

```

void F_SpeSubMpe::F_searchInit(const FloatVec& F_crossCorr,
                                const FloatVec& F_autoCorr,
                                FloatVec& F_crossCorrUpd,
                                ShortVec& F_mpePosition,
                                FloatVec& F_pulseAmp,
                                ShortVec& F_postTaken)
{
    /* temporary variables */
    int pos, i;

    /* search init */
    for (i=0; i < F_nMpePulses; i++)
        F_pulseAmp[i] = 0.0;

    for (i=0; i < F_subframeLength; i++)
        F_postTaken[i] = 0;

    /* get first position */
    pos = F_maxMagIndex(F_crossCorr, F_postTaken);

    F_mpePosition[0] = pos;
    F_postTaken[pos] = pos+1;

    for (i=0; i < F_subframeLength; i++)
        F_crossCorrUpd[i] = F_crossCorr[i];

    F_pulseAmp[0] = F_crossCorr[pos]/F_autoCorr[0];

    F_updateCrossCorr(F_autoCorr,
                      pos,
                      F_pulseAmp[0],
                      F_crossCorrUpd);
}

void F_SpeSubMpe::F_searchRest(const FloatVec& F_autoCorr,
                                const FloatVec& F_crossCorr,
                                FloatVec& F_crossCorrUpd,
                                ShortVec& F_mpePosVector,
                                FloatVec& F_pulseAmp,
                                ShortVec& F_postTaken)
{
    /* search rest of pulses (optmethod 2) */

    /* temporary variables */
    FloatVec F_corrTerms(F_nMpePulses+1);
    FloatVec F_crossCorrTerms(F_nMpePulses);
    int pulse;
    int i, j;
    int pos;

    for (pulse=1 ; pulse < F_nMpePulses; pulse++) {
        /* get position with maximum value */
        pos = F_maxMagIndex(F_crossCorrUpd, F_postTaken);
        F_mpePosVector[pulse] = pos;
        F_postTaken[pos] = pos+1;
    }
}

```

```

/* set up vector using autoCorr */
F_corrTerms[0] = F_autoCorr[0];
for (i=0; i < pulse+1; i++)
    for (j=0; j < i; j++)
        F_corrTerms[i+j] =
            F_autoCorr[abs(F_mpePosVector[i] -
                           F_mpePosVector[j])];

/* set up vector using crossCorr */
for (i=0; i < pulse+1; i++)
    F_crossCorrTerms[i] = F_crossCorr[F_mpePosVector[i]];

/* solve for new optimal amplitudes */
F_solveNewAmps(F_corrTerms,
               F_crossCorrTerms,
               pulse+1, F_pulseAmp);

if (pulse != (F_nMpePulses-1)) {
    for (i=0; i < F_subframeLength; i++)
        F_crossCorrUpd[i] = F_crossCorr[i];
    for (i=0; i <= pulse; i++)
        F_updateCrossCorr(F_autoCorr,
                           F_mpePosVector[i],
                           F_pulseAmp[i],
                           F_crossCorrUpd);
}
}

void F_SpeSubMpe::F_openLoopQuantize(const Float& F_excNormFactor,
                                      FloatVec& F_pulseAmp,
                                      ShortVec& F_mpeAmpVector,
                                      ShortVec& F_mpeSignVector,
                                      Shortint& F_mpeBlockMaxCode)
{
    /* temporary variables */
    Float blockMax;
    Float idealBlockMax;
    Float blockMaxNorm;
    Float normPulseAmp;
    int pulse;
    Float temp;

    /* get blockmax value */
    blockMax = 0.0;
    for (pulse=0; pulse < F_nMpePulses; pulse++) {
        temp = fabs(F_pulseAmp[pulse]);
        if (temp > blockMax)
            blockMax = temp;
    }
    idealBlockMax = blockMax;

    /* quantize blockmax */
    blockMaxNorm = blockMax / F_excNormFactor;

