INTERNATIONAL TELECOMMUNICATION UNION

ITU-T

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

G.711

Appendix I

(09/99)

SERIES G: TRANSMISSION SYSTEMS AND MEDIA, DIGITAL SYSTEMS AND NETWORKS

Digital transmission systems – Terminal equipments – Coding of analogue signals by pulse code modulation

Pulse code modulation (PCM) of voice frequencies

Appendix I: A high quality low-complexity algorithm for packet loss concealment with G.711

ITU-T Recommendation G.711 – Appendix I

(Previously CCITT Recommendation)

ITU-T G-SERIES RECOMMENDATIONS

TRANSMISSION SYSTEMS AND MEDIA, DIGITAL SYSTEMS AND NETWORKS

INTERNATIONAL TELEPHONE CONNECTIONS AND CIRCUITS	G.100–G.199
INTERNATIONAL ANALOGUE CARRIER SYSTEM	
GENERAL CHARACTERISTICS COMMON TO ALL ANALOGUE CARRIER-TRANSMISSION SYSTEMS	G.200–G.299
INDIVIDUAL CHARACTERISTICS OF INTERNATIONAL CARRIER TELEPHONE SYSTEMS ON METALLIC LINES	G.300–G.399
GENERAL CHARACTERISTICS OF INTERNATIONAL CARRIER TELEPHONE SYSTEMS ON RADIO-RELAY OR SATELLITE LINKS AND INTERCONNECTION WITH METALLIC LINES	G.400–G.449
COORDINATION OF RADIOTELEPHONY AND LINE TELEPHONY	G.450–G.499
TESTING EQUIPMENTS	
TRANSMISSION MEDIA CHARACTERISTICS	
DIGITAL TRANSMISSION SYSTEMS	
TERMINAL EQUIPMENTS	G.700–G.799
General	G.700–G.709
Coding of analogue signals by pulse code modulation	**G.710–G.719**
Coding of analogue signals by methods other than PCM	G.720–G.729
Principal characteristics of primary multiplex equipment	G.730–G.739
Principal characteristics of second order multiplex equipment	G.740–G.749
Principal characteristics of higher order multiplex equipment	G.750–G.759
Principal characteristics of transcoder and digital multiplication equipment	G.760–G.769
Operations, administration and maintenance features of transmission equipment	G.770–G.779
Principal characteristics of multiplexing equipment for the synchronous digital hierarchy	G.780–G.789
Other terminal equipment	G.790–G.799
DIGITAL NETWORKS	G.800–G.899
DIGITAL SECTIONS AND DIGITAL LINE SYSTEM	G.900–G.999

For further details, please refer to ITU-T List of Recommendations.

ITU-T RECOMMENDATION G.711

PULSE CODE MODULATION (PCM) OF VOICE FREQUENCIES

APPENDIX I

A high quality low-complexity algorithm for packet loss concealment with G.711

Summary

Packet Loss Concealment (PLC) algorithms, also known as frame erasure concealment algorithms, hide transmission losses in an audio system where the input signal is encoded and packetized at a transmitter, sent over a network, and received at a receiver that decodes the packet and plays out the output. Many of the standard CELP-based speech coders have PLC algorithms built into their standards. The algorithm described here provides a method for Recommendation G.711.

Source

Appendix I to ITU-T Recommendation G.711 was prepared by ITU-T Study Group 16 (1997-2000) and was approved under the WTSC Resolution No. 1 procedure on 30 September 1999.

CONTENTS

Recommendation G.711

PULSE CODE MODULATION (PCM) OF VOICE FREQUENCIES

APPENDIX I

A high quality low-complexity algorithm for packet loss concealment with G.711

(Geneva, 1999)

I.1 Introduction

Packet Loss Concealment (PLC) algorithms, also known as frame erasure concealment algorithms, hide transmission losses in an audio system where the input signal is encoded and packetized at a transmitter, sent over a network, and received at a receiver that decodes the packet and plays out the output. Many of the standard CELP-based speech coders, such as Recommendations G.723.1 [1], G.728 [2] and G.729 [3], have PLC algorithms built into their standards. The algorithm described here provides a method for Recommendation G.711.

