

Department of Electronic and Information Engineering
電子及資訊工程學系

The Hong Kong Polytechnic University

Center for Multimedia Signal Processing

CELP Encoders

Dr. M. W. Mak

enmwak@polyu.edu.hk

Tel: 27666257

Fax: 23628439

URL: www.en.polyu.edu.hk/~mwmak/mypage.htm

February 17, 2001

Summary

This document describes the C implementation of the Code-Excited Linear Prediction (CELP 3.2a) encoder available from ftp://svr-ftp.eng.cam.ac.uk/comp.speech/coding/celp_3.2a.tar.gz.

CONTENTS

<i>CONTENTS</i>	2
<i>1. FS-1016 Coding Characteristics</i>	3
<i>2. FS-1016 Bit Assignment</i>	3
<i>3. CELP Encoding Algorithm</i>	4
3.1 HighPassFilter [zerofilt.c and polefilt.c]	6
3.2 ComputeLPC [autohf.c]	8
3.3 LpcToLsp [pctolsp4.c]	12
3.4 QuantizeLsp [lsp34.c]	15
3.5 InterpolateLsp [intanaly.c]	17
3.6 MakeSubFrame()	18
3.7 LspToLpc [lsptopc.c]	19
3.8 CodebookSearch [csub.c]	20
3.9 ComputeInitialState [config.c]	21
3.10 ComputeImpulseResponse [impulse.c]	22
3.11 PrepareModifyAdpCbkgain [mexcite1() of mexcite.c]	23
3.12 PitchGain [pgain.c]	25
3.13 PitchSearch [psearch.c]	29
3.14 FracDelay [delay.c]	34
3.15 LongFracDelay [ldelay.c]	37
3.16 PitchSynthesis [pitchvq.c]	39
3.17 StochasticCbkgain [cbsearch.c]	41
3.18 CodebookGain [cgain.c]	43
3.19 PackTau [packtau.c]	46
3.20 Pack [pack.c]	47
<i>Appendix A: Code Book Search Methods</i>	48
<i>Appendix B: Zero Input Response</i>	51

1. FS-1016 Coding Characteristics

	Spectrum	Pitch	Code Book
	-----	-----	-----
Update	30 ms l1=240	30/4 = 7.5 ms lp=60	30/4 = 7.5 ms l=60
Order	10	256 (max) x 60 1 gain	512 (max) x 60 1 gain
Analysis	Open loop Correlation 30 ms Hamming no preemphasis 15 Hz BW exp	Closed loop Modified MSPE VQ, weight=0.8 range 20 to 147 (w/ fractions)	Closed loop MSPE VQ weight=0.8 shift by 2 samples 77% sparsely
Bits per Frame	34 indep LSP [3444433333]	index: 8+6+8+6 gain(-1,2): 5*4	index: 9*4 gain(+/-): 5*4
Bit Rate	1133.3 bps	1600 bps	1866.67 bps

NOTE: The remaining 200 bps are used as follows: 1 bit per frame for synchronization, 4 bits per frame for forward error correction and 1 bit per frame to provide future expansion(s) of the coder.

2. FS-1016 Bit Assignment

	Bit	Bit		Bit
	-----	-----		-----
lsp 1	1-3	lsp 6	20-22	
lsp 2	4-7	lsp 7	23-25	
lsp 3	8-11	lsp 8	26-28	
lsp 4	12-15	lsp 9	29-31	
lsp 5	16-19	lsp 10	32-34	
Subframe:	1	2	3	4
	-----	-----	-----	-----
pitch delay	35-42	87-94
delta delay	62-67	114-119
pitch gain	43-47	68-72	95-99	120-124
cbindex	48-56	73-81	100-108	125-133
cbgain	57-61	82-86	109-113	134-138
future bit	139			
error control	140-143			
sync	144			

The sync bit (144) begins with 0 in the first frame, then alternates between 1 and 0 on successive frames.

3. CELP Encoding Algorithm

```

int    nLsp=10;           // No. of LSP coefficients per frame
int    nSubFrms=4;       // No. of subframes per analysis frame
int    pOrder=10;        // Prediction order (p)
float  lpc[pOrder+1];    // LP coeffs. lpc[0] must be 1.0
float  lsp[pOrder];      // LSP coefficients
float  lspSubframe[nSubFrms][pOrder]; // Interpolated LSP for each subframe
int    lspIndex[nLsp];   // Index to LSP coding table
int    lspTbl[nLsp][16]; // LSP quantization table
int    nBitsPitchDelay=8; // 8 bits for pitch delay (m)
int    nBitsPitchDDelay=6; // 6 bits for delta delay (f)
int    nBitsPitchGain=5; // 5 bits for pitch gain
int    lspAlloc[10]={3, 4, 4, 4, 4, 3, 3, 3, 3, 3}; // Bit allocation for LSP coefficients
int    cbindex;         // Stochastic codebook index for each subframe
float  gindex;         // Stochastic codebook gain index for each subframe
int    nBitsCbIndex=9; // 9 bits for codebook index
int    nBitsCbGain=5;  // 5 bits for codebook gain
int    x[1083];        // Stochastic codebook
int    frmSize=240;    // LP analysis frame size
int    subFrmSize=60; // Sub-frame size (l)
float  sNew[frmSize]; // New speech frame
float  sOld[frmSize]; // Old speech frame
float  sSub[frmSize]; // Frame for CELP closed-loop analysis
short  stream[144];   // Encoded bit stream
int    tauptr;        // Pitch delay pointer
int    minptr;        // Minimum pitch delay pointer
int    pindex;        // Pitch gain index. True floating value stored in bb[2]
float  bb[4];         // Define the pitch syn. filter. bb[0]=pitch delay
// bb[2]=pitch gain, bb[1]=bb[3]=0
float  h[60];         // Impulse response of the perceptually weighted LP filter
float  e0[60];        // The target excitation signal (see Eqn. 18)

```

CelpEncode()

```

{
    int pointer;           // Points to current encoding bit position in stream[]
    float v[60];          // Excitation vector

    Open(pcmBitStream); // Open PCM bitstream for encoding

    // Read 480 bytes (240 samples) from pcmBitStream and convert to float
    while ((sNew=Read(pcmBitStream, 480))!=eof) {

        // High pass filter the current frame, return results in sNew[0...239]
        HighPassFilter(sNew, frmSize);

        // Compute the LP coefficients
        ComputeLPC(sNew, frmSize, pOrder, lpc);

        // Convert to LSP coefficients
        LpcToLsp(lpc, pOrder, lsp);

        // Quantize the LSP parameters
        QuantizeLsp(lsp, pOrder, lspAlloc, lspIndex);
    }
}

```

```

// For each LSP coefficient, pack LSP index to stream[]
for (i=0; i<nLsp; i++) {
    Pack(lspAlloc[i], lspIndex[i], stream, &pointer);
}

// Interpolate the LSP coefficients based on the previous and current LSP coeffs.
InterpolateLsp(lsp, lspSubframe);

// Make sSub[] for CELP close-loop analysis
MakeSubFrames(sNew, sOld, sSub);

// For each subframe, search the stochastic and adaptive code books
k = 0;
for (i=0; i<nSubFrms; i++) {

    // Convert interpolated subframe LSP to LP coefficients
    LspToLpc(lspSubframe[i], lpc);

    // Stochastic codebook and adaptive codebook (pitch) search. Update tauptr, pindex,
    // minptr, bb[], cbindex, and gindex
    CodebookSearch(&sSub[k],&v[k]);

    // Pack parameter indices in bit stream array
    if (((i+1) % 2) != 0)
        // Pack odd subframe. Note that minptr is set to 0 in psearch.c for odd subframe
        PackTau(tauptr-minptr, nBitsPitchDelay, pdencode, stream, &pointer);
    else
        // Pack even subframe. Lag relative to the previous odd subframe
        Pack(tauptr-minptr, nBitsPitchDDelay, stream, &pointer);

    // Pack pitch lag index into bit stream array
    Pack(pindex, nBitsPitchGain, stream, &pointer);

    // Pack stochastic codebook gain and index into bit stream array.
    // Note that cbindex={1...512}. So it needs to be decrement before packing
    cbindex--;
    pack(cbindex, nBitsCbkJndex, stream, &pointer);
    pack(gindex, nBitsCbkJgain, stream, &pointer);
    k = k + subFrmSize;           // Index to the next subframe in sSub[]
}
// Pack sync bit
sync = sync ^ 1;                // Alternate sync bit
pointer = 143;                  // Access sync bit location
pack(sync, 1, stream, &pointer);

// shift speech buffer for processing the next frame
for (i = 0; i < frmSize; i++)
    sOld[i] = sNew[i];
}
}

```

3.1 HighPassFilter [zerofilt.c and polefilt.c]

This function filters the current frame based on a pole-zero high-pass filter of the form

$$H(z) = H_{zero}(z)H_{pole}(z)$$

where $H_{zero}(z) = 0.946 - 1.892z^{-1} + 0.946z^{-2}$ and $H_{pole}(z) = \frac{1}{1 - 1.889033z^{-1} + 0.8948743z^{-2}}$.

The frequency and phase responses of the filter is shown in Fig. 1 below. The high-pass filter is used to prevent undesired low-frequency components (e.g. 50 Hz hum noise) from entering the encoder.

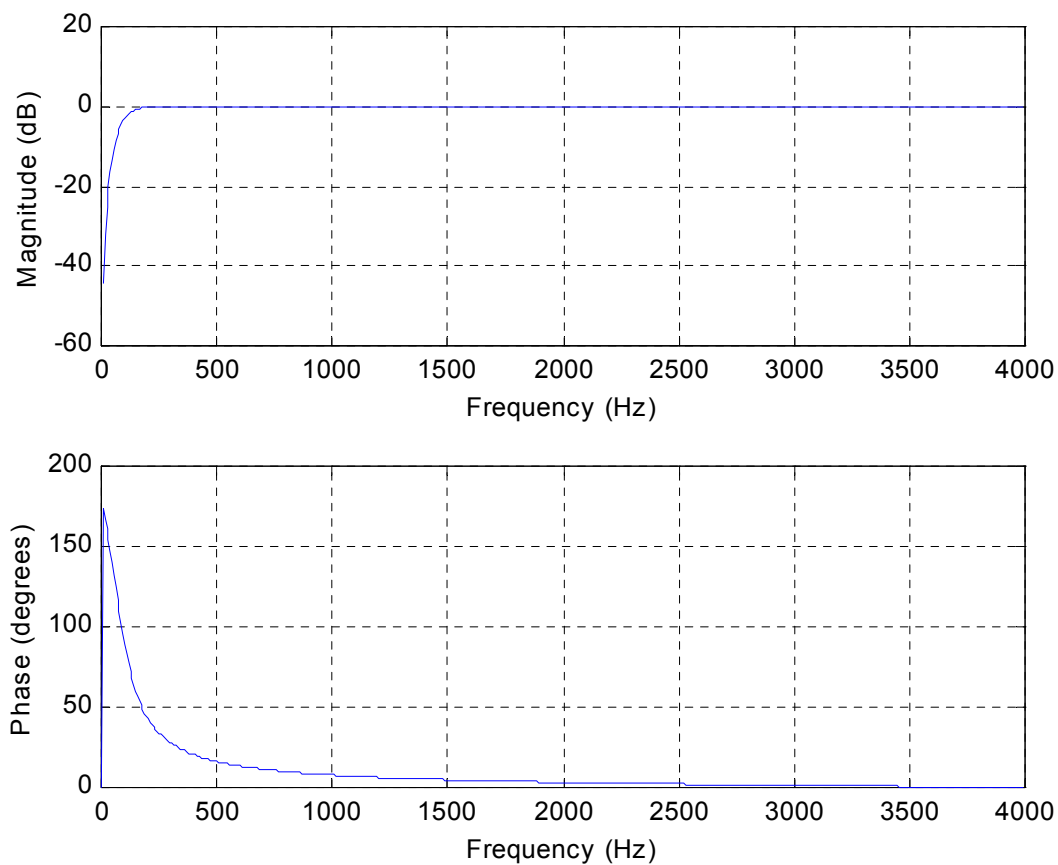


Fig. 1 Frequency and phase response of the high pass filter $H(z)$

```
HighPassFilter(float speech[], int len)
```

```
{
    float za[3] = {0.0,0.0,0.0};
    float a[3] = {1.0, -1.889033, 0.8948743};
    float zb[3] = {0.0,0.0,0.0};
    float b[3] = {0.946,-1.892,0.946};

    zerofilt(b,2,zb,speech, len);
    polefilt(a,2,za,speech, len);
}
```

// zerofilt: Implement the FIR filter $\left(\sum_{k=0}^p b_k z^{-k}\right)$, i.e. $y[n] = \sum_{k=0}^p b_k x[n-k]$

*// b[0...p] are the filter coefficients, p is the filter order, z[0...p] represents the filter memory,
// xy[0...N-1] contains input/output, and N is the data length.*

```
zerofilt(float b[], int p, float z[], float xy[], int N)
```

```
{
    int t, j;
    float ar;
    for (t = 0; t < N; t++) {
        z[0] = xy[t];
        for (ar = 0.0; j = p; j > 0; j--) {
            ar += z[j] * b[j]; // ar = b1x[t-1] + b2x[t-2] for p=2
            z[j] = z[j-1];
        }
        xy[t] = ar + z[0] * b[0]; // y[t] = b0x[t] + b1x[t-1] + b2x[t-2] for p=2
    }
}
```

// polefilt: Implement the IIR filter $\left(1 + \sum_{k=1}^p a_k z^{-k}\right)^{-1}$, i.e. $y[n] = x[n] - \sum_{k=1}^p a_k y[n-k]$

*// a[0...p] are the filter coefficients, p is the filter order, z[0...p] represents the filter memory,
// xy[0...N-1] contains input/output, and N is the data length. Note: a[0] must be equal to 1.0.*

```
polefilt(float a[], int p, float z[], float xy[], int N)
```

```
{
    int t, j;
    for (t = 0; t < N; t++) {
        z[0] = xy[t];
        for (j = p; j > 0; j--) {
            z[0] = z[0] - a[j] * z[j]; // -a1y[t-1] - a2y[t-2] for p=2
            z[j] = z[j-1]; // shift filter memory
        }
        xy[t] = z[0]; // y[t] = x[t] - a1y[t-1] - a2y[t-2] for p=2
    }
}
```

3.2 ComputeLPC [autohf.c]

This function computes the bandwidth expanded LP coefficients of the current frame using the Durbin's recursive procedure. The inverse filter should be of the form

$$A(z) = 1 + \sum_{k=1}^p a_k z^{-k}$$

where $\{a_k\}_{k=1}^p$ are the predictor coefficients and p is the prediction order. The poles' bandwidth is expanded by multiplying each LP coefficients by a factor, as follows:

$$a'_k = a_k \alpha^k$$

where a'_k are the LP coefficients with bandwidth expanded poles. In FS1016, $\alpha = 0.994127$.

The Durbin's recursive procedure can be specified as follows:

$$E^{(0)} = R(0) \quad (1a)$$

$$k_i = - \left\{ R(i) + \sum_{j=1}^{i-1} \alpha_j^{(i-1)} R(|i-j|) \right\} / E^{(i-1)}, \quad 1 \leq i \leq p \quad (1b)$$

$$\alpha_i^{(i)} = k_i \quad 1 \leq j \leq i-1 \quad (1c)$$

$$\alpha_j^{(i)} = \alpha_j^{(i-1)} + k_i \alpha_{i-j}^{(i-1)} \quad 1 \leq j \leq i-1 \quad (1c)$$

$$E^{(i)} = (1 - k_i^2) E^{(i-1)} \quad (1d)$$

$$a_j = \alpha_j^{(p)} \quad (1e)$$

where $a_j, j = 1, \dots, p$, are the prediction coefficients.

To test the stability of the bandwidth expanded LP filter, we convert the bandwidth expanded LP coefficients $\{a'_j\}_{j=1}^p$ to reflection coefficients $\{k_i\}_{i=1}^p$, as follows:

$$k_i = \alpha_i^{(i)} \quad (2a)$$

$$\alpha_j^{(i-1)} = \frac{\alpha_j^{(i)} - \alpha_i^{(i)} \alpha_{i-j}^{(i)}}{1 - k_i^2} \quad 1 \leq j \leq i-1 \quad (2b)$$

where the index i takes values $p, p-1, \dots, 1$ in that order, and initially $\alpha_j^{(p)} = a'_j, 1 \leq j \leq p$. The LP filter is stable if $|k_i| < 1, i = 1, \dots, p$.