```

[illegible]

```

/* temporary variables */
ShortVec tempPosVector(F_nMpePulses);
ShortVec tempAmpVector(F_nMpePulses);
ShortVec tempSignVector(F_nMpePulses);
int maxVal;
int maxI = 0;
int i, j;

/* Create temporary vectors */
for (i = 0; i < F_nMpePulses; i++) {
    tempPosVector[i] = F_mpePosVector[i];
    tempAmpVector[i] = F_mpeAmpVector[i];
    tempSignVector[i] = F_mpeSignVector[i];
}
/* fix ordering, the positions are ordered decreasingly */
for (i = 0; i < F_nMpePulses; i++) {
    maxVal = -1;
    for (j = 0; j < F_nMpePulses; j++) {
        if (tempPosVector[j] > maxVal) {
            maxVal = tempPosVector[j];
            maxI = j;
        }
    }
    /* exclude found vector from search */
    tempPosVector[maxI] = -10;
    /* order pulses */
    F_mpePosVector[i] = maxVal;
    F_mpeAmpVector[i] = tempAmpVector[maxI];
    F_mpeSignVector[i] = tempSignVector[maxI];
}
}

void F_SpeSubMpe::F_makeCodeWords(const ShortVec& F_mpePosVector,
                                   Shortint& F_mpePositionCode,
                                   const ShortVec& F_mpeAmpVector,
                                   Shortint& F_mpeAmpCode,
                                   const ShortVec& F_mpeSignVector,
                                   Shortint& F_mpeSignCode)
{
    /* temporary variables */
    int i;

    /* code position vector into 14 bits */
    F_mpePositionCode = 0;
    for (i = 0; i < F_nMpePulses; i++)
        F_mpePositionCode = F_mpePositionCode +
            F_mpeCombTable[(F_nMpePulses - i - 1)*F_subframeLength+
                            F_mpePosVector[i]];

    F_mpeSignCode = 0;
    for (i = 0; i < F_nMpePulses ; i++)
        F_mpeSignCode |= (F_mpeSignVector[i] << i);

    F_mpeAmpCode = 0;
    for (i = 0; i < F_nMpePulses ; i++)

```

```
F_mpeAmpCode |= (F_mpeAmpVector[i] << i*F_mpeAmpBits);
}

void F_SpeSubMpe::F_makeMpeResidual(
    const FloatVec& F_mpeInnovation,
    const FloatVec& F_wCoeff,
    const FloatVec& F_wLtpResidual,
    FloatVec& F_wMpeResidual)
{
    /* temporary variables */
    int i, m;
    Float signal;
    FloatVec state(F_nrCoeff);

    /* set zero state */
    for (i=0; i < F_nrCoeff; i++)
        state[i] = 0.0;

    /* calculate new target for subsequent TBPE search */
    for (i=0; i < F_subframeLength; i++) {
        signal = F_mpeInnovation[i];
        for (m=F_nrCoeff-1; m>0; m--) {
            signal -= F_wCoeff[m]*state[m];
            state[m] = state[m-1];
        }
        signal -= F_wCoeff[0]*state[0];
        state[0] = signal;
        F_wMpeResidual[i] = F_wLtpResidual[i]-signal;
    }
}
```

F_SpeSubTbpe.cc

```

/*
 * class F_SpeSubTbpe
 *
 * Transformed Binary Pulse Excited codebook
 *
 * COPYRIGHT (C) 1995 ERICSSON RADIO SYSTEMS AB
 *
 */
#include "F_SpeSubTbpe.hh"
#include <iostream.h>

F_SpeSubTbpe::F_SpeSubTbpe()
{

void F_SpeSubTbpe::main(const FloatVec& F_wMpeResidual,
                        const FloatVec& F_wCoeff,
                        const Float      F_excNormFactor,
                        FloatVec& F_tbpeInnovation,
                        Shortint& F_tbpeGainCode,
                        Shortint& F_tbpeIndexCode,
                        Shortint& F_tbpeGridCode,
                        Shortint& F_tbpeMatrixCode)
{
    Float F_gain = F_search(F_wMpeResidual,
                            F_wCoeff,
                            F_tbpeInnovation,
                            F_tbpeIndexCode,
                            F_tbpeGridCode,
                            F_tbpeMatrixCode);
    Float F_normGain = F_gain / F_excNormFactor;
    F_tbpeGainCode = F_quantize(F_normGain);
    Float F_tbpeGain = F_excNormFactor *
        F_tbpeGainQuantTable[F_tbpeGainCode];
    for(Shortint i = 0; i < F_subframeLength; i++)
        F_tbpeInnovation[i] = F_tbpeInnovation[i] * F_tbpeGain;
}

void F_SpeSubTbpe::F_crossCorr(const FloatVec& v1,
                               const FloatVec& v2,
                               FloatVec& F_corr)
{
    for (Shortint i = 0; i < F_subframeLength; i++) {
        Float acc = 0.0;
        for (Shortint j = i; j < F_subframeLength; j++)
            acc += v1[j] * v2[j - i];
        F_corr[i] = acc;
    }
}