The objective of PLC is to generate a synthetic speech signal to cover missing data (erasures) in a received bit stream. Ideally, the synthesized signal will have the same timbre and spectral characteristics as the missing signal, and will not create unnatural artifacts. Since speech signals are often locally stationary, it is possible to use the signals' past history to generate a reasonable approximation to the missing segment. If the erasures are not too long, and the erasure does not land in a region where the signal is rapidly changing, the erasures may be inaudible after concealment.

I.2 Algorithm description

To add PLC to a G.711 system that currently does not conceal losses, changes are only required in the receiver. The G.711-encoded audio data is sampled at 8 kHz. In this appendix it is assumed to be partitioned into 10 ms frames (80 samples). By adjusting a few parameters, other packet sizes or sampling rates can be accommodated.

I.2.1 Good frames

During normal operation (good packets or frames) the receiver decodes the received packet and sends its output to the audio port. Two minor changes are made to the receiver when it processes good frames to support PLC.

1) A copy of the decoded output is saved in a circular history buffer that is 48.75 ms (390 samples) long. The history buffer is used to calculate the current pitch period and extract waveforms during an erasure. This buffering does not introduce any delay into the output signal.

2) The output is delayed by 3.75 ms (30 samples) before it sent to the audio port. This algorithm delay, used for an Overlap Add (OLA) at the start of an erasure, allows the PLC code to make a smooth transition between the real and synthesized signal.

I.2.2 First bad frame

At the start of the erasure, the circular history buffer is copied to a non-circular buffer, called the pitch buffer, that is easier to work with. The contents of the pitch buffer are used for the duration of the erasure. An additional copy of the most recent 1/4 pitch period, called the `lastq` buffer, is made in case the erasure lasts longer than 10 ms.

I.2.3 Pitch detection

First, the pitch period is estimated by finding the peak of the normalized cross-correlation of the most recent 20 ms of speech in the history buffer with the previous speech at taps from 5 (40 samples) to 15 ms (120 samples). This corresponds to frequencies of 200 to 66 Hz. The pitch range was chosen based on a range used in G.728's post-filter. While G.728 uses a lower bound of 2.5 ms (20 samples), here it is increased to 40 samples so the same pitch period is not repeated more than twice in a single 10 ms erased frame. To lower complexity, the pitch estimation is calculated in two phases. First, a coarse search is performed on a 2:1 decimated signal, and then a finer search is performed in the vicinity of the peak of the coarse search. The complexity can be lowered with a slight degradation in quality by skipping the fine search. In the following the term *wavelength* is also used to refer to the output value of this calculation, since the missing signal may be either voiced or unvoiced speech.

From Waveform Shift Overlap Add (WSOLA), it is known that the normalized cross-correlation function can be replaced with either a non-normalized cross correlation, or a cross-Average Magnitude Difference Function (AMDF) and similar overall performance results will be obtained.

I.2.4 Synthetic signal generation for first 10 ms

For the first 10 ms of the erasure, the best results are obtained by generating the synthesized signal from the last pitch period with no attenuation. Only the most recent 1.25 pitch periods of the pitch buffer are used during the first 10 ms. To insure a smooth transition between the real and synthetic signal, and a smooth transition if the pitch period is repeated multiple times, an Overlap Add (OLA) is performed using a triangular window on 1/4 of the pitch period between the last and next to last pitch period. For 1/4 wavelength the signal starting at 1.25 pitch periods from the end of the pitch buffer is multiplied by an up-sloping ramp and is added to the last 0.25 pitch period in the `lastq` buffer multiplied by a down-sloping ramp. If complexity is not an issue, the triangular windows may be replaced with Hanning windows in all the OLA operations.

The result of the OLA replaces both the tail of the pitch buffer and the tail of the history buffer. It is also output by the receiver during the tail of the last good frame, replacing the original signal. This introduces the algorithm delay – the tail of the last frame cannot be output until it is known whether the next frame is erased. If an erasure occurs the signal in the tail of the last good frame is modified by the OLA to insure a smooth transition to the synthesized signal.

The synthesized signal for the 10 ms during the erasure is generated by placing a pointer one pitch period back from the end of the pitch buffer, and copying the samples to the output. If the pitch period is shorter than 10 ms, when the pointer rolls off the end of the pitch buffer the pointer is set back exactly one pitch period before continuing. If the pitch period is short (the frequency is high), the last pitch period in the pitch buffer is repeated multiple times during the 10 ms erasure