ComputeLPC(float speech[], int frmSize, int pOrder, float lpc[])

```
{
    float c0; // Correlation coeff c0
    float c[10]; // Correlation c1,...,c10 stored in c[0...9]
    float aTemp[11]; // Temporary storage for LP coeffs
    float s[240]; // Hamming windowed speech
    float rc[10]; // Reflection coeffs.
    int i,j;

    Hamming(speech, frmSize, s); // Apply Hamming window to speech
    ComputeCorrelation(s, frmSize, pOrder, &c0, c);
    aTemp[0] = 1.0; // a0 = 1.0
    DurbinRecursion(c0, c, &aTemp[1], pOrder); // Compute LP using Durbin's Recursion
    BandwidthExpansion(aTemp, lpc, pOrder); // Output bandwidth expanded LP in lpc[0...10]
    LpcToRc(lpc, rc, pOrder);
}
```

```

    for (i=0; i<pOrder; i++) {
        if (fabs(rc[i]) > 1.0) {
            for (j=0; j<pOrder; j++)
                lpc[i+1] = 0.0;
            break;
        }
    }
}

// Output Hamming windowed speech in s[]
Hamming(float speech[], int frmSize, float s[])
{
    int i;
    float h240[240]; // Hamming window factor for window width = 240
    float h60[60]; // Hamming window factor for window width = 60

    if (frmSize == 240) {
        for (i=0; i<240; i++)
            s[i] = h240[i]*speech[i];
    }
    else
        for (i=0; i<60; i++)
            s[i] = h60[i]*speech[i];
}

h[240] = {
0.08000000, 0.08015895, 0.08063570, 0.08142991, 0.08254104, 0.08396832, 0.08571076, 0.08776715,
0.09013609, 0.09281592, 0.09580480, 0.09910066, 0.10270123, 0.10660401, 0.11080631, 0.11530523,
0.12009765, 0.12518026, 0.13054956, 0.13620182, 0.14213315, 0.14833944, 0.15481641, 0.16155958,
0.16856429, 0.17582569, 0.18333877, 0.19109835, 0.19909904, 0.20733533, 0.21580153, 0.22449178,
0.23340007, 0.24252026, 0.25184603, 0.26137094, 0.27108842, 0.28099174, 0.29107405, 0.30132840,
0.31174770, 0.32232474, 0.33305221, 0.34392271, 0.35492871, 0.36606262, 0.37731673, 0.38868327,
0.40015439, 0.41172216, 0.42337857, 0.43511559, 0.44692509, 0.45879891, 0.47072885, 0.48270666,
0.49472407, 0.50677277, 0.51884443, 0.53093072, 0.54302327, 0.55511373, 0.56719375, 0.57925497,
0.59128906, 0.60328771, 0.61524262, 0.62714553, 0.63898821, 0.65076249, 0.66246021, 0.67407331,
0.68559374, 0.69701356, 0.70832487, 0.71951984, 0.73059075, 0.74152995, 0.75232987, 0.76298304,
0.77348212, 0.78381983, 0.79398905, 0.80398273, 0.81379397, 0.82341600, 0.83284216, 0.84206593,
0.85108095, 0.85988098, 0.86845994, 0.87681191, 0.88493110, 0.89281192, 0.90044891, 0.90783679,
0.91497046, 0.92184499, 0.92845563, 0.93479781, 0.94086714, 0.94665944, 0.95217070, 0.95739710,
0.96233505, 0.96698112, 0.97133210, 0.97538499, 0.97913699, 0.98258550, 0.98572815, 0.98856275,
0.99108735, 0.99330021, 0.99519979, 0.99678478, 0.99805410, 0.99900685, 0.99964238, 0.99996026,
0.99996026, 0.99964238, 0.99900685, 0.99805410, 0.99678478, 0.99519979, 0.99330021, 0.99108735,
0.98856275, 0.98572815, 0.98258550, 0.97913699, 0.97538499, 0.97133210, 0.96698112, 0.96233505,
0.95739710, 0.95217070, 0.94665944, 0.94086714, 0.93479781, 0.92845563, 0.92184499, 0.91497046,
0.90783679, 0.90044891, 0.89281192, 0.88493110, 0.87681191, 0.86845994, 0.85988098, 0.85108095,
0.84206593, 0.83284216, 0.82341600, 0.81379397, 0.80398273, 0.79398905, 0.78381983, 0.77348212,
0.76298304, 0.75232987, 0.74152995, 0.73059075, 0.71951984, 0.70832487, 0.69701356, 0.68559374,
0.67407331, 0.66246021, 0.65076249, 0.63898821, 0.62714553, 0.61524262, 0.60328771, 0.59128906,
0.57925497, 0.56719375, 0.55511373, 0.54302327, 0.53093072, 0.51884443, 0.50677277, 0.49472407,
0.48270666, 0.47072885, 0.45879891, 0.44692509, 0.43511559, 0.42337857, 0.41172216, 0.40015439,
0.38868327, 0.37731673, 0.36606262, 0.35492871, 0.34392271, 0.33305221, 0.32232474, 0.31174770,
0.30132840, 0.29107405, 0.28099174, 0.27108842, 0.26137094, 0.25184603, 0.24252026, 0.23340007,
0.22449178, 0.21580153, 0.20733533, 0.19909904, 0.19109835, 0.18333877, 0.17582569, 0.16856429,
0.16155958, 0.15481641, 0.14833944, 0.14213315, 0.13620182, 0.13054956, 0.12518026, 0.12009765,
0.11530523, 0.11080631, 0.10660401, 0.10270123, 0.09910066, 0.09580480, 0.09281592, 0.09013609,
0.08776715, 0.08571076, 0.08396832, 0.08254104, 0.08142991, 0.08063570, 0.08015895, 0.08000000
}

h[60] = {

```

```

0.08000000, 0.08260599, 0.09039444, 0.10327710, 0.12110800, 0.14368512, 0.17075265, 0.20200389,
0.23708477, 0.27559779, 0.31710660, 0.36114088, 0.40720170, 0.45476719, 0.50329839, 0.55224544,
0.60105374, 0.64917028, 0.69604988, 0.74116137, 0.78399362, 0.82406132, 0.86091050, 0.89412363,
0.92332440, 0.94818195, 0.96841464, 0.98379322, 0.99414344, 0.99934804, 0.99934804, 0.99414344,
0.98379322, 0.96841464, 0.94818195, 0.92332440, 0.89412363, 0.86091050, 0.82406132, 0.78399362,
0.74116137, 0.69604988, 0.64917028, 0.60105374, 0.55224544, 0.50329839, 0.45476719, 0.40720170,
0.36114088, 0.31710660, 0.27559779, 0.23708477, 0.20200389, 0.17075265, 0.14368512, 0.12110800,
0.10327710, 0.09039444, 0.08260599, 0.08000000
}

```

// ComputeCorrelation() implements the autocorrelation function $r[m] = \sum_{n=0}^{N-1-m} s[n]s[n+m]$, Rabiner (1993), p. 114.

// After this function, $c_0 = r[0]$ and $c[0...pOrder-1] = r[1...pOrder]$

```

ComputeCorrelation(float s[], int frmSize, int pOrder, float *c0, float c[])
{

```

```

    int m,n;

```

```

    for (*c0=0.0, n=0; n<frmSize; n++)           // Calculate c0
        *c0 += s[n]*s[n];

```

```

    for (m=1; m<=pOrder; m++) {                 // Calculate c[0...pOrder-1]
        c[m-1] = 0.0;
        for (n=0; n<N-m; n++)
            c[m-1] += s[n]*s[n+m];
    }
}

```

// Code added by M.W. Mak to implement the Durbin's recursive procedure, Eqn. 1

```

#define DIV_THRES 0.0000001

```

```

#define signof(x) (x>0 ? 1 : -1)

```

```

DurbinRecursion(c0, c, lpc, pOrder)           // Output LP coeffs in lpc[0...pOrder-1]
int pOrder;
float c0, *c, *lpc;                          // c0 = R[0], c[0...pOrder-1] = R[1...pOrder]
{

```

```

    int    i,j;
    float  sum;
    float  K[11];                             // reflection coefficient
    float  E[11];
    float  a[11][11];                         // variable alpha in Eqn. 1. a[0][1] and a[1][0] not used
    float  R[11];

```

// Initialize intermediate variables

```

for (i=0; i<=pOrder; i++) {
    K[i] = E[i] = R[i] = 0.0;
    for (j=0; j<=pOrder; j++)
        a[i][j] = 0.0;
}

```

// Calculate LPC parameters

```

R[0] = c0;
for (i=1; i<=pOrder; i++) R[i] = c[i-1];
E[0] = R[0];                                     // Eqn. 1a
if (fabs(E[0]) < DIV_THRES)
    E[0] = signof(E[0]) * DIV_THRES;

```

```

K[1] = -R[1] / E[0]; // Eqn. 1b
a[1][1]=K[1]; // Eqn. 1c
E[1]=(1-K[1]*K[1]) * E[0]; // Eqn. 1d

for (i=2;i<=pOrder;i++) {
    sum=0.0;
    for (j=1;j<i;j++)
        sum += a[j][i-1] * R[i-j]; // Eqn. 1b
    K[i] = -(R[i]+sum)/E[i-1]; // Eqn. 1b
    a[i][i] = K[i]; // Eqn. 1c
    for (j=1;j<i;j++)
        a[j][i] = a[j][i-1] + K[i] * a[i-j][i-1]; // Eqn. 1c
    E[i]=(1-K[i]*K[i]) * E[i-1]; // Eqn. 1d
    if (fabs(E[i]) < DIV_THRES)
        E[i] = signof(E[i]) * DIV_THRES;
}
for (j=1;j<=pOrder;j++) // Eqn. 1e
    lpc[j-1] = a[j][pOrder];
}
#undef DIV_THRES 0.0000001
#undef signof(x)

BandwidthExpansion(float a[], lpc[], pOrder)
{
    float bwFactor[11] = {
        1.00000000, 0.99412700, 0.98828849, 0.98248427, 0.97671414, 0.97097790, 0.96527535,
        0.95960629, 0.95397052, 0.94836785, 0.94279809
    };

    int i;
    for (i = 0; i <= pOrder; i++)
        lpc[i] = a[i]*bwFactor[i];
}

// Output reflection coefficients in rc[0...pOrder-1]
LpcToRc(lpc, rc, p)
int p; // Prediction order, must be 10
float *lpc; // LP coefficients a[0...p]
float *rc; // Reflection coeffs rc[0...p-1]
{
    int i,j;
    float alpha[11][11];
    float *k = rc - 1; // Index of k starts from 1, as in Makhoul's paper or Eqn. 2

    for (i=0; i<=p; i++) // Initialize intermediate variable
        for (j=0; j<=p; j++)
            alpha[i][j]=0.0;
    for (j=1; j<=p; j++)
        alpha[j][p] = lpc[j];
    for (i=p; i>=1; i--) {
        k[i] = alpha[i][i]; // Eqn. 2a
        for (j=1; j<i; j++)
            alpha[j][i-1]=(alpha[j][i]-alpha[i][i]*alpha[i-j][i])/(1-k[i]*k[i]); // Eqn. 2b
    }
}

```

3.3 LpcToLsp [pctolsp4.c]

This function converts LP coefficients to LSP coefficients for quantization and interpolation purposes (see *QuantizeLsp()* below). The function implements the following equations:

$$\begin{aligned} A_0 &= 1 \\ B_0 &= 1 \\ A_k &= (a_k - a_{11-k}) + A_{k-1} \\ B_k &= (a_k - a_{11-k}) - B_{k-1} \end{aligned}$$

where $k = 1, \dots, 10$ and A_k and B_k are the coefficients of $P(z)$ and $Q(z)$ respectively, i.e.,

$$\begin{aligned} P(z) &= A_0 z^{10} + A_1 z^9 + \dots + A_{10} \\ Q(z) &= B_0 z^{10} + B_1 z^9 + \dots + B_{10} \end{aligned}$$

Since the coefficients of $P(z)$ and $Q(z)$ are symmetrical (i.e. $A_k = A_{10-k}$ and $B_k = B_{10-k}$), they can be reduced to polynomial of order 5, as follows:

$$\begin{aligned} P(z) &= z^5 [A_0(z^5 + z^{-5}) + A_1(z^4 + z^{-4}) + \dots + A_5] \\ Q(z) &= z^5 [B_0(z^5 + z^{-5}) + B_1(z^4 + z^{-4}) + \dots + B_5] \end{aligned} \quad (3)$$

As all roots are on the unit circle, we only need to evaluate $P(z)$ and $Q(z)$ on the unit circle. Let $z = e^{j\omega}$, then we have $z^1 + z^{-1} = 2 \cos(\omega)$ and

$$\begin{aligned} P(\omega) &= 2e^{j5\omega} [A_0 \cos(5\omega) + A_1 \cos(4\omega) + \dots + \frac{1}{2} A_5] \\ Q(\omega) &= 2e^{j5\omega} [B_0 \cos(5\omega) + B_1 \cos(4\omega) + \dots + \frac{1}{2} B_5] \end{aligned} \quad (4)$$

The LSFs are the roots of $P(\omega)$ and $Q(\omega)$. Note that the roots of $P(\omega)$ and $Q(\omega)$ interlace with each other on the unit circle, i.e., the following is always satisfied:

$$0 \leq \omega_{q,0} \leq \omega_{p,0} \leq \omega_{q,1} \leq \omega_{p,1} \leq \dots \leq \omega_{q,9} \leq \omega_{p,9} \leq \pi. \quad (5)$$

This characteristic enables us to find the roots efficiently using a method called bisection search. Specifically, the upper half of the unit circle is divided into N ($N = 128$) intervals. Then, we find the first interval $[\pi i/N, \pi(i+1)/N; i = 0, \dots, N-1]$ where a zero-crossing of $Q(\omega)$ exists. If there is a zero-crossing of $Q(\omega)$ in the interval $[\pi i'/N, \pi(i'+1)/N]$, we use bisection search to look for the zero-crossing (root) in this interval. Once the first root of $Q(\omega)$ is found, we start to look for the first root of $P(\omega)$. We do this by determining whether there is a root of $P(\omega)$ between the interval $[\omega_{q,0}, \pi(i'+1)/N]$. If a zero crossing is found in the interval $[\pi i''/N, \pi(i''+1)/N]$ or $[\omega_{q,0}, \pi(i'+1)/N]$, we look for the root in this interval using bisection search. This process is repeated until all roots are found.

```
LpcToLsp(float a[], // LP coefficients a[0...p] , a[0] must be 1.0
        int p, // Prediction order
        float freq[], // LSP coefficients freq[0...p-1]
        int *lspflag) // == TRUE if there is problem.
{
    static float lastfreq[24]; // Storing the LSP of previous pass. Note: array of 10 floats is enough
    float A[11], B[11];
    int k, i;
    int evenRoot; // ==1 when k=0,2,4,6,8; ==0 otherwise
    float oldOmega, newOmega, oldQ, newQ; // New and old freq in radian

    // Initialize A_k and B_k
    A[0] = B[0] = 1.0;
```

```

for (k=1; k<=p; k++) {
    A[k] = a[k]-a[p+1-k]+A[k-1];
    B[k] = a[k]+a[p+1-k]-B[k-1];
}

// We start from omega=0 and look for sign change in funcPQ(B,omega) until omega=pi. Whenever a
// sign change occurs, we use bisection search to look for the root that falls into the current interval of omega.
// As  $\omega_{q0} < \omega_{p0} < \omega_{q1} < \omega_{p1} < \dots < \pi$ , we start from looking for the smallest root of  $Q(\omega)$ , then we
// look for the first root of  $P(\omega)$ , then the next root of  $Q(\omega)$ , and then the next root of  $P(\omega)$ . This alternating
// process continues until the last root of  $P(\omega)$  is found. Note that the roots of  $Q(\omega)$  will be stored in freq[0],
// freq[2],...,freq[8] and the roots of  $P(\omega)$  will be stored in freq[1], freq[3],...,freq[9].
k = 0;
i = 1;
oldOmega = 0.0;
oldQ = funcPQ(B, oldOmega); // Begin by looking for the first root of  $Q(\omega)$ 
evenRoot = TRUE;
while (k<p && i<=N) {
    newOmega = PI*(float)i/(float)N;
    if (evenRoot)
        newQ = funcPQ(B,newOmega); // k=0,2,4,6,8; Eqn. 4
    else
        newQ = funcPQ(A,newOmega); // k=1,3,5,7,9; Eqn. 4
    if ((oldQ * newQ) > 0.0) {
        oldQ = newQ; // If no sign change, there is no root between
        oldOmega = newOmega; // oldOmega and newOmega
        i++; // Try the next interval
        continue;
    }

    // A root of Q (or P) is found between oldOmega and newOmega; use bisection search to locate it.
    if (evenRoot)
        freq[k] = bisection(B,oldOmega,newOmega); // Looking for root of  $Q(\omega)$ 
    else
        freq[k] = bisection(A,oldOmega,newOmega); // Looking for root of  $P(\omega)$ 

    // Now, we locate the next root of P (or Q), starting from the root that we have just found
    oldOmega = freq[k];
    if (evenRoot)
        oldQ = funcPQ(A,oldOmega); // Start from the current root
    else
        oldQ = funcPQ(B,oldOmega); // Start from the current root
    k += 1;
    evenRoot = !evenRoot; // Alternate root
}

// Convert to normalized frequencies, i.e.  $f=0\dots0.5$ 
for (i=0; i<p; i++)
    freq[i] = freq[i]/(2*PI);

// Check ill-condition cases
*ispflag = CheckIllCondition(freq,lastfreq,p);

// Prepare for next pass
for (i=0; i<p; i++)
    lastfreq[i] = freq[i];
}