```

```

void F_SpeSubTbpe::F_crossCorrOfTransfMatrix(
    const FloatVec& v1,
    const Shortint grid,
    const Shortint matrix,
    FloatVec& F_crossCorr)
{
    for (Shortint m = 0; m < F_nrTbpePulses; m++) {
        Float acc = 0.0;
        for (Shortint n = 0; n < F_nrTbpePulses; n++)
            acc += v1[grid + n * F_tbpeGridSpace] *
                F_tbpeTransfTable[(m+matrix * F_nrTbpePulses) *
                    F_nrTbpePulses + n];
        F_crossCorr[m] = acc;
    }
}

void F_SpeSubTbpe::F_zeroStateFilter(const FloatVec& in,
    const FloatVec& F_denCoeff,
    FloatVec& out)
{
    /* zero state search filter */
    FloatVec F_state(F_nrCoeff);
    for (int i=0; i < F_nrCoeff; i++)
        F_state[i] = 0.0;

    for (i = 0; i < F_subframeLength; i++) {
        Float signal = in[i];
        for (Shortint m = F_nrCoeff-1; m > 0; m--) {
            signal -= F_denCoeff[m] * F_state[m];
            F_state[m] = F_state[m-1];
        }
        signal -= F_denCoeff[0] * F_state[0];
        F_state[0] = signal;
        out[i] = signal;
    }
}

void F_SpeSubTbpe::F_construct(const Shortint index,
    const Shortint grid,
    const Shortint matrix,
    FloatVec& vec)
{
    /* zero result vector */
    for (int i=0; i < F_subframeLength; i++)
        vec[i] = 0.0;

    for (Shortint j=0; j < F_nrTbpePulses; j++) {
        Float sum = 0.0;
        Shortint itemp = index;
        for (Shortint i=0; i < F_nrTbpePulses; i++) {
            if (itemp & 1)
                sum += F_tbpeTransfTable[(i + matrix*F_nrTbpePulses) *
                    F_nrTbpePulses + j];
            else

```

```

        sum -= F_tbpeTransfTable[(i + matrix*F_nrTbpePulses) *
                                F_nrTbpePulses + j];
        itemp >>= 1;
    }
    vec[grid + j * F_tbpeGridSpace] = sum;
}
}

Float F_SpeSubTbpe::F_search(const FloatVec& F_wMpeResidual,
                             const FloatVec& F_wCoeff,
                             FloatVec& F_tbpeInnovation,
                             Shortint& F_tbpeIndexCode,
                             Shortint& F_tbpeGridCode,
                             Shortint& F_tbpeMatrixCode)
{
    FloatVec F_filtered(F_subframeLength);

    /* calculate weighting filter impulse response */
    FloatVec F_ires(F_subframeLength);
    F_ires[0] = 1.0;
    for (int i=1; i<F_subframeLength; i++)
        F_ires[i] = 0.0;
    F_zeroStateFilter(F_ires, F_wCoeff, F_ires);

    /* compute correlation between impulse response and speech */
    FloatVec F_corrIS(F_subframeLength);
    F_crossCorr(F_wMpeResidual, F_ires, F_corrIS);

    /* test for all grids and all matrices */
    Float F_bestCorr = 0.0;
    Float F_bestPower = 1.0;

    F_tbpeIndexCode = 0;
    F_tbpeGridCode = 0;
    F_tbpeMatrixCode = 0;

    for (int F_matrix = 0; F_matrix < F_nrTbpeMatrices; F_matrix++)
        for (int F_grid = 0; F_grid < F_nrTbpeGrids; F_grid++) {

            /* calculate cross correlations */
            FloatVec F_cross(F_nrTbpePulses);
            F_crossCorrOfTransfMatrix(F_corrIS,
                                     F_grid,
                                     F_matrix,
                                     F_cross);

            /* approximate pulses with sign of cross correlation */
            Shortint F_index = 0;
            FloatVec F_signVector(F_nrTbpePulses);
            for (i=0; i<F_nrTbpePulses; i++)
                F_signVector[i] = -1.0;
            for (i = 0; i < F_nrTbpePulses; i++)
                if (F_cross[i] > 0) {
                    F_signVector[i] = 1;
                    F_index |= (1<<i);
                }
        }
}