```

```

/*
  Code added by MW Mak
  funcPQ: return the value of P(omega) or Q(omega) depending on the values of c[]. This function Implements Eqn. 4
*/
float funcPQ(float c[], float omega)
{
    float value;
    int i;

    value = 0.0;
    for (i=0; i<5; i++)
        value += c[i]*cos((5-i)*omega);
    value += c[5]*0.5;
    return value;
}

/*
  Code derived from Numerical Recipe in C
  bisection: perform bisection search for root between x1 and x2
*/

#define JMAX 40
float bisection(float coeff[], float x1, float x2)
{
    int j;
    float dx,f,fmid,xmid,rtb;
    float xacc = EPS;

    f=funcPQ(coeff,x1);
    fmid=funcPQ(coeff,x2);
    if (f*fmid >= 0.0) {
        printf("Root must be bracketed for bisection search.\n");
        return 0;
    }
    if (f<0.0) {
        rtb = x1;
        dx = x2-x1;
    } else {
        dx = x1-x2;
        rtb = x2;
    }
    for (j=1;j<=JMAX;j++) {
        dx *= 0.5;
        xmid = rtb+dx;
        fmid=funcPQ(coeff,xmid);
        if (fmid <= 0.0)
            rtb=xmid;
        if (fabs(dx) < xacc || fmid == 0.0)
            return rtb;
    }
    printf("Too many bisections in bisection()\n");
    return 0;
}
#undef JMAX

```

3.4 QuantizeLsp [lsp34.c]

This function quantizes the LSP parameters according to the bit allocation table `lspAlloc[0...9]`, and returns the quantization index in `lspIndex[0...9]` and the quantized frequencies in `lsp[0...9]`. The quantization process checks and ensures the monotonicity of the quantized LSP. Whenever the quantized frequencies are not monotonic, the quantization is adjusted by either quantizing the current LSP to the next higher level or by quantizing the previous LSP to the next lower level, depending on which option gives the minimum quantization error. For example, the following diagram shows the case where the quantized value of `lsp[5]` is smaller than that of `lsp[4]`.

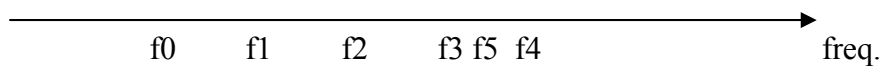


Fig. 2 Quantized LSPs on the frequency axis

In this example, we could either increase `f5` or decrease `f4` to ensure monotonicity.

```
QuantizeLsp(float lsp[], int pOrder, int lspAlloc[], int lspIndex[])
{
    int levels, i, j;
    float dist, low, errorup, errordn;

    for (i = 0; i < pOrder; i++) {
        // Sequentially find closest quantized LSP indexes
        lsp[i] *= 8000; // Make freq. scale compatible with lspTbl[][]
        levels = (1 << lspAlloc[i]) - 1; // No. of levels = 2^lspAlloc[i]
        low = fabs(lsp[i] - lspTbl[i][0]); // Search from lspTbl[i][0] to lspTbl[i][levels] to find the
        lspIndex[i] = 0; // closest frequency to lsp[i]. Then, the index to the closest
        for (j = 1; j <= levels; j++) { // frequency in lspTbl[] is assigned to lspIndex[i]
            dist = fabs(lsp[i] - lspTbl[i][j]);
            if (dist < low) {
                low = dist; // Update lowest dist found so far
                lspIndex[i] = j; // Update index to closest freq.
            }
        }

        // Adjust quantization if non-monotonically quantized values of lsp occur
        if (i > 0) {
            if (lspTbl[i][lspIndex[i]] <= lspTbl[i - 1][lspIndex[i - 1]]) {

                // Compute error for quantizing the current lsp (lsp[i]) to the next higher level
                // while keeping the quantizing level of the previous lsp unchanged
                errorup = fabs(lsp[i] - lspTbl[i][mmin(lspIndex[i] + 1, levels)]) +
                    fabs(lsp[i - 1] - lspTbl[i - 1][lspIndex[i - 1]]);

                // Compute error for quantizing the previous lsp (lsp[i-1]) to the next
                // lower level while keeping the quantizing level of the current lsp unchanged
                errordn = fabs(lsp[i] - lspTbl[i][lspIndex[i]]) +
                    fabs(lsp[i - 1] - lspTbl[i - 1][mmax(lspIndex[i - 1] - 1, 0)]);

                // Adjust index for minimum error and preserve monotonicity
                if (errorup < errordn) {
```

```

// Round up: using the next higher level for the current lsp
lspIndex[i] = mmin(lspIndex[i] + 1, levels);

// Use higher level to preserve monotonicity
while (lspTbl[i][lspIndex[i]] < lspTbl[i-1][lspIndex[i-1]])
    lspIndex[i] = mmin(lspIndex[i] + 1, levels);
}
else if (i == 1) // Round down
    // i=1 is a special case for round down
    // Use the next lower level for the previous lsp. Monotonicity is
    // automatically preserved because of the characteristics of lspTbl[ ][ ].
    // For example, if lsp[0]=379 and lsp[1]=379.5, then lspIndex[0]=5
    // and lspIndex[1]=5. As a result, lspTbl[0][lspIndex[0]]=340, which
    // is larger than lspTbl[1][lspIndex[1]]=360. Therefore, we only need
    // to reduce lspIndex[0] by 1 to yield lspTbl[0][lspIndex[0]]=280, which
    // is now smaller than lspTbl[1][lspIndex[1]]
    lspIndex[i - 1] = mmax(lspIndex[i - 1] - 1, 0);

// Need to check whether round down will lead to non-monotonicity in lower
// order lsp
else if (lspTbl[i - 1][mmax(lspIndex[i - 1] - 1, 0)] > lspTbl[i - 2][lspIndex[i - 2]])
    lspIndex[i - 1] = mmax(lspIndex[i - 1] - 1, 0);

// Impossible to ensure monotonicity by rounding down, use round up
// instead.
else {
    lspIndex[i] = mmin(lspIndex[i] + 1, levels);
    while (lspTbl[i][lspIndex[i]] < lspTbl[i-1][lspIndex[i-1]])
        lspIndex[i] = mmin(lspIndex[i] + 1, levels);
}
}
}

// Quantize lsp frequencies using indexes found above
for (i = 0; i < pOrder; i++)
    lsp[i] = lspTbl[i][lspIndex[i]] / 8000;
}

```

```

#define mmax(A,B) ((A)>(B)?(A):(B))
#define mmin(A,B) ((A)<(B)?(A):(B))

```

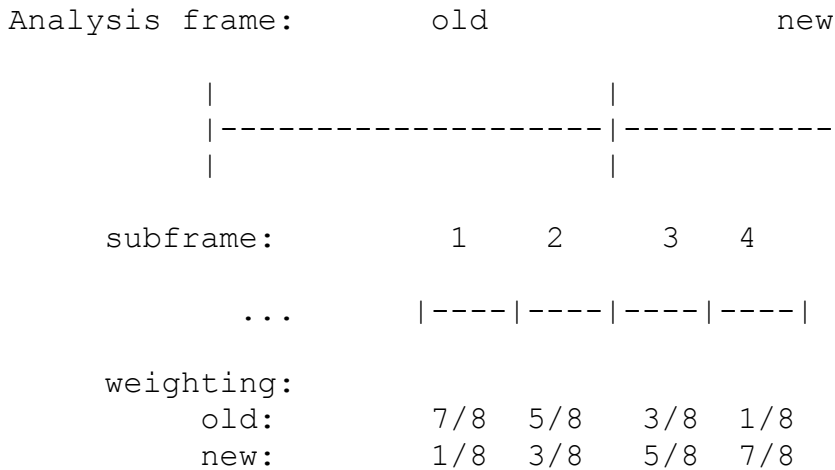
```

lspTbl[0] = {100, 170, 225, 250, 280, 340, 420, 500, 0, 0, 0, 0, 0, 0, 0, 0}
lspTbl[1] = {210, 235, 265, 295, 325, 360, 400, 440, 480, 520, 560, 610, 670, 740, 810, 880}
lspTbl[2] = {420, 460, 500, 540, 585, 640, 705, 775, 850, 950, 1050, 1150, 1250, 1350, 1450, 1550}
lspTbl[3] = {620, 660, 720, 795, 880, 970, 1080, 1170, 1270, 1370, 1470, 1570, 1670, 1770, 1870, 1970}
lspTbl[4] = {1000, 1050, 1130, 1210, 1285, 1350, 1430, 1510, 1590, 1670, 1750, 1850, 1950, 2050, 2150, 2250}
lspTbl[5] = {1470, 1570, 1690, 1830, 2000, 2200, 2400, 2600, 0, 0, 0, 0, 0, 0, 0, 0}
lspTbl[6] = {1800, 1880, 1960, 2100, 2300, 2480, 2700, 2900, 0, 0, 0, 0, 0, 0, 0, 0}
lspTbl[7] = {2225, 2400, 2525, 2650, 2800, 2950, 3150, 3350, 0, 0, 0, 0, 0, 0, 0, 0}
lspTbl[8] = {2760, 2880, 3000, 3100, 3200, 3310, 3430, 3550, 0, 0, 0, 0, 0, 0, 0, 0}
lspTbl[9] = {3190, 3270, 3350, 3420, 3490, 3590, 3710, 3830, 0, 0, 0, 0, 0, 0, 0, 0}

```

3.5 InterpolateLsp [intanaly.c]

This function linearly interpolates LSPs for CELP subframe analysis. This is a combination of inter- and intra-frame interpolation. The LSPs are interpolated from two transmitted frames: old and new. The interpolation is calculated as follows:



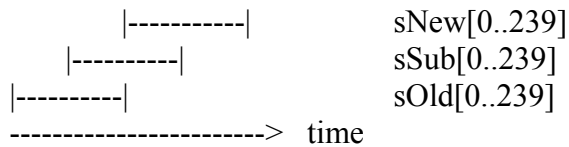
```
InterpolateLsp(float lspNew[],           // LSP of current frame
               float lspSubframe[][10]) // Interpolated subframe LSPs
{
    int i, j;
    static float lspOld[10] = { .03, .05, .09, .13, .19, .23, .29, .33, .39, .44 }; // LSPs of previous frame
    float w[2][4] = {
        0.875, 0.625, 0.375, 0.125,
        0.125, 0.375, 0.625, 0.875
    };

    // For each subframe, interpolate the LSP based on the previous and the current sub-frame
    for (i = 0; i < 4; i++) {
        for (j = 0; j < 10; j++)
            lspSubframe[i][j] = w[0][i]*lspOld[j] + w[1][i]*lspNew[j];

        // Save sub-frame LSPs for the next call
        for (j = 0; j < 10; j++)
            lspOld[j] = lspNew[j];
    }
}
```

3.6 *MakeSubFrame()*

This function makes a buffer `sSub[0..frmSize-1]` for subsequence CELP close-loop analysis. The buffer `sSub[]` will be filled with the last `frmSize/2` samples from `sOld[]` and the first `frmSize/2` samples from `sNew[]`, as follows:



```
MakeSubFrame(float sNew[], float sOld[], float sSub[], int frmSize)
```

```
{
    int i;
    for (i = 0; i < frmSize/2 ; i++) {
        sSub[i] = sOld[i + frmSize/2];
        sSub[i + frmSize/2] = sNew[i];
    }
}
```

3.7 LspToLpc [lsptopc.c]

This function convert the quantized LSPs to LP coefficients.

```

LspToLpc(float lspSubframe[], float lpc[])
{
    int i, j, k, noh, lspflag;
    float freq[10], p[5], q[5];
    float a[6], a1[6], a2[6];
    float b[6], b1[6], b2[6];
    float pi, xx, xf;

    // Initialization
    pi = 3.1415926535897931032;
    noh = 5;
    for (j = 0; j < 10; j++) freq[j] = lspSubframe[j];
    for (i = 0; i < noh + 1; i++)
        a[i] = a1[i] = a2[i] = b[i] = b1[i] = b2[i] = 0.0;

    // Compute  $c[j] = p[j] = -2 \cos \omega_j$  for even  $j$  and  $c[j] = q[j] = -2 \cos \theta_j$  for odd  $j$ , where  $\omega_j$  and  $\theta_j$ 
    // are the even and odd number LSPs given by  $\text{freq}[j] * 2 * \pi / 8000$ .
    for (i = 0; i < noh; i++) {
        p[i] = -2. * cos(2. * pi * freq[2 * i]);
        q[i] = -2. * cos(2. * pi * freq[2 * i + 1]);
    }

    // Compute the impulse response of  $A(z) = \frac{1}{2} [P_{11}(z) + Q_{11}(z)]$ 
    xf = 0.0;
    for (k = 0; k < 11; k++) {
        xx = 0.0;
        if (k == 0) xx = 1.0;
        a[0] = xx + xf;
        b[0] = xx - xf;
        xf = xx;
        for (i = 0; i < noh; i++) {
            a[i + 1] = a[i] + p[i] * a1[i] + a2[i];
            b[i + 1] = b[i] + q[i] * b1[i] + b2[i];
            a2[i] = a1[i];
            a1[i] = a[i];
            b2[i] = b1[i];
            b1[i] = b[i];
        }
        if (k != 0)
            lpc[k - 1] = -.5 * (a[noh] + b[noh]);
    }

    // Convert to the configuration of LP parameters
    for (i = 9; i >= 0; i--)
        lpc[i + 1] = -lpc[i];
    lpc[0] = 1.0;
}

```

3.8 CodebookSearch [csub.c]

This function determines the adaptive codebook index (tauptr) and gain (pindex) of the current subframe. It then determines the stochastic codebook index (cbindex) and gain index (gindex) based on the adaptive codebook vector selected in the first stage.

```
CodebookSearch(float s[], float v[])           // s[0...59]: Current subframe
{                                               // v[0...59]: Excitation vector upon return
    // Initialize e0[0...59] to zero
    setr(60, 0.0, e0);

    // Compute target and remove zero input responses (memory responses). More specifically,
    // e0[0...59] will be equal to  $W(\mathbf{s} - \hat{\mathbf{s}}^{(0)})$ , see Eqn. 18.
    ComputeInitialState(s, d1a, d2a, d3a, d4a, 0, 1, 1, 1);

    movefr(11, d2b, d2a);                       // Copy memory of  $1/A(z)$ , from d2b[] to d2a[]
    movefr(11, d3b, d3a);                       // Copy memory of  $A(z)$ 
    movefr(11, d4b, d4a);                       // Copy memory of  $1/A(z/\gamma)$ 
                                                // Note that d1a have not been updated, so no need to copy

    // Compute impulse response of  $\frac{W(z)}{A(z)} = W(z)H(z) = \left(1 + \sum_{k=1}^{10} a_k \gamma^k\right)^{-1}$ 
    ComputeImpulseResponse(lpc, h, 60);

    // Prepare for modifying adaptive codebook gain. Determine the cross-correlation without taking the
    // the pitch synthesis filter into account
    PrepareModifyAdpCbkGain();                 // Compute  $\|W(\mathbf{s} - \hat{\mathbf{s}}^{(0)})\|$ 

    // Find optimal pitch lag (tauptr) and its associated gain (pindex)
    PitchSearch();

    // Find initial error with pitch VQ. After the function call, e0[0...59] will contain the target excitation
    // for the pitch synthesis filter and the perceptually weighted LP filter (Eqn. 18). Note: d1a, d2a,
    // d3a, and d4a will be updated.
    setr(60, 0.0, e0);
    ComputeInitialState(s, d1a, d2a, d3a, d4a, 1, 1, 1, 1);

    // Determine the cross-correlation with the pitch synthesis filter taken into account.
    // e0[] =  $W(\mathbf{s} - \hat{\mathbf{s}}^{(0)}) - WH\mathbf{u}$  before calling this function
    ResidualCrossCorrelation();

    // Search the stochastic code book and fill excitation vector v[] with gain scaled code word
    // Update stochastic codebook index (cbindex) and gain (gindex). It will call ModifyAdpCbkGain()
    StochasticCbkSearch(60, v);

    // Update filter states for the next pass
    movefr(60, v, e0);
    ComputeInitialState(s, d1b, d2b, d3b, d4b, 1, 1, 1, 1);
    movefr(idb, d1b, d1a);
    movefr(11, d2b, d2a);
    movefr(11, d3b, d3a);
    movefr(11, d4b, d4a);
}
```