```

```

/* construct filtered excitation vector */
F_construct(F_index, F_grid, F_matrix, F_tbpeInnovation);
F_zeroStateFilter(F_tbpeInnovation,
                  F_wCoeff, F_filtered);

/* compute power and correlations */
Float power = 0;
for (Shortint j = 0; j < F_subframeLength; j++)
    power += F_filtered[j] * F_filtered[j];
Float corr = 0;
for (j = 0; j < F_nrTbpePulses; j++)
    corr += F_cross[j] * F_signVector[j];

/* make decision */
if (corr*corr*F_bestPower > F_bestCorr*F_bestCorr* power) {
    F_bestCorr = corr;
    F_bestPower = power;
    F_tbpeIndexCode = F_index;
    F_tbpeGridCode = F_grid;
    F_tbpeMatrixCode = F_matrix;
}

F_construct(F_tbpeIndexCode, F_tbpeGridCode, F_tbpeMatrixCode,
            F_tbpeInnovation);
return F_bestCorr/F_bestPower;
}

Shortint F_SpeSubTbpe::F_quantize(const Float value)
{
    Shortint i = 0;
    if (value > F_tbpeGainLimitTable[F_nrTbpeCbGainLevel - 2])
        i = F_nrTbpeCbGainLevel - 1;
    else
        while (value > F_tbpeGainLimitTable[i])
            i++;
    return i;
}

```

F_SpeMain.hh

[illegible]

F_SpeSubMpe.hh

```

/*
 * class F_SpeSubMpe
 *
 * Multipulse innovation analysis
 *
 * COPYRIGHT (C) 1995 ERICSSON RADIO SYSTEMS AB
 *
 */
#ifndef F_SpeSubMpe_h
#define F_SpeSubMpe_h
#include "F_speDef.hh"

class F_SpeSubMpe {
public:
    F_SpeSubMpe();
    /* constructor */

    void main(
        const FloatVec& F_wCoeff,           /* in */
        const Float    F_excNormfactor,     /* in */
        const FloatVec& F_wLtpResidual,     /* in */
        FloatVec&       F_mpeInnovation,    /* out */
        Shortint&       F_mpePositionCode,  /* out */
        Shortint&       F_mpeAmpCode,       /* out */
        Shortint&       F_mpeSignCode,      /* out */
        Shortint&       F_mpeBlockMaxCode,  /* out */
        FloatVec&       F_wMpeResidual);    /* out */
    /* Main routine for module F_SpeSubMpe */

    Shortint F_maxMagIndex(
        const FloatVec& F_corrVec,           /* in */
        const ShortVec& F_posTaken);         /* in */
    /* Search for pulse position with max correlation so far */

    void F_solveNewAmps(
        const FloatVec& F_a,                 /* in */
        const FloatVec& F_c,                 /* in */
        const Shortint  F_nPulse,           /* in */
        FloatVec&       F_b);               /* out */
    /* Solve for optimal amplitudes (serves as a replacement */
    /* for cholesky) */

    void F_updateCrossCorr(
        const FloatVec& F_autoCorr,          /* in */
        const Shortint  F_pos,              /* in */
        const Float     F_gain,             /* in */
        FloatVec&       F_crossCorrUpd);    /* out */
    /* Update crosscorrelation vector */

    void F_calcImpResp(
        const FloatVec& F_wCoeff,           /* in */