3.9 ComputeInitialState [config.c]

This function compute the initial states of all filters ($1/P(z)$, $1/A(z)$, $A(z)$ and $1/A(z/0.8)$) according to the values of four switches: isw1, isw2, isw3, and isw4. If isw1=0 and isw2=isw3=isw4=1, this function computes $e0[0...59] = W(\mathbf{s} - \hat{\mathbf{s}}^{(0)})$. If all switches are equal to 1, this function computes $e0[0...59] = W(\mathbf{s} - \hat{\mathbf{s}}^{(0)}) - WH\mathbf{u}$, where \mathbf{u} is the excitation in the previous stage.

```
// s[0...59]: speech of the current subframe
// d1[0...10]: memory of 1/P(z)
// d2[0...10]: memory of 1/A(z)
// d3[0...10]: memory of A(z)
// d4[0...10]: memory of 1/A(z/0.8)
ComputeInitialState(float s[], float d1[], float d2[], float d3[], float d4[] int isw1, int isw2, int isw3, int isw4)
{
    float fctemp[11];           // Bandwidth expanded lpc
    int i;

    setr(11, 0.0, fctemp);     // Set bandwidth expanded lpc to zero

    // Pitch synthesis: This function outputs synthetic speech due to the filter memory of 1/P(z) and the
    // input e0[0...59]. e0 is a zero vector of length 60 when the CELP coder is initialized
    // idb = 209 defines the length of the pitch search buffer.
    // bb[0..3] defines the pitch synthesis filter, with bb[0] = pitch delay,
    // bb[2] = pitch gain, and bb[1]=bb[3]=0.0
    if (isw1 != 0)
        PitchSynthesis(e0, 60, d1, 209, bb); // e0[0...59]=u

    // Implement synthesis filter 1/A(z), output synthetic speech in e0. When the coder is initialized,
    // e0[0..59] = 0.0. A call to polefilt() with e0[] = 0 is equivalent to computing the zero input response
    // (see Campbell et al. 1991 p. 151, Eq. 12).
    if (isw2 != 0) // If isw1 equals to 0, compute e0[] =  $\hat{\mathbf{s}}^{(0)}$ 
        polefilt(lpc, 10, d2, e0, 60); // If isw1 not equals to 0, compute  $e0[] = \hat{\mathbf{s}}^{(0)} + H\mathbf{u}$ 

    // Remove the zero input response
    for (i = 0; i < 1; i++) // If isw1=0,  $e0[] = \mathbf{s} - \hat{\mathbf{s}}^{(0)}$ 
        e0[i] = s[i] - e0[i]; // If isw1!=0,  $e0[] = \mathbf{s} - \hat{\mathbf{s}}^{(0)} - H\mathbf{u}$ 

    // Implement the perceptual weighting filter, i.e.  $W(z) = A(z)/A(z/\text{gamma})$  and update its memory
    if (isw3 != 0)
        zerofilt(lpc, 10, d3, e0, 60);
    if (isw4 != 0) {
        bwexp(0.8, lpc, fctemp, 10); //  $a'_k = a_k (0.8)^k; k = 0, \dots, 10$ 
        polefilt(fctemp, 10, d4, e0, 60); // Compute  $e0[] = W(\mathbf{s} - \hat{\mathbf{s}}^{(0)} - H\mathbf{u})$  or  $e0[] = W(\mathbf{s} - \hat{\mathbf{s}}^{(0)})$ 
    }
}
```

3.10 ComputeImpulseResponse [impulse.c]

This function computes the impulse response of the perceptual LP filter

$$\frac{W(z)}{A(z)} = W(z)H(z) = \left(1 + \sum_{k=1}^{10} a_k \gamma^k\right)^{-1}$$

where $\gamma = 0.8$.

```

ComputeImpulseResponse(float lpc[], float h[], int len)
{
    float z[11];           // Filter memory
    float fctemp[11];     // Bandwidth expanded LP

    setr(len, 0.0, h); h[0] = 1.0; // Make impulse
    setr(11, 0.0, z);      // clear the filter memory
    bwexp(0.8, lpc, fctemp, 10); // Bandwidth expansion
    polefilt(fctemp, 10, z, h, len); // Impulse response stored in h[0...len-1]
}

bwexp(alpha, pc, pcexp, n)
int n;
float alpha, pc[], pcexp[];
{
    int i;
    float factor[11];
    for (i = 0; i <= n; i++)
        factor[i] = (float)pow(alpha,(double)(i));
    for (i = 0; i <= n; i++)
        pcexp[i] = pc[i]*factor[i];
}

```

3.11 PrepareModifyAdpCbkgain [mexcite1() of mexcite.c]

ResidualCrossCorrelation [mexcite2() of mexcite.c]

ModifyAdpCbkgain [mexcite3() of mexcite.c]

This three functions work together to implement the modified excitation as specified in FS1016. The relative adaptive code book excitation component is increased in voiced regions by decreasing the stochastic code book excitation component. The amount of reduction in the stochastic component depends on the efficiency of the adaptive component. More reconstruction burden is placed on the adaptive component when its efficiency is high. The efficiency is measured by the closeness (in the cross-correlation sense) of the residual signals before and after pitch prediction. When the efficiency is high (e.g. > 0.9), the stochastic component is amplified slightly (e.g., by one quantizer level). This modification of adaptive codebook gain provides a perceptually superior synthetic speech.

The procedure for modifying the stochastic gain outside the search loop is:

- 1) Measure the efficiency of the adaptive component (ccor)
- 2) Search the stochastic code book for the optimal codeword
- 3) Modify the stochastic code book gain

Mathematically, the three functions implement the following equations:

$$R = \frac{\langle W(\mathbf{s} - \hat{\mathbf{s}}^{(0)}), W(\mathbf{s} - \hat{\mathbf{s}}^{(0)}) - WH\mathbf{u} \rangle}{\|W(\mathbf{s} - \hat{\mathbf{s}}^{(0)})\|^2} \quad (6)$$

where \langle, \rangle denotes cross-correlation and $\| \cdot \|$ denotes Euclidean norm. The gain is modify as follows:

$$g'_i = \begin{cases} 0.2g_i & \text{if } |R| < 0.04 \\ 1.4g_i\sqrt{|R|} & \text{if } |R| > 0.81 \\ g_i\sqrt{|R|} & \text{otherwise} \end{cases} \quad (7)$$

```
extern float e0[60];
```

```
static float ccor;
```

```
static float e1, e0save[60];           // e1 = Euclidean norm of the first error signal, \|W(s - s^(0))\|^2
```

```
PrepareModifyAdpCbkgain()           // Note: e0[] = W(s - s^(0))
```

```
{
    int i;

    e1 = 1e-6;                         // Just in case the energy is zero
    for (i = 0; i < 60; i++) {
        e0save[i] = e0[i];             // Prepare for the call to ResidualCrossCorrelation()
        e1 += e0[i] * e0[i];           // Compute \|W(s - s^(0))\|^2
    }
}
```

```
// Compute cross-correlation (corr) of the residual signals before and after pitch prediction
```

```
ResidualCrossCorrelation()           // Note: e0[] = W(s - s^(0)) - WHu
```

```
{
    int i;
    ccor = 1e-6;                       // Just in case the cross-correlation is zero
    for (i = 0; i < 60; i++)
```

```
        ccor += e0[i] * e0save[i];           // Compute  $\langle W(\mathbf{s} - \hat{\mathbf{s}}^{(0)}), W(\mathbf{s} - \hat{\mathbf{s}}^{(0)}) - WH\mathbf{u} \rangle$   
                                           // where  $e0save[] = W(\mathbf{s} - \hat{\mathbf{s}}^{(0)})$   
    ccor = ccor / e1;                       // Compute R  
}  
  
// Modify the codebook gain according to Eqn. 7 above  
ModifyAdpCbkgain(float *cgain)  
{  
    float scale;  
    scale = sqrt(fabs(ccor));  
    if (scale < 0.2)  
        scale = 0.2;                       //  $|R| < 0.04$   
    else if (scale > 0.9)  
        scale = scale * 1.4;              //  $|R| > 0.81$   
    *cgain = *cgain * scale;  
}
```

3.12 PitchGain [pgain.c]

This function returns the match score and its corresponding gain for a given excitation vector. The function's operation can be divided into two phases. It begins with passing the excitation vector $\text{ex}[0\dots59]$ ($\mathbf{x}^{(l)}$ in Campbell, 1993 terminology) through the perceptually weighted LP filter $H_w(z)$. More specifically, it computes

$$\mathbf{y}^{(m)} = H_w \mathbf{x}^{(m)} \quad (8)$$

where the superscript m denotes the current pitch lag selected in *PitchSearch()*. In order to reduce computation overhead, end-point correction is used. This technique makes use of the results in the previous passes as the basis for the current pass. Denote $x[0\dots59]$ as the input to the filter $H_w(z)$ in the first call to this function, then $y[0\dots59]$ (the output) should be computed as follows:

$$y[n] = \sum_{j=0}^{\min(n, L'-1)} h[j]x[n-j] \quad L' \leq 60 \quad (9)$$

where $h[j]$, $j = 0, \dots, L' - 1$, denotes the truncated impulse response of $H_w(z)$. Now, if we delay $x[0\dots59]$ by 1 sample unit such that $x'[0] = x_{\text{new}}$, $x'[1] = x[0], \dots, x'[59] = x[58]$, then the new output will become

$$y'[n] = \begin{cases} h[0]x'[0] & n = 0 \\ \sum_{j=0}^{\min(n, L'-1)} h[j]x'[n-j] = \sum_{j=0}^{\min(n-1, L'-1)} h[j]x[(n-1)-j] + h[n]x'[0] = y[n-1] + h[n]x'[0] & n = 1, \dots, L'-1 \\ y[n-1] & n = L', \dots, 59 \end{cases} \quad (10)$$

When $m < 60$ (the subframe length), the excitation $\mathbf{x}^{(m)}$ is constructed by replicating the last m samples of the excitation history, as shown in the figure below.

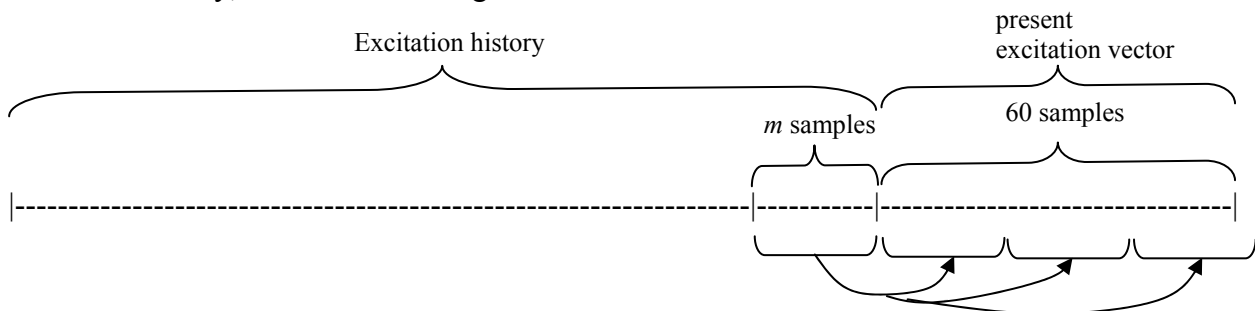


Fig. 3. Replication of excitation signal

Let $x[0\dots59]$ be the excitation sequence without the replication of delayed samples, i.e. $x[0\dots m-1]$ contain the last m samples of the excitation history and $x[m] = x[m+1] = \dots = x[59] = 0$. We also let $x_r[0\dots59]$ be the excitation sequence with the replication of delayed samples. Therefore, we have for $n = 0, \dots, m-1$

$$x_r[n] = x[n] \quad (11a)$$

$$x_r[m+n] = x[n] \quad (11b)$$

$$x_r[\min(2m+n, 59)] = x[n]. \quad (11c)$$

Now, let $y[0\dots59]$ and $y_r[0\dots59]$ be the filter output for input $x[0\dots59]$ and $x_r[0\dots59]$ respectively. For $n = 0, \dots, m-1$, the filter output is

$$y_r[n] = \sum_{j=0}^{\min(n, L'-1)} h[j]x_r[n-j] = \sum_{j=0}^{\min(n, L'-1)} h[j]x[n-j] = y[n] \quad (12)$$

Note that we can use Eqn. 11a to simplify $y_r[\cdot]$ because the indexes to $x_r[\cdot]$ is in the range $[0, m-1]$.

The filter outputs $y_r[m \dots 2m - 1]$ are

$$\begin{aligned}
 y_r[n + m] &= \sum_{j=0}^{\min(n+m, L'-1)} h[j] x_r[m + n - j] \\
 &= \sum_{j=0}^n h[j] x_r[m + n - j] + \sum_{j=n+1}^{\min(n+m, L'-1)} h[j] x_r[m + n - j] \\
 &= \sum_{j=0}^n h[j] x[n - j] + \sum_{j=n+1}^{\min(n+m, L'-1)} h[j] x[m + n - j] \\
 &= y[n] + \sum_{j=0}^{\min(n+m, L'-1)} h[j] x[m + n - j] \\
 &= y[n] + y[n + m]
 \end{aligned} \tag{13}$$

By changing the index to $i = n + m$ s.t. $i \in [m, 59]$, we can express Eqn. 13 in the form

$$y_r[i] = y[i - m] + y[i]. \tag{14}$$

Note that we have made use of the fact that the indexes to $x_r[\cdot]$ in the first term of Eqn. 13 are in the range $[m, 2m - 1]$ and the indexes in the second term are in the range $[0, m - 1]$. Hence, we can apply Eqn. 11b for simplifying the first term of Eqn. 13 and Eqn. 11a for simplifying the second term of Eqn. 13.

Similarly the filter outputs $y_r[2m \dots 59]$ are

$$\begin{aligned}
 y_r[n + 2m] &= \sum_{j=0}^{\min(n+2m, L'-1)} h[j] x_r[2m + n - j] \\
 &= \sum_{j=0}^n h[j] x_r[2m + n - j] + \sum_{j=n+1}^{\min(n+m, L'-1)} h[j] x_r[2m + n - j] + \sum_{j=n+m+1}^{\min(n+2m, L'-1)} h[j] x_r[2m + n - j] \\
 &= \sum_{j=0}^n h[j] x[n - j] + \sum_{j=n+1}^{\min(n+m, L'-1)} h[j] x[m + n - j] + \sum_{j=n+m+1}^{\min(n+2m, L'-1)} h[j] x[2m + n - j] \\
 &= y[n] + \sum_{j=0}^{\min(n+m, L'-1)} h[j] x[m + n - j] + \sum_{j=0}^{\min(n+2m, L'-1)} h[j] x[2m + n - j] \\
 &= y[n] + y[m + n] + y[2m + n].
 \end{aligned} \tag{15}$$

Again, the indexes to $x_r[\cdot]$ in the first, second and third terms of Eqn. 15 are in the range $[2m, 59]$, $[m, 2m - 1]$ and $[0, m - 1]$ respectively. Hence we have used Eqns. 11c, 11b and 11a respectively to simplify the three terms of Eqn. 15. If we change the index to $i = n + 2m$ s.t. $i \in [2m, 59]$, we can express Eqn. 15 in the following form:

$$\begin{aligned}
 y_r[i] &= y[i - 2m] + y[i - m] + y[i] \\
 &= y[i - 2m] + y'_r[i]
 \end{aligned} \tag{16}$$

where $y'_r[2m \dots 59]$ are the samples in $y_r[\cdot]$ computed in (14).

In the second phase, **PitchGain()** computes the cross correlation $\mathbf{y}^{(m)T} \mathbf{e}^{(0)}$ and energy $\mathbf{y}^{(m)T} \mathbf{y}^{(m)}$ in order to determine the gain

$$\mathcal{G}_m = \frac{\mathbf{y}^{(m)T} \mathbf{e}^{(0)}}{\mathbf{y}^{(m)T} \mathbf{y}^{(m)}} \tag{17}$$

where $\mathbf{e}^{(0)}$ is defined as

$$\mathbf{e}^{(0)} = W(\mathbf{s} - \hat{\mathbf{s}}^{(0)}) - WH\mathbf{u} \quad (18)$$

where $\hat{\mathbf{s}}^{(0)}$ is the zero input response (memory response) and \mathbf{u} is the excitation vector of the previous stage. Then the matching score

$$S_m = \mathbf{g}_m \mathbf{y}^{(m)T} \mathbf{e}^{(0)} \quad (19)$$

is computed.

```

float PitchGain(float ex[]      //  $\mathbf{x}^{(m)}$  : Excitation signal
                int first      // ==0 if end-point correction is required to compute filter output
                int m          // Pitch lag
                int len,       //  $L'$  : Length of truncated impulse response
                float *match)   // Match score  $S_m$ 
{
    float cor, eng;           // Cross-correlation and energy
    float yr[60];            //  $\mathbf{y}^{(m)T}$  : filter's output, with lag smaller than 60 taken care of
    float pgain;             //  $\mathbf{g}_m$  : Pitch gain
    static float y[60];      // filter's output before taking care of  $m < 60$ 
    int i, j;

    // First phase
    if (first) {
        for (i = 0; i < 60; i++) { // Implement Eqn. 9
            y[i] = 0.0;
            for (j = 0; j <= i && j < len; j++)
                y[i] += h[j] * ex[i - j]; //  $h[j]$  is computed in ComputeImpulseResponse()
        }
    } else { // Implement Eqn. 10
        for (i = len - 1; i > 0; i--)
            y[i - 1] += ex[0] * h[i]; // Eqn. 10b
        for (i = 59; i > 0; i--) // Eqn. 10c
            y[i] = y[i - 1]; //  $y[j]$  represents both  $y[j]$  and  $y'[j]$  in Eqn. 10
        y[0] = ex[0] * h[0]; // Eqn. 10a
    }

    // For pitch lags ( $m$ ) shorter than the frame size (60), we replicate the short adaptive codeword to
    // fill the full codeword length by overlapping and adding
    for (i = 0; i < 60; i++)
        yr[i] = y[i];
    if (m < 60) {
        for (i = m; i < 60; i++) // Replicate codeword
            yr[i] = y[i] + y[i - m]; // Implement Eqn. 14
        if (m < 30) {
            for (i = 2 * m; i < 60; i++)
                yr[i] = yr[i] + y[i - 2 * m]; // Implement Eqn. 16
        }
    }

    // Calculate correlation and energy
    cor = 0.0;
    eng = 0.0;
    for (i = 0; i < 60; i++) {
        cor += yr[i] * e0[i]; // Implement  $\mathbf{y}^{(m)T} \mathbf{e}^{(0)}$ 
    }
}

```

```
        eng += yr[i] * yr[i];           // Implement  $\mathbf{y}^{(m)T} \mathbf{y}^{(m)}$ 
    }

    if (eng <= 0.0)
        eng = 1.0;
    pgain = cor / eng;                 // Implement Eqn. 17
    *match = cor * pgain;             // Implement Eqn. 19

    return (pgain);
}
```

3.13 PitchSearch [psearch.c]

This function finds the optimal pitch lag (τ_{ptr}) and determines its corresponding gain (p_{index} and $bb[2]$). Close-loop analysis is used to look for the pitch lag that minimizes a modified minimum squared prediction error (MSPE) of the perceptually weighted error signal.

When *PitchSearch()* is called the first time (i.e. the optimal pitch delay of the first subframe is to be searched), the function initializes the pitch gain to zero and the optimal pitch lag to 20. For subsequent calls to *PitchSearch()*, the function determines the search range by checking the subframe number (1, 2, 3 or 4) of the current subframe. For every odd subframe, 128 integer delays and 128 non-integer delays (8 bits) ranging from 20 to 147 samples (see $p_{delay}[0\dots255]$ below) are used. For every even subframe, pitch delays are searched from -31 to $+32$ samples relative to the previous subframe.

Before searching the optimal pitch delay, the excitation history is updated by copying the memory of the pitch synthesis filter $1/P(z)$, $d1b[0\dots208]$, to the excitation history $v0[10\dots218]$. Excitation sequence are constructed by extracting overlapping segments from $v0[]$. For example, in the case of smallest lag ($m=20$), the excitation sequence is $v0[199\dots218]$, meaning that it needs to be replicated (see *PitchGain* above) to form a sequence of 60 samples; in the case of largest integer lag ($m=38$, i.e. $p_{delay}[62]$) in even subframe, the excitation sequence is $v0[181\dots218]$. The figure below depicts the excitation history's structure.

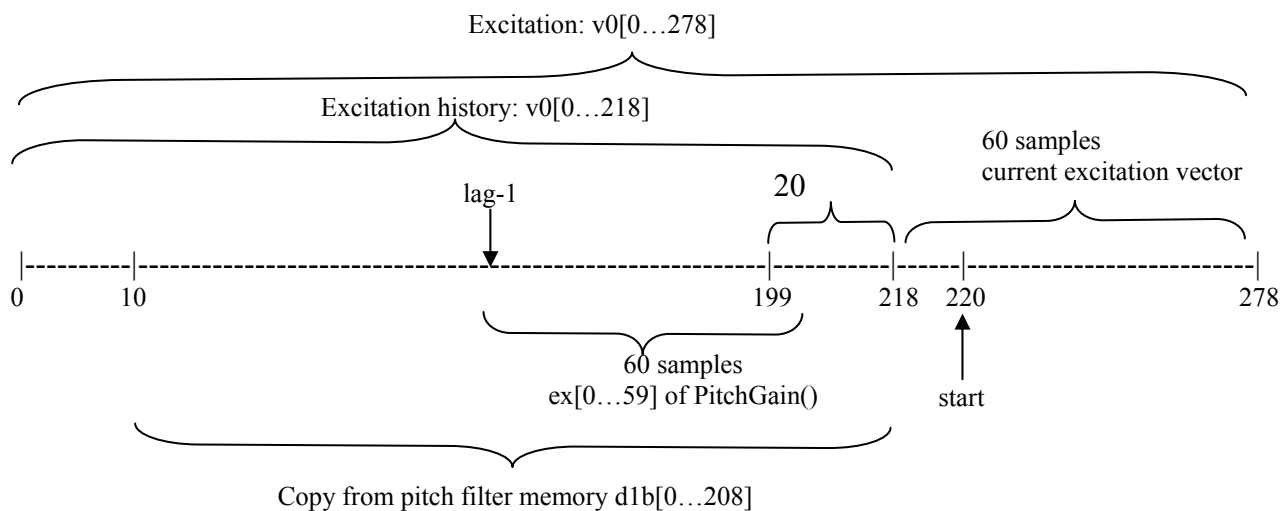


Fig. 4 Structure of excitation history

The function calls *PitchGain* to calculate the gain for each of the integer delays that fall into the allowable range (20 to 147 for odd subframes and -31 to $+32$ samples relative to the previous subframe for the even subframes). Then, the lag that gives the maximum match score (Eqn. 19) is found. To reduce computation complexity, the function checks the neighborhood of the sub-multiples of the optimal integer lag to find the largest score in these sub-multiple lags, where the sub-multiples are defined in the matrix $submult[0\dots255][0\dots3]$. The lag with the largest score in the neighborhood of the sub-multiples will be selected if it is within $\frac{1}{2}$ dB (0.88) of the minimum squared prediction error (MSPE). This strategy ensures smooth pitch contour. Then, the match scores of the neighborhood of the optimal lag and those correspond to the fractional lags are checked. This procedure is called 2-stage hierarchical search because not all fractional lags between 20 and 147 are considered, which is computationally expensive. Instead, only the fractional lags that are close to the optimal integer lags are considered.

PitchSearch()

```

{
    int i; // Index to pdelay[0...255]
    int start;
    int m; // Integer part of lags
    int lag; // Index to v0[0...278]
    int first, bigptr, subptr, topptr, maxptr, bufptr; // Indexes to match[0...255] and pdelay[0...255]
    static int oldptr = {1}; // Optimal int lag in the previous subframe
    float g[256], match[256], emax; // Gain, match score and maximum match score
    int nrange = 3; // Maximum +/- offset to the optimal int lag in
    // neighborhood search
    float v0[279], v0shift[60]; // Excitation buffer and buffer for non-integer lags
    float frac; // Fractional component of non-integer lag

    // Initialize arrays MAXBUFPTR=279, MAXPD=256, LEN=30, bufptr=279
    for (i = 0; i < 279; i++) v0[i] = 0.0;
    for (i = 0; i < 60; i++) v0shift[i] = 0.0;
    for (i = 0; i < 256; i++) g[i] = match[i] = 0.0;
    bufptr = 279;

    // Update adaptive code book (pitch memory)
    // idb = 209, d1b[0..208] is the memory of 1/P(z). Copy d1b[0..208] to &v0[10]
    movefr(idb, d1b, &v0[10]);

    // For the first subframe of the first frame, initialize pitch gain to zero and pitch delay to 20
    if (nseg == 1) {
        bb[2] = 0.0; // Initialize pitch gain to zero. bb[0...3] is a global array containing pitch gain and lag
        bb[0] = 20; // Initialize pitch delay to 20
    } else {
        // Find allowable pointer range (minptr to maxptr)
        if ((nseg % 2) == 0) {
            // Delta delay coding on even subframes. Delays are delta searched and coded with a
            // 6-bit offset relative to the previous subframe
            minptr = oldptr - (plevel2/2 - 1); // Max negative offset to previous subframe
            maxptr = oldptr + (plevel2/2); // Max positive offset to previous subframe
            if (minptr < 0) { // Adjust pointers if out of range
                minptr = 0;
                maxptr = plevel2 - 1; // maxptr = 63 as plevel2 = 64
            }
            if (maxptr > plevel1 - 1) {
                maxptr = plevel1 - 1; // maxptr = 255
                minptr = plevel1 - plevel2; // minptr = 191
            }
        }
        else {
            // Determine the pointers for full range coding on odd subframes
            minptr = 0;
            maxptr = plevel1 - 1; // maxptr = 255
        }
    }

    start = 220
    // Find gains and match scores for integer pitch delays using end-point correction on unity spaced delays
    first = TRUE; // Used in PitchGain() for end-point correction
    for (i = minptr; i <= maxptr; i++) {
        m = (int) pdelay[i]; // Integer part of the delay. pdelay[0..255] = {20.0,...,147.0}
        frac = pdelay[i] - m; // Fractional part of the delay
        if (fabs(frac) < 1.e-4) {
            // There is no fractional delay in pdelay[i], consider integer delay only
            lag = start - m;
        }
    }
}

```

```

        g[i] = PitchGain(&v0[lag-1], 60, first, m, 30, &match[i]);    // See PitchGain() above
        first = FALSE;                                             // Prepare for the next integer lag, for which
    }                                                            // end-point correction will be used
    else
        match[i] = 0.0;                                           // Don't consider all fractional delay (Hierarchical)
}

// Find pointer to top (MSPE) match score (topptr) and determine the best match score
topptr = minptr;                                                 // Initialize pointer
emax = match[topptr];                                           // Initialize max. match score
for (i = minptr; i <= maxptr; i++) {
    if (match[i] > emax) {
        topptr = i;                                             // A bigger match score is found, update pointer
        emax = match[topptr];                                   // and the best match score
    }
}

// Now, we check the submultiples of the optimal lag found above to see whether the best score of these
// submultiples is within 1 dB of the MSPE. For example, for submult[228][]={3,128,64,34},
// pdelay[228]=120, pdelay[128]=60], pdelay[68]=40, and pdelay[34]=30, we search for lags around
// 60, 40, and 30, which are sub-multiples of pitch lag 120. We select the best submultiple if its score is
// within 1dB of MSPE. This approach favors smooth pitch contour.
tauptr = topptr;
if ((nseg % 2) != 0) {
    for (i = 1; i <= submult[topptr][0]; i++) {                // submult[][0]={0,1,2, 3}

        // Find the best neighborhood (+/- 8 offset from the best lag) match for given submultiple
        bigptr = submult[topptr][i];                            // Look at the i-th submultiple
        for (subptr = mmax(submult[topptr][i] - 8, minptr); subptr <= mmin(submult[topptr][i] + 8,
            maxptr); subptr++) {
            if (match[subptr] > match[bigptr])
                bigptr = subptr;
        }

        // Select the submultiple if its score is within 1 dB of MSPE match
        if (match[bigptr] >= 0.88 * match[topptr])
            tauptr = bigptr;
    }
}

// Now tauptr indexes to the optimal lag in pdelay[]. We search tauptr's neighboring delays and find the
// gain and match score for the neighboring delays. We also consider fractional delays around tauptr
bigptr = tauptr;
for (i = mmax(tauptr - nrange, minptr); i <= mmin(tauptr + nrange, maxptr); i++) {
    if (i != tauptr) {
        m = (int) pdelay[i];
        frac = pdelay[i] - m;
        lag = start - m;
        if (fabs(frac) < 1.e-4)
            g[i] = PitchGain(&v0[lag - 1], 60, TRUE, m, 30, &match[i]);
        else {
            FracDelay(v0, start, 60, frac, m, v0shift);
            // TRUE => no end-point correction. 70=> No excitation duplication
            g[i] = PitchGain(v0shift, 60, TRUE, 70, 30, &match[i]);
        }
        if (match[i] > match[tauptr])
            bigptr = i;
    }
}
tauptr = bigptr;                                               // Update the best lag

```

```

// Given chosen pointer to delay (tauptr), recompute its gain to correct errors accumulated in recursions
// and errors due to truncation. This part is optional for floating-point DSP.
m = (int) pdelay[tauptr];
frac = pdelay[tauptr] - m;
lag = start - m;
if (fabs(frac) < 1.e-4)
    g[tauptr] = PitchGain(&v0[lag - 1], 60, TRUE, m, 60, &match[tauptr]);
else {
    FracDelay(v0, start, 60, frac, m, v0shift);
    g[tauptr] = PitchGain(v0shift, 60, TRUE, 70, 60, &match[tauptr]);
}

// Place pitch parameters in bb[]
bb[2] = g[tauptr];
bb[0] = pdelay[tauptr];

// Save pitch pointer for determining delta delay
oldptr = tauptr;
} // End else

// Pitch gain quantization bb[2]
bb[2] = pitchencode(bb[2], &pindex);
}

pdelay[256] = {
20.00000, 20.33334, 20.66667, 21.00000, 21.33334, 21.66667,
22.00000, 22.33334, 22.66667, 23.00000, 23.33334, 23.66667,
24.00000, 24.33334, 24.66667, 25.00000, 25.33334, 25.66667,
26.00000, 26.25000, 26.50000, 26.75000, 27.00000, 27.25000,
27.50000, 27.75000, 28.00000, 28.25000, 28.50000, 28.75000,
29.00000, 29.25000, 29.50000, 29.75000, 30.00000, 30.25000,
30.50000, 30.75000, 31.00000, 31.25000, 31.50000, 31.75000,
32.00000, 32.25000, 32.50000, 32.75000, 33.00000, 33.25000,
33.50000, 33.75000, 34.00000, 34.33334, 34.66667, 35.00000,
35.33334, 35.66667, 36.00000, 36.33334, 36.66667, 37.00000,
37.33334, 37.66667, 38.00000, 38.33334, 38.66667, 39.00000,
39.33334, 39.66667, 40.00000, 40.33334, 40.66667, 41.00000,
41.33334, 41.66667, 42.00000, 42.33334, 42.66667, 43.00000,
43.33334, 43.66667, 44.00000, 44.33334, 44.66667, 45.00000,
45.33334, 45.66667, 46.00000, 46.33334, 46.66667, 47.00000,
47.33334, 47.66667, 48.00000, 48.33334, 48.66667, 49.00000,
49.33334, 49.66667, 50.00000, 50.33334, 50.66667, 51.00000,
51.33334, 51.66667, 52.00000, 52.33334, 52.66667, 53.00000,
53.33334, 53.66667, 54.00000, 54.33334, 54.66667, 55.00000,
55.33334, 55.66667, 56.00000, 56.33334, 56.66667, 57.00000,
57.33334, 57.66667, 58.00000, 58.33334, 58.66667, 59.00000,
59.33334, 59.66667, 60.00000, 60.33334, 60.66667, 61.00000,
61.33334, 61.66667, 62.00000, 62.33334, 62.66667, 63.00000,
63.33334, 63.66667, 64.00000, 64.33334, 64.66667, 65.00000,
65.33334, 65.66667, 66.00000, 66.33334, 66.66667, 67.00000,
67.33334, 67.66667, 68.00000, 68.33334, 68.66667, 69.00000,
69.33334, 69.66667, 70.00000, 70.33334, 70.66667, 71.00000,
71.33334, 71.66667, 72.00000, 72.33334, 72.66667, 73.00000,
73.33334, 73.66667, 74.00000, 74.33334, 74.66667, 75.00000,
75.33334, 75.66667, 76.00000, 76.33334, 76.66667, 77.00000,
77.33334, 77.66667, 78.00000, 78.33334, 78.66667, 79.00000,
79.33334, 79.66667, 80.00000, 81.00000, 82.00000, 83.00000,
84.00000, 85.00000, 86.00000, 87.00000, 88.00000, 89.00000,
90.00000, 91.00000, 92.00000, 93.00000, 94.00000, 95.00000,
96.00000, 97.00000, 98.00000, 99.00000, 100.00000, 101.00000,
102.00000, 103.00000, 104.00000, 105.00000, 106.00000, 107.00000,
108.00000, 109.00000, 110.00000, 111.00000, 112.00000, 113.00000,
114.00000, 115.00000, 116.00000, 117.00000, 118.00000, 119.00000,
120.00000, 121.00000, 122.00000, 123.00000, 124.00000, 125.00000,