```

```
FloatVec& F_impResp); /* out */
/* Calculate impulse response of IIR-filter */
/* coefficients wCoeff */

void F_autoCorrelate(
    const FloatVec& F_impResp, /* in */
    FloatVec& F_autoCorr); /* out */
/* Compute autocorrelation vector of impulse response */

void F_crossCorrelate(
    const FloatVec& F_impResp, /* in */
    const FloatVec& F_wLtpResidual, /* in */
    FloatVec& F_crossCorr); /* out */
/* Compute crosscorrelation between input speech */
/* and impulse response */

void F_searchInit(
    const FloatVec& F_crossCorr, /* in */
    const FloatVec& F_autoCorr, /* in */
    FloatVec& F_crossCorrUpd, /* out */
    ShortVec& F_mpePosVector, /* out */
    FloatVec& F_pulseAmp, /* out */
    ShortVec& F_postTaken); /* out */
/* Initialize search and search for first pulse */

void F_searchRest(
    const FloatVec& F_autoCorr, /* in */
    const FloatVec& F_crossCorr, /* in */
    FloatVec& F_crossCorrUpd, /* out */
    ShortVec& F_mpePosVector, /* out */
    FloatVec& F_pulseAmp, /* out */
    ShortVec& F_postTaken); /* out */
/* Search rest of pulses (optmethod 2) */

void F_openLoopQuantize(
    const Float& F_excEnergy, /* in */
    FloatVec& F_pulseAmp, /* out */
    ShortVec& F_mpeAmpVector, /* out */
    ShortVec& F_mpeSignVector, /* out */
    Shortint& F_mpeBlockMaxCode); /* out */
/* Calculate blockMax and openloop quantize blockmax */
/* and pulses */

void F_makeInnVector(
    const FloatVec& F_pulseAmp, /* in */
    const ShortVec& F_mpePosVector, /* in */
    FloatVec& F_mpeInnovation); /* out */
/* Make innovation vector */
```

```
void F_orderPositions(  
    ShortVec& F_mpePosVector,          /* in/out */  
    ShortVec& F_mpeAmpVector,          /* in/out */  
    ShortVec& F_mpeSignVector);        /* in/out */  
/* Order positions (optimum position encoding) */  
  
void F_makeCodeWords(  
    const ShortVec& F_mpePosVector,    /* in */  
    Shortint&      F_mpePositionCode, /* out */  
    const ShortVec& F_mpeAmpVector,    /* in */  
    Shortint&      F_mpeAmpCode,       /* out */  
    const ShortVec& F_mpeSignVector,    /* in */  
    Shortint&      F_mpeSignCode);     /* out */  
/* Construct codewords */  
  
void F_makeMpeResidual(  
    const FloatVec& F_mpeInnovation,    /* in */  
    const FloatVec& F_wCoeff,           /* in */  
    const FloatVec& F_wLtpResidual,     /* in */  
    FloatVec&      F_wMpeResidual);     /* out */  
/* Make new weighed residual with MPE contribution removed */  
};  
#endif
```

F_SpeSubTbpe.hh

```

/*
 * class F_SpeSubTbpe
 *
 * Transformed Binary Pulse Excited codebook
 *
 * COPYRIGHT (C) 1995 ERICSSON RADIO SYSTEMS AB
 */

#ifndef F_SpeSubTbpe_h
#define F_SpeSubTbpe_h

#include "F_speDef.hh"
#include "FloatVec.hh"

class F_SpeSubTbpe {
public:
    F_SpeSubTbpe();
    /* constructor */

    void F_SpeSubTbpe::main(
        const FloatVec& F_wMpeResidual,
        /* in, Weighted MPE residual = F_wLtpResidual with MPE */
        const FloatVec& F_wCoeff,
        /* in, weighted direct form coeff */
        const Float F_excNormFactor,
        /* in, Excitation normaliz. factor */
        FloatVec& F_tbpeInnovation,
        /* out, TBPE innovation, quantized gain included */
        Shortint& F_tbpeGainCode,
        /* out, TBPE gain code */
        Shortint& F_tbpeIndexCode,
        /* out, TBPE pulse sign code */
        Shortint& F_tbpeGridCode,
        /* out, TBPE grid code */
        Shortint& F_tbpeMatrixCode);
        /* out, TBPE transform matrix code */
    /* Main routin for TBPE codebook search */

    void F_crossCorr(
        const FloatVec& v1,          /* in, Target vector 1 */
        const FloatVec& v2,          /* in, Target vector 2 */
        FloatVec& F_corr);          /* out, Cross correlated vector */
    /* Calculate cross correlation */

    void F_crossCorrOfTransfMatrix(
        const FloatVec& v1,          /* in, Target vector */
        const Shortint grid,        /* in, The grid number */
        const Shortint matrix,      /* in, The matrix number */
        FloatVec& F_crossCorr);     /* out, Cross correlated */
        /* vector */