```

```

126.00000,    127.00000,    128.00000,    129.00000,    130.00000,    131.00000,
132.00000,    133.00000,    134.00000,    135.00000,    136.00000,    137.00000,
138.00000,    139.00000,    140.00000,    141.00000,    142.00000,    143.00000,
144.00000,    145.00000,    146.00000,    147.00000
}

```

// Truncated submult[][]]. See submult.h for full list

```
float submult[256][4] = {
```

```

    0,  -1,  -1,  -1,
    0,  -1,  -1,  -1,
    0,  -1,  -1,  -1,
    .   .   .   .
    .   .   .   .
    .   .   .   .
    0,  -1,  -1,  -1,
    0,  -1,  -1,  -1,
    0,  -1,  -1,  -1,
    0,  -1,  -1,  -1,
    1,   0,  -1,  -1,
    1,   0,  -1,  -1,
    1,   1,  -1,  -1,
    1,   2,  -1,  -1,
    1,   2,  -1,  -1,
    1,   3,  -1,  -1,
    .   .   .   .
    .   .   .   .
    .   .   .   .
    1,  32,  -1,  -1,
    1,  33,  -1,  -1,
    1,  33,  -1,  -1,
    2,  34,   0,  -1,
    2,  35,   0,  -1,
    2,  35,   1,  -1,
    2,  36,   1,  -1,
    2,  37,   1,  -1,
    2,  37,   2,  -1,
    .   .   .   .
    .   .   .   .
    .   .   .   .
    2,  67,  19,  -1,
    2,  67,  20,  -1,
    2,  68,  20,  -1,
    3,  68,  21,   0,
    3,  70,  22,   1,
    3,  71,  23,   2,
    3,  73,  25,   2,
    3,  74,  26,   3,
    3,  75,  27,   4,
    .   .   .   .
    .   .   .   .
    .   .   .   .
    3, 162,  91,  55,
    3, 164,  92,  56,
    3, 165,  93,  57,
    3, 167,  94,  58,
    3, 168,  95,  58
}

```

3.14 FracDelay [delay.c]

The integer delay m imposes restriction on the optimal pitch search because of the inherent sampling resolution of the signal. The pitch search will be most effective if we could consider the excitation history as a continuous signal such that the best lag m_f (where m_f is a real number) that gives the best similarity between synthetic speech and the perceptually weighted speech is selected. This improvement in temporal resolution can be achieved by the procedure shown in the figure below.

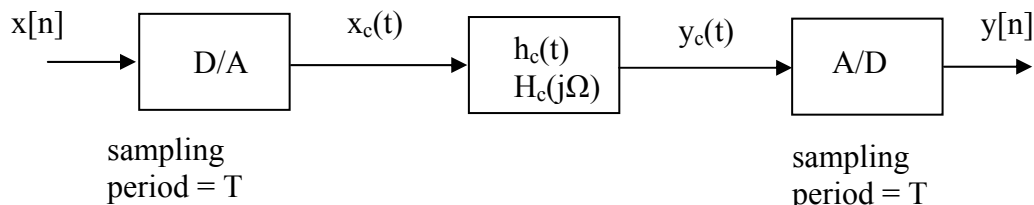


Fig. 5 Estimation of fractional delay by interpolation

First, we convert $x[n]$ to continuous-time signal by interpolation, i.e.

$$x_c(t) = \sum_{k=-\infty}^{\infty} x[k] \frac{\sin(\pi(t - kT)/T)}{\pi(t - kT)/T} \quad (20)$$

Let $H_c(j\Omega)$ be an all-pass linear phase filter with transfer function

$$H_c(j\Omega) = e^{-jm_f\Omega T} \quad (21)$$

Note that the digital domain counterpart of $H_c(j\Omega)$ is $H(e^{j\omega}) = H_c(j\omega/T) = e^{-jm_f\omega}$, which is an all-pass linear phase digital filter. Eqn. 21 implies that $y_c(t)$ is a delayed version of $x_c(t)$. More specifically, we have

$$y_c(t) = x_c(t - m_f T) \quad (22)$$

Substituting Eqn. 20 into Eqn. 22 and using the relationship $y[n] = y_c(nT)$, we obtain

$$\begin{aligned} y[n] &= x_c(nT - m_f T) = \sum_{k=-\infty}^{\infty} x[k] \frac{\sin[\pi(nT - m_f T - kT)/T]}{\pi(nT - m_f T - kT)/T} \\ &= \sum_{k=-\infty}^{\infty} x[k] \frac{\sin[\pi(n - m_f - k)]}{\pi(n - m_f - k)} \end{aligned} \quad (23)$$

Let $m_f = m + f$ where m is an integer and $f = \frac{1}{4}, \frac{1}{3}, \frac{1}{2}, \frac{2}{3}, \frac{3}{4}$, $y[n]$ can be expressed as

$$y[n] = x[n] * \frac{\sin[\pi(n - m - f)]}{\pi(n - m - f)} = \sum_{k=-\infty}^{\infty} \frac{\sin[\pi(k - m - f)]}{\pi(k - m - f)} x[n - k] \quad (24)$$

where $*$ is the convolution operator. Now, we let $j = m - k$, make use of the identity $\sin(-x) = -\sin(x)$, and rearrange the summation index to obtain

$$y[n] = \sum_{j=-\infty}^{\infty} \frac{\sin[\pi(-j - f)]}{\pi(-j - f)} x[n - m + j] = \sum_{j=-\infty}^{\infty} \frac{\sin[\pi(j + f)]}{\pi(j + f)} x[n - m + j] = \sum_{j=-\infty}^{\infty} \frac{\sin[\pi(j + f)]}{\pi(j + f)} x[n - m + j] \quad (25)$$

Eqn. 25 cannot be implemented exactly on digital computers. However, we can truncate the summation using the technique of FIR filter design, as follows:

$$y[n] = \sum_{j=-N/2}^{N/2-1} \frac{\sin[\pi(j + f)]}{\pi(j + f)} h(j + f) x[n - m + j] \quad (26)$$

where

$$h(k) = 0.54 + 0.46 \cos \frac{k\pi}{N/2} \quad -N/2 \leq k \leq N/2 \quad (27)$$

Using the notation of Campbell (1990), Eqns. 26 and 27 can be rewritten as

$$\begin{aligned} y[n] &= \sum_{j=-N/2}^{N/2-1} \frac{\sin[\pi(j+f)]}{\pi(j+f)} h(12(j+f)) x[n-m+j] \\ &= \sum_{j=-N/2}^{N/2-1} w_f[j] x[n-m+j] \end{aligned} \quad (28)$$

where

$$h(k) = 0.54 + 0.46 \cos \frac{k\pi}{6N} \quad (29)$$

Note that Eqn. 28 and Eqn. 29 are identical to (5), (6) and (7) of Campbell (1990).

```

FracDelay(float x[],           // Array v0[] in PitchSearch()
          int start,          // Variable "start" in PitchSearch. Always equal to 220
          int l,              // Subframe length = 60
          int f,              // Fractional part of delay
          int m,              // Integer part of delay
          float y[])          // Output: Excitation with fractional delay incorporated
{
    static float wsinc[8][5] = { // w_f(j); f = 1/4, 1/3, 1/2, 2/3, 3/4 and j = -4, ..., 3
-0.0053321980  -0.0071928948  -0.0104601551  -0.0117125697  -0.0109093422
 0.0232803505  0.0320459865  0.0463415422  0.0497310422  0.0450416803
-0.0809952095  -0.1090071052  -0.1519472152  -0.1591962278  -0.1432515532
 0.2768400609  0.3880135715  0.6143282652  0.8140309453  0.8923586011
 0.8923586011  0.8140309453  0.6143282652  0.3880135119  0.2768400311
-0.1432515532  -0.1591962278  -0.1519472003  -0.1090070903  -0.0809952021
 0.0450416729  0.0497310348  0.0463415347  0.0320459791  0.0232803430
-0.0109093394  -0.0117125660  -0.0104601523  -0.0071928930  -0.0053321971
    }
    int i, j, k, index;

    index = qd(f); // Find the index of closest fractional delay

    for (i = 0; i < l; i++) { // Implement Eqn. 28. Note that k is the running index j in Eqn. 28
        x[start+i-1] = 0.0; // and j is to index w_f(j) in Eqn. 28.
        for (k = -4, j = 0; j < 8; j++) {
            x[start+i-1] += x[start+i-1-m+k] * wsinc[j][index];
            k++;
        }
    }

    // The array v0[] in PitchSearch()/PitchGain() must be zero above "start" because of overlap and
    // add convolution techniques used in PitchGain(). See the first phase of PitchGain().
    for (i = 0; i < l; i++) {
        y[i] = x[start+i-1];
        x[start+i-1] = 0.0;
    }
}

int qd(float f)

```

```
{
    static float frac[5] = {0.25, 0.33333333, 0.5, 0.66666667, 0.75};
    int i, index;

    for (i = 0; i < 5; i++) {
        if (fabs(f - frac[i]) < 1.e-2)
            index = i;
    }
    return(index);
}
```

3.15 LongFracDelay [ldelay.c]

This function is identical to *FracDelay* above, except that it uses a longer window (40 samples) to perform the interpolation.

```

FracDelay(float x[],           // Array v0[] in PitchSearch()
          int start,          // Variable "start" in PitchSearch. Always equal to 220
          int l,              // Subframe length = 60
          int f,              // Fractional part of delay
          int m,              // Integer part of delay
          float y[])          // Excitation with fractional delay incorporated
{
    static float wsinc[40][5] = { //  $w_f(j)$ ;  $f = \frac{1}{4}, \frac{1}{3}, \frac{1}{2}, \frac{2}{3}, \frac{3}{4}$  and  $j = -20, \dots, 19$ 
-0.0009157528  -0.0011301822  -0.0013290340  -0.0011766120  -0.0009726766
  0.0010664382   0.0013298646   0.0015951599   0.0014385806   0.0011996499
-0.0013749709  -0.0017250286  -0.0020920311  -0.0019047754  -0.0015951820
  0.0018631391   0.0023425117   0.0028510056   0.0026026941   0.0021818711
-0.0025545254  -0.0032114214  -0.0039062425  -0.0035624271  -0.0029845031
  0.0034753943   0.0043642647   0.0052961037   0.0048179631   0.0040311604
-0.0046560653  -0.0058386908  -0.0070653162  -0.0064092022  -0.0053549218
  0.0061330274   0.0076801768   0.0092683081   0.0083850641   0.0069964975
-0.0079522254  -0.0099462019  -0.0119744064  -0.0108083375  -0.0090083424
  0.0101742521   0.0127128689   0.0152761098   0.0137634557   0.0114612682
-0.0128828110  -0.0160857141  -0.0193026867  -0.0173693206  -0.0144553734
  0.0161990318   0.0202180520   0.0242434125   0.0218014307   0.0181390028
-0.0203068778  -0.0253437161  -0.0303894430  -0.0273320973  -0.0227433555
  0.0255011376   0.0318392329   0.0382142961   0.0344087631   0.0286503248
-0.0322854556  -0.0403517857  -0.0485420339  -0.0438203216  -0.0365377814
  0.0415940434   0.0520915091   0.0629395396   0.0570935421   0.0477298014
-0.0553679951  -0.0696018860  -0.0847808793  -0.0776027218  -0.0651931837
  0.0783885419   0.0992630646   0.1228656545   0.1145323291   0.0971909687
-0.1263953149  -0.1628061831  -0.2095094770  -0.2046700865  -0.1784717441
  0.2991485298   0.4124546945   0.6357170343   0.8264719844   0.8999969959
  0.8999969959   0.8264719844   0.6357169747   0.4124546349   0.2991485298
-0.1784717441  -0.2046701014  -0.2095094770  -0.1628061682  -0.1263953000
  0.0971909612   0.1145323217   0.1228656545   0.0992630497   0.0783885419
-0.0651931763  -0.0776027218  -0.0847808719  -0.0696018785  -0.0553679913
  0.0477297977   0.0570935383   0.0629395321   0.0520915054   0.0415940396
-0.0365377814  -0.0438203216  -0.0485420302  -0.0403517783  -0.0322854482
  0.0286503229   0.0344087631   0.0382142924   0.0318392292   0.0255011357
-0.0227433518  -0.0273320954  -0.0303894412  -0.0253437087  -0.0203068759
  0.0181390010   0.0218014307   0.0242434107   0.0202180464   0.0161990318
-0.0144553725  -0.0173693206  -0.0193026830  -0.0160857085  -0.0128828101
  0.0114612654   0.0137634547   0.0152761079   0.0127128661   0.0101742502
-0.0090083415  -0.0108083356  -0.0119744046  -0.0099461991  -0.0079522235
  0.0069964956   0.0083850622   0.0092683071   0.0076801744   0.0061330264
-0.0053549209  -0.0064092013  -0.0070653148  -0.0058386894  -0.0046560639
  0.0040311590   0.0048179622   0.0052961023   0.0043642633   0.0034753936
-0.0029845024  -0.0035624264  -0.0039062414  -0.0032114205  -0.0025545249
  0.0021818704   0.0026026936   0.0028510047   0.0023425107   0.0018631388
-0.0015951816  -0.0019047750  -0.0020920306  -0.0017250280  -0.0013749705
  0.0011996495   0.0014385802   0.0015951595   0.0013298644   0.0010664379
-0.0009726765  -0.0011766119  -0.0013290339  -0.0011301821  -0.0009157527
    }
    int i, j, k, index;

    index = qd(f); // Find the index of closest fractional delay from f

    for (i = 0; i < l; i++) { // Implement Eqn. 28. Note that k is the running index in Eqn. 28
        x[start+i-1] = 0.0; // and j is to index  $w_f(j)$  in Eqn. 28.
    }
}

```

```
        for (k = -20, j = 0; j < 40; j++) {
            x[start+i-1] += x[start-m+i+k-1] * wsinc[j][index];
            k++;
        }
    }

    // The v0 array in PitchSearch()/PitchGain() must be zero above "start" because of overlap and
    // add convolution techniques used in PitchGain.
    for (i = 0; i < l; i++) {
        y[i] = x[start+i-1];
        x[start+i-1] = 0.0;
    }
}
```

3.16 PitchSynthesis [pitchvq.c]

This function implement the pitch synthesis filter

$$\frac{1}{P(z)} = \frac{1}{1 - \beta z^{-m}} \quad (30)$$

where β is the pitch gain and m is the pitch lag. For non-integer lag, the function calls **LongFracDelay** that uses a long window for interpolating samples.

```
PitchSynthesis(float rar,           // Storing the data segment to be filtered and the filtered segment
               int idim,           // Dimension of rar[ ]. idim = 60
               float buf[],        // Memory of 1/P(z)
               int idimb,         // Dimension of buf[ ]. idimb = 209
               float b[])         // Define P(z). b[0]=m, b[2]=β
{
    int k, m, i, start;
    float buf2[60];               // Memory after fractional delay being taken care of
    float frac;

    k = idimb - idim;             // k = 209-60=149
    start = k + 1;
    m = (int)b[0];                // Integer part of non-integer delay
    frac = b[0] - m;              // Fractional part of non-integer delay

    // Update memory of 1/P(z)
```

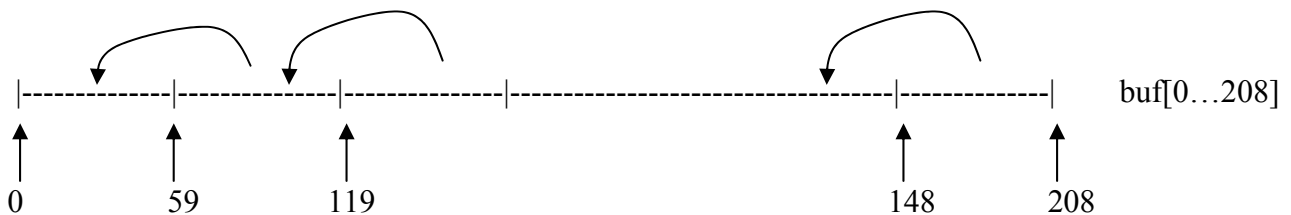


Fig. 6 Update memory of 1/P(z)

```
for (i = 0; i < k; i++)
    buf[i] = buf[i + idim];

// Update memory with selected pitch memory from selected delay (m)
```

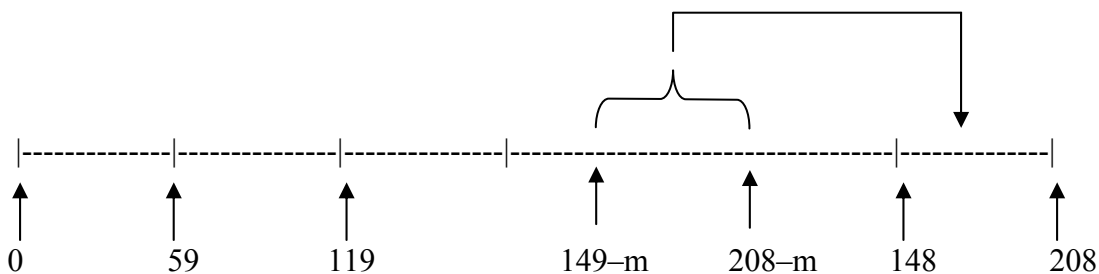


Fig. 7 Make excitation for current subframe

```
if (fabs(frac) < 1.e-4) {
    for (i = k; i < idimb; i++)           // Integer delay
        buf[i] = buf[i - m];
}
if (fabs(frac) > 1.e-4) {               // Fractional delay exists, require interpolation
```

```

LongFracDelay(buf, start, idim, frac, m, buf2);
for (i = 0; i < idim; i++)
    buf[i + k] = buf2[i];
}

```

// Return rar[0...59] as output of pitch synthesis filter

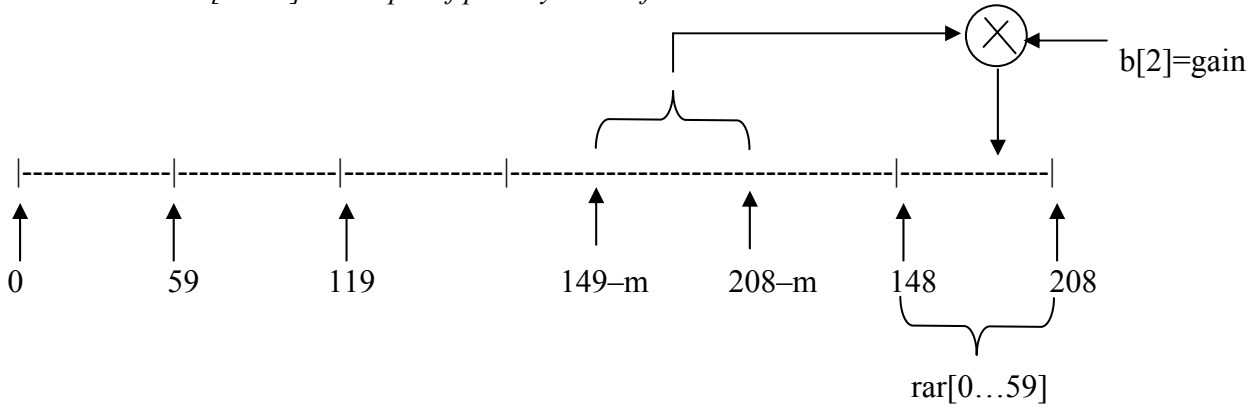


Fig. 8 Gain scale adaptive code vector

```

for (i = 0; i < idim; i++)
    buf[i + k] = rar[i] += b[2] * buf[i + k];
}

```

3.17 StochasticCbkSearch [cbsearch.c]

This function finds the index of the stochastic code vector and its corresponding gain from the stochastic codebook $x[0\dots1083]$, which contains sparse, overlapped (shift by -2), and ternary valued samples ($-1, 0, +1$). The search is performed by closed-loop analysis that aims at minimizing the MSPE of the perceptually weighted error signal.

The stochastic codebook contains $2*512+60=1084$ samples of $-1, 0$, and $+1$. The k -th ($k=0,\dots,512$) codevector c_k is formed by $x[2(512-k)\dots2(512-k)+59]$. For example,

$$c_{512} = x[0\dots59]$$

$$c_{511} = x[2\dots61]$$

$$c_{510} = x[4\dots63]$$

.

.

$$c_1 = x[1022\dots1081]$$

$$c_0 = x[1024\dots1083]$$

Therefore, the excitation vector due to the k -th stochastic code vector is

$$v[i] = x[i+2*(512-k)] * cbkGain \quad i = 0, \dots, 59 \text{ and } k = 0, \dots, 512$$

```

StochasticCbkSearch(float v[]) // Output the stochastic excitation vector in v[0...59]
{
    int i;
    int codeword; // Index to the stochastic codebook x[0...1083]
    float emax; // Maximum match score
    float gain[512]; // Gain of 512 code word
    float err[512]; // Match score of 512 code word
    float quangain; // Gain after quantization

    // Find gain and match score (err[]) for each code word and search for the best code word
    // with max match score. Codewords are overlapped by shifts of -2 along the code vector x[]
    // Note that the code word x[1024...1083] is not used and cbindex={1,...,512}
    codeword = 1022; // Initialize index to point to c1
    cbindex = 1; // Initialize code word index
    gain[0] = CodebookGain(&x[codeword], 60, TRUE, 30, &err[0]);
    emax = err[0];
    codeword -= 2; // Index to the next codeword, c2
    for (i = 1; i < 512; i++) {
        gain[i] = CodebookGain(&x[codeword], 60, FALSE, 30, &err[i]);
        codeword -= 2;
        if (err[i] >= emax) {
            emax = err[i]; // Update error max found so far
            cbindex = i + 1; // Update codebook index. cbindex={1,...,512}
        }
    }

    // Now, x[2*(512-cbindex)...2*(512-cbindex)+59] is the best code word and its corresponding
    // match score is stored in emax. We need to calculate the gain due to this code word. Note that
    // err[] and gain[] start from index 0. This part is optional for floating-point DSP
    codeword = 2*(512 - cbindex); // Adjust index to x[], which points to the best code word
    gain[cbindex-1] = CodebookGain(&x[codeword], 60, TRUE, 60, &err[cbindex-1]);

    // Update gain based on modified MPSE

```

```
ModifyAdpCbkGain(&gain[cbindex - 1]);
```

```
// Gain quantization
```

```
gain[cbindex-1] = gainencode(gain[cbindex-1], &gindex);
```

```
// Scale selected code word vector
```

```
for (i = 0; i < 60; i++)
```

```
    v[i] = gain[cbindex - 1] * x[i + codeword];
```

```
}
```

3.18 CodebookGain [cgain.c]

This function computes the gain and match score of an input code word according to Eqns. 17 and 19. Similar to **PitchGain**, the function uses end-point correction (Eqn. 10) to reduce computation complexity. The function also makes use of the fact that the input code word contains -1 , 0 , and $+1$ only to reduce computation complexity. For example, to evaluate Eqn. 9, the function decomposes the convolution sum as follows:

$$\begin{aligned}
 y[0] &= h[0]x[0] \\
 y[1] &= h[0]x[1] + h[1]x[0] \\
 y[2] &= h[0]x[2] + h[1]x[1] + h[2]x[0] \\
 y[3] &= h[0]x[3] + h[1]x[2] + h[2]x[1] + h[3]x[0] \\
 &\vdots \\
 y[n] &= \begin{cases} h[0]x[n] + h[1]x[n-1] + h[2]x[n-2] + h[3]x[n-3] + \dots + h[n]x[0] & n \leq L' - 1 \\ h[0]x[n] + h[1]x[n-1] + h[2]x[n-2] + h[3]x[n-3] + \dots + h[L' - 1]x[n - L' + 1] & n > L' - 1 \end{cases} \\
 &\vdots \\
 y[59] &= h[0]x[59] + h[1]x[58] + h[2]x[57] + \dots + h[L' - 1]x[59 - L' + 1]
 \end{aligned} \tag{31}$$

We can see that if $L' = 60$, $h[j]x[0]$ will appear on the last item of every convolution sum, $h[j]x[1]$ will appear on the second last item of every sum, and so on. As a result, we can reduce the number of computation by accumulating the product $h[j]x[i]$ of each sum to the array $y[0..59]$. When $L' < 60$, special care must be taken in the accumulation because some of the products are truncated and do not appear in the convolution sum.

```

float CodebookGain(float ex[],           // Excitation vector from the stochastic codebook
                  int l,                // Length of ex[]. l = 60 in CELP
                  int first,           // TRUE/FALSE. For end-point correction
                  int len,             // Length of truncated impulse response of perceptual LP filter
                  float *match)        // Match score (Eqn. 19) to be returned
{
    float cor;                          // Cross-correlation, numerator of Eqn. 17
    static float y[60];
    static float y59save;                // y[58] in previous call
    static float y60save;                // y[59] in previous call
    static float eng;                    // Energy, denominator of Eqn. 17
    int i, j;

    if (first)
    {
        // For the first code word, calculate and save the convolution of the code word and the truncated
        // impulse response. A "scalar times vector accumulation" method is used to exploit
        // (skip) zero samples of the code words.
        for (i = 0; i < l; i++) y[i] = 0.0; // Prepare for the accumulation
        for (i = 0; i < l; i++) {
            if (round(ex[i]) != 0.0) // Check if ex[0]==0. Do it here saves lots of
                // checking as each ex[i] needs to be checked once only.
                for(j = 0; j < l-i && j < len; j++)
                    y[i+j] += ex[i]*h[j]; // Accumulate the last i-th items in Eqn. 31
        }
    }
}

```

```

else {
    // End-point correct the convolution sum on subsequent code words (see Eqn. 10)
    // NOTE: The data movements in the 2 loops with "y[i] = y[i - 1]" are performed many times
    // and can be quite time consuming. Therefore, special precautions should be taken
    // when implementing this. Some implementation suggestions:
    // 1. Circular buffers with pointers to eliminate data moves.
    // 2. Fast "block move" operation as offered on some DSPs.

    // First shift
    if (round(ex[1]) != 0) {
        if (round(ex[1]) == 1)
            for (i = len - 1; i > 0; i--)
                y[i - 1] += h[i]; // ex[1]=1, Eqn. 10b
        else
            for (i = len - 1; i > 0; i--)
                y[i - 1] -= h[i]; // ex[1]=-1, Eqn. 10b
    }
    for (i = 1 - 1; i > 0; i--)
        y[i] = y[i - 1]; // Eqn. 10b,c
    y[0] = ex[1] * h[0]; // Eqn. 10a

    // Second shift
    if (round(ex[0]) != 0) {
        if (round(ex[0]) == 1)
            for (i = len - 1; i > 0; i--)
                y[i - 1] += h[i]; // Eqn. 10b
        else
            for (i = len - 1; i > 0; i--)
                y[i - 1] -= h[i]; // Eqn. 10b
    }
    for (i = 1 - 1; i > 0; i--)
        y[i] = y[i - 1]; // Eqn. 10b,c
    y[0] = ex[0] * h[0]; // Eqn. 10a
}

// Calculate correlation and energy:
// e0[] = spectrum & pitch prediction residual, y = error weighting filtered code words
// Note: CELP's computations are focused here. For a 512 code book this
// correlation takes 4 MIPS
cor = 0.0;
for (i = 0; i < 1; i++)
    cor += y[i] * e0[i];

// Determine energy on subsequent code words using end-point correction
if (round(ex[0]) == 0 && round(ex[1]) == 0 && !first)
    eng = eng - y59save * y59save - y60save * y60save; // The last two samples of previous call
// do not contribute to the current eng.
else {
    eng = 0.0; // First call, or one or both of the last
    for (i = 0; i < 1; i++) // two samples have contribution to
        eng += y[i] * y[i]; // the energy of current subframe
}
y59save = y[1 - 2]; // Prepare for the next call
y60save = y[1 - 1];

```

```
if (eng <= 0.0) eng = 1.0;
    cgain = cor / eng;           // Eqn. 17
*match = cor * cgain;         // Eqn. 19
return (cgain);
}
```

3.19 PackTau [packtau.c]

This function converts a decimal value into a binary value, which is to be decoded by unpacktau.c in the CELP synthesizer section.

```

PackTau(int value,           // Index to pdecode[0...255] array for encoding
        int bits,           // Number of bits for encoding
        int pdecode[],      // Pitch delay encoding table pdecode[0...255]
        short array,        // Bit stream, containing 0 and 1. array[0...143]
        int *pointer)       // Before execution, it indexes to the beginning of encoded bit stream in array[]
                                // After execution, it indexes to the next position of array[] to be filled with
                                // encoded bit-stream
{
    int i;

    // Change index to permuted index
    value = pdecode[value];

    // Insert in bitstream
    for (i = 0; i < bits; (*pointer)++, i++)
        array[*pointer] = (value & (1 << i)) >> i;
}

```

For example, to pack the 3-rd bit of decimal value 200 (binary value 1100 1000) to bit-stream starting at position 86 (*pointer = 86), the function performs the followings:

$1 \ll 3$	\Rightarrow	0000 1000
value = 200	\Rightarrow	1100 1000
value & (1<<3)	\Rightarrow	0000 1000
(value & (1<<3)) >> 3	\Rightarrow	0000 0001
	\Rightarrow	array[86] = 1

```

pdecode[255] = {
66, 70, 71, 87, 86, 89, 88, 174, 190, 186, 184, 188, 172, 168, 148, 132, 140,
156, 158, 142, 134, 150, 10, 2, 11, 3, 27, 19, 9, 1, 25, 17, 243, 247,
231, 227, 229, 225, 241, 245, 97, 101, 117, 113, 109, 125, 105, 121, 123, 127, 107,
193, 192, 195, 194, 210, 211, 209, 208, 48, 50, 58, 49, 51, 59, 63, 55, 62,
54, 52, 74, 75, 78, 79, 95, 94, 92, 93, 84, 85, 80, 81, 170, 166, 162,
182, 178, 187, 176, 185, 180, 189, 164, 160, 169, 173, 149, 133, 157, 141, 137, 153,
136, 152, 144, 128, 154, 138, 130, 146, 26, 18, 0, 8, 6, 14, 15, 7, 23,
31, 13, 5, 29, 21, 251, 255, 235, 239, 237, 234, 238, 236, 230, 226, 228, 224,
244, 240, 96, 100, 116, 112, 115, 114, 108, 124, 104, 120, 122, 126, 106, 110, 111,
196, 197, 201, 200, 199, 203, 198, 202, 214, 218, 219, 215, 217, 213, 216, 212, 32,
40, 56, 34, 42, 57, 41, 33, 35, 43, 39, 47, 37, 45, 61, 53, 60, 46,
44, 38, 36, 73, 72, 76, 77, 68, 69, 64, 65, 167, 163, 183, 179, 177, 181,
165, 161, 151, 135, 159, 143, 129, 145, 155, 139, 131, 147, 24, 16, 4, 12, 22,
30, 20, 28, 249, 250, 253, 233, 254, 232, 252, 67, 242, 246, 248, 91, 90, 99,
98, 119, 118, 82, 83, 102, 103, 204, 205, 171, 207, 206, 222, 191, 223, 221, 220,
175}

```

3.20 Pack [pack.c]

This function converts a decimal value into a binary value, which is to be decoded by unpack.c in the CELP synthesizer section. The packing algorithm is identical to **PackTau**, except for the absence of permutation table pdencode[].

```
Pack(  int value,           // Index to pdencode[0...255] array for encoding
      int bits,           // Number of bits for encoding
      short array,       // Bit stream, containing 0 and 1. array[0...143]
      int *pointer)      // Before execution, it indexes to the beginning of encoded bit stream in array[]
                          // After execution, it indexes to the next position of array[] to be filled with
                          // encoded bit-stream
{
    int i;

    for (i = 0; i < bits; (*pointer)++, i++)
        array[*pointer] = (value & (1 << i)) >> i;
}
```

Appendix A: Code Book Search Methods

In Analysis-by-Synthesis speech coding, the speech signals within a subframe can be described by a vector $\mathbf{s} \equiv [s_0 \dots s_{L-1}]^T$ and the synthetic speech by a vector $\hat{\mathbf{s}} \equiv [\hat{s}_0 \dots \hat{s}_{L-1}]^T$. Generally, the reconstructed residual vector (the excitation vector) is constructed from an adaptive and a fixed codebook contribution, as shown in Fig. A-1(b) below. The procedures for searching these two code books are almost identical, except for the differences in the code books' structure and the target vectors. The search algorithm begins with searching the adaptive code book, followed by searching the stochastic code book. Separating the search procedure into two stages reduces computation complexity considerably.