```

```
/* Calculate cross correlation for the transformation matrix */

void F_zeroStateFilter(
    const FloatVec& in,          /* in, Vector to be filtered */
    const FloatVec& F_denCoeff, /* in, Direct form coefficient */
    FloatVec& out);             /* out, Filtered vector */
/* Zero state filter with coefficients F_denCoeff */

void F_construct(
    const Shortint index,      /* in, Index code */
    const Shortint grid,      /* in, Grid code */
    const Shortint matrix,    /* in, Matrix code */
    FloatVec& vec);           /* out, Constructed excitation */
/* Construct a excitation vector */

Float F_search(
    const FloatVec& F_wMpeResidual,
        /* in, Weighted MPE residual= F_wLtpResidual with MPE */
        /* innovation removed */
    const FloatVec& F_wCoeff,
        /* in, Weighted direct form coeffs */
    FloatVec& F_tbpeInnovation,
        /* out, TBPE innovation, quantized gain included */
    Shortint& F_tbpeIndexCode,
        /* out, TBPE pulse sign code */
    Shortint& F_tbpeGridCode,
        /* out, TBPE grid code */
    Shortint& F_tbpeMatrixCode);
        /* out, TBPE transform matrix code */
/* search for best index,
 * approximate index with sign of correlation,
 * examine all grids and matrices
 * return optimal innovation, gainCode, index, grid, matrix
 */

Shortint F_quantize(
    const Float value);        /* in, value to be quantized */
/* Quantize TBPE gain */
};
#endif
```

REFERENCES

- [1] P. Kroon, E. Deprettere
A class of analysis-by-synthesis predictive coders for high quality speech coding at rates between 4.6 and 16 kbit/s.
IEEE Jour. Sel. Areas Com., Vol. SAC-6, No. 2, Feb. 1988.
- [2] H. Chen, W.C. Wong, C.C. Ko
Low-delay hybrid vector excitation linear predictive speech coding
Electronics letters Vol. 29 no. 25 1993
- [3] D. Lin
Code-excited linear prediction using a mixed source model
Proc. ASSP DSP workshop, 1986
- [4] D. Lin
Ultra-fast CELP coding using deterministic multi-codebook innovations.
IEEE ICASSP-92, San Francisco, 1992.
- [5] N. Moreau, P. Dymarski
Mixed excitation celp coder.
Eurospeech-89, Paris, Sep. 1989.
- [6] K. Ozawa
A hybrid speech coding based on multi-pulse and CELP at 3.2 kb/s.
IEEE ICASSP-90, Albuquerque, 1990.
- [7] R. Zinser, S. Koch
4800 and 7200 bit/sec hybrid codebook multipulse coding.
IEEE ICASSP-89, Glasgow, 1989
- [8] R. Zinser
Hybrid switched multi-pulse/stochastic speech coding technique.
US Patent # 5060269
- [9] B. Atal, J. Remde
A new model of LPC excitation for producing natural-sounding speech at low bit rates.
IEEE ICASSP-82, Paris, 1982.
- [10] P. Vary, K. Hellwig, R. Hofmann
A regular-pulse excited linear predictive codec.
Speech Communication 7, North-Holland, 1988.
- [11] R.A. Salami
Binary code excited linear prediction (BCELP): New approach to celp coding of speech without codebooks.
Electronics letters, vol. 25 no. 6 march 1989.
- [12] R. Salami
Binary pulse excitation: A novel approach to low com-

- plexity CELP coding.
Kluwer Academic Pub., Advances in speech coding, 1991.
- [13] I. Gerson, M. Jasiuk
Vector sum excited linear prediction (VSELP).
Kluwer Academic Pub., Advances in speech coding, 1991.
- [14] R. Cox, W.B. Kleijn, P. Kroon
Robust celp coders for noisy backgrounds and noisy channels.
IEEE ICASSP-89, Glasgow, 1989.
- [15] N. Cox
Error control and index assignment for speech codecs.
Kluwer Academic Press, 1993.
- [16] T.B.Minde
Excitation pulse positioning method in a linear predictive speech coder.
US Patent # 5193140



CLAIMS

1. An analysis-by-synthesis linear predictive speech coder, characterized by a synthesis part comprising:

means (34) for generating a multi-pulse excitation (MPE);

5 means (36) for generating a transformed binary pulse excitation (TBPE); and

means (38) for combining said multi-pulse excitation and said transformed binary pulse excitation.

10 2. The speech coder of claim 1, characterized by said multi-pulse excitation (MPE) generating means (34) comprising means for generating pulses in restricted pulse positions.

3. The speech coder of claim 2, characterized by said multi-pulse excitation (MPE) generating means (34) comprising means for phase position coding.

15 4. The speech coder of claim 1, 2 or 3, characterized by said synthesis part further comprising an adaptive code book (14) for generating an adaptive excitation.

20 5. The speech coder of claim 4, characterized by means (18, 38; g_I , g_M , g_T) for combining said multi-pulse (MPE), transformed binary pulse (TBPE) and adaptive excitations.

This Sheet Has Been Replaced