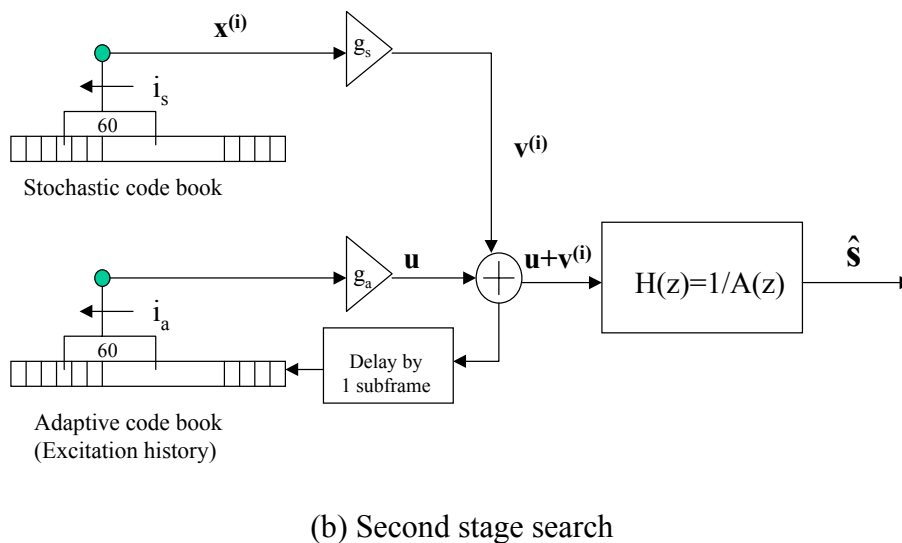
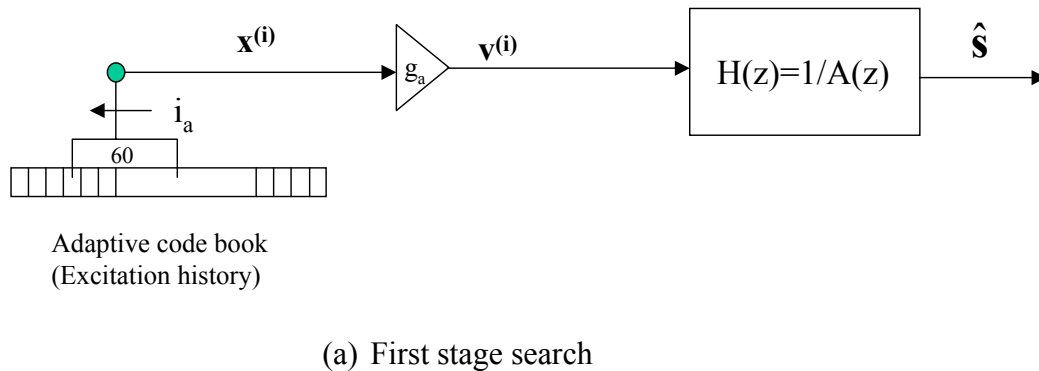


Fig. A-1. Adaptive and stochastic code book search. Note that the optimal code word $\mathbf{v}^{(m)}$ found in the first stage will become the excitation \mathbf{u} in the second stage search.

Let $\mathbf{v}^{(i)}$ represents the i -th code word of the adaptive code book (1st stage) or the i -th code word of the stochastic code book (2nd stage). The objective is to select the codebook entries that result in the smallest difference between the original speech signal and the synthetic speech. Let \mathbf{H} and \mathbf{W} be $L \times L$ lower triangular matrices whose columns contain the truncated impulse responses of the LP filter and the error weighting filter:

$$H = \begin{bmatrix} h_0 & 0 & 0 & \cdot & \cdot & \cdot & \cdot \\ h_1 & h_0 & 0 & \cdot & \cdot & \cdot & \cdot \\ h_2 & h_1 & h_0 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & h_0 & 0 \\ h_{L-1} & \cdot & \cdot & \cdot & \cdot & h_1 & h_0 \end{bmatrix} \quad W = \begin{bmatrix} w_0 & 0 & 0 & \cdot & \cdot & \cdot & \cdot \\ w_1 & w_0 & 0 & \cdot & \cdot & \cdot & \cdot \\ w_2 & w_1 & w_0 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & w_0 & 0 \\ w_{L-1} & \cdot & \cdot & \cdot & \cdot & w_1 & w_0 \end{bmatrix} \quad (\text{A-1})$$

The synthetic speech \hat{s} for the i -th adaptive codebook entry (1st stage) or for the i -th stochastic codebook entry (2nd stage) can be written as

$$\hat{s}_n^{(i)} = \sum_{k=0}^{L-1} h_k (v_{n-k}^{(i)} + u_{n-k}) \quad n = 0, \dots, L-1 \quad (\text{A-2})$$

where for the first stage $\{u_k\}_{k=-1}^{-L+1}$ are the excitation history in the adaptive codebook and $\{u_k\}_{k=0}^{L-1} = 0$, and for the second stage $\{u_k\}_{k=-1}^{-L+1}$ are the excitation history and $\{u_k\}_{k=0}^{L-1}$ are the gain scaled adaptive excitation vector found in the first stage (see Fig. 8). The excitation signal and its corresponding synthetic speech are shown in Fig. A-2 below.

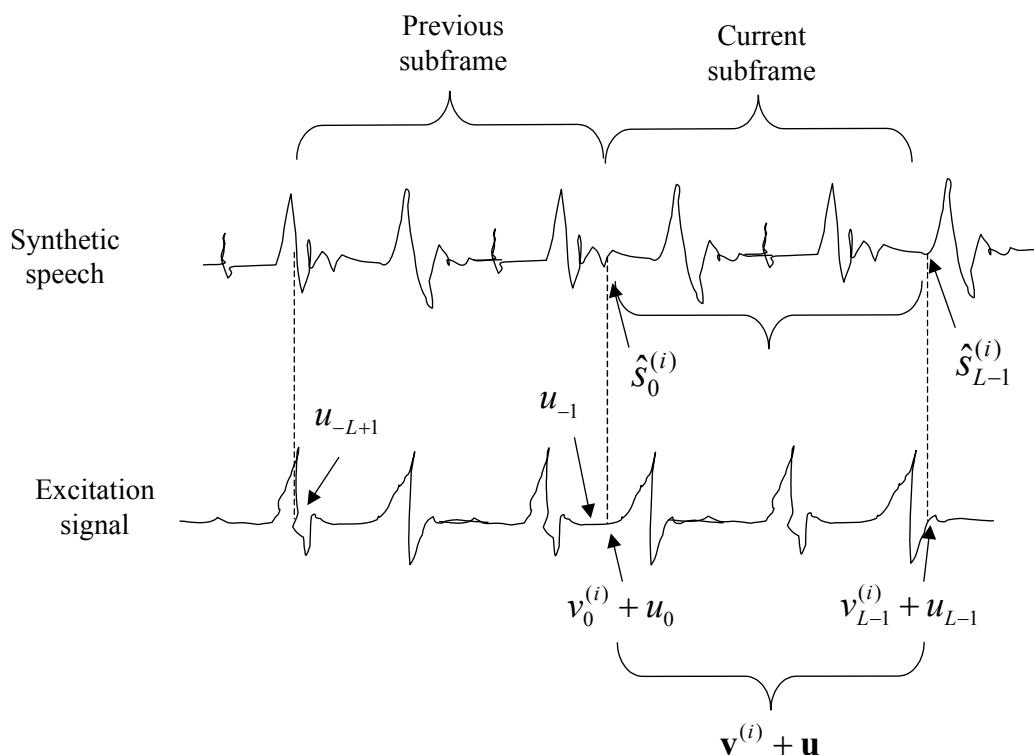


Fig. A-2. Excitation signal and synthetic speech

We can express Eqn. A-2 in matrix form

$$\begin{bmatrix} \hat{s}_0^{(i)} \\ \hat{s}_1^{(i)} \\ \vdots \\ \hat{s}_{L-1}^{(i)} \end{bmatrix} = \begin{bmatrix} h_0(v_0^{(i)} + u_0) \\ h_0(v_1^{(i)} + u_1) + h_1(v_0^{(i)} + u_0) \\ \vdots \\ h_0(v_{L-1}^{(i)} + u_{L-1}) + h_1(v_{L-2}^{(i)} + u_{L-2}) + \dots + h_{L-1}(v_0^{(i)} + u_0) \end{bmatrix} + \begin{bmatrix} h_1 u_{-1} + \dots + h_{L-1} u_{-L+1} \\ h_2 u_{-2} + \dots + h_{L-1} u_{-L+2} \\ \vdots \\ 0 \end{bmatrix} \quad (\text{A-3})$$

which is equivalent to

$$\hat{\mathbf{s}}^{(i)} = H(\mathbf{v}^{(i)} + \mathbf{u}) + \hat{\mathbf{s}}^{(0)} \quad (\text{A-4})$$

where $\hat{\mathbf{s}}^{(0)}$ is the zero-input (memory) response with

$$\hat{s}_n^{(0)} = \sum_{k=n+1}^{L-1} h_k u_{n-k} \quad n = 0, \dots, L-1 \quad (\text{A-5})$$

Using Eqn. A-4, the weighted error signal can be expressed as

$$\begin{aligned} \mathbf{e}^{(i)} &= W(\mathbf{s} - \hat{\mathbf{s}}^{(i)}) \\ &= W\mathbf{s} - WH\mathbf{u} - WH\mathbf{v}^{(i)} - W\hat{\mathbf{s}}^{(0)} \\ &= W(\mathbf{s} - \hat{\mathbf{s}}^{(0)}) - WH\mathbf{u} - WH\mathbf{v}^{(i)} \\ &= \mathbf{e}^{(0)} - WH\mathbf{v}^{(i)} \\ &= \mathbf{e}^{(0)} - WH\mathbf{g}_i \mathbf{x}^{(i)} \end{aligned} \quad (\text{A-6})$$

where $\mathbf{e}^{(0)} = W(\mathbf{s} - \hat{\mathbf{s}}^{(0)}) - WH\mathbf{u}$ is the target signal. Let $\mathbf{y}^{(i)} = WH\mathbf{x}^{(i)}$, the weighted error can be simplified to

$$\mathbf{e}^{(i)} = \mathbf{e}^{(0)} - \mathbf{g}_i \mathbf{y}^{(i)} \quad (\text{A-7})$$

Our objective is to find the values of i and \mathbf{g}_i that minimize the weighted error. This can be achieved by setting the derivative of the total squared error $E_i = \|\mathbf{e}^{(i)}\|^2$ to zero, which results in Eqns 17 and 19.

Appendix B: Zero Input Response

Let us consider an IIR filter of the form

$$H(z) = \frac{1}{A(z)} = \frac{1}{1 + \sum_{k=1}^p a_k z^{-k}} \quad (\text{B-1})$$

We denote $x[n]$ as the input and $y[n]$ as the output, where $n = 0, \dots, L-1$. The output can be expressed in terms of the LP coefficients a_k

$$y[n] = x[n] - \sum_{k=1}^p a_k y[n-k] \quad (\text{B-2})$$

or the truncated impulse response of $H(z)$

$$y[n] = \sum_{k=0}^{L-1} h[k]x[n-k] \quad (\text{B-3})$$

The zero input response (also referred to as the memory response) is defined as the output $y_z[n]$ when the inputs $x[n], n = 0, \dots, L-1$, are identically zero. More specifically,

$$y_z[n] = \sum_{k=0}^{L-1} h[k]x_z[n-k] \quad n = 0, \dots, L-1 \quad (\text{B-4})$$

where $x_z[j] = 0 \forall j = 0, \dots, L-1$ and $x_z[j] = x[j] \forall j = -1, \dots, -L+1$, i.e. the previous $L-1$ inputs.

When $n = 0$, Eqn. B-4 gives

$$y_z[0] = \sum_{k=0}^{L-1} h[k]x_z[-k] = \sum_{k=1}^{L-1} h[k]x[-k] \quad (\text{B-5})$$

When $n = 1$, we have

$$\begin{aligned} y_z[1] &= \sum_{k=0}^{L-1} h[k]x_z[1-k] = h[0]x_z[1] + h[1]x_z[0] + \sum_{k=2}^{L-1} h[k]x_z[1-k] \\ &= h[0]0 + h[1]0 + \sum_{k=2}^{L-1} h[k]x_z[1-k] = \sum_{k=2}^{L-1} h[k]x[1-k] \end{aligned} \quad (\text{B-6})$$

When $n = L-1$, we have

$$\begin{aligned} y_z[L-1] &= \sum_{k=0}^{L-1} h[k]x_z[L-1-k] \\ &= h[0]x_z[L-1] + h[1]x_z[L-2] + \dots + h[L-1]x_z[0] \\ &= 0 \end{aligned} \quad (\text{B-7})$$

Note the correspondence between Eqns. B-5 to B-7 and the last term of Eqn. A-3.

It is not necessary to compute the zero input response using Eqn. B-4. Instead, we can use Eqn. B-2 as follows:

$$y_z[n] = -\sum_{k=1}^p a_k y_z[n-k] \quad n = 0, \dots, L-1 \quad (\text{B-8})$$

where $y_z[n-1], y_z[n-2], \dots, y_z[n-p]$ are the memory of the IIR filter shown in Fig. B-1 below.

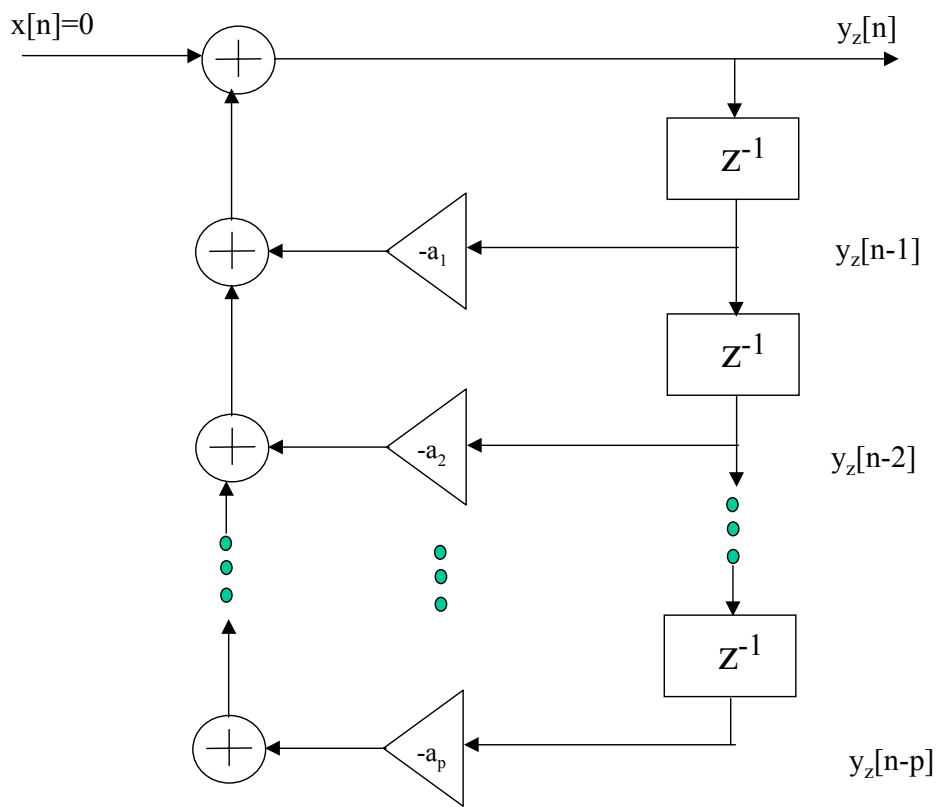


Fig. B-1 Computation of zero input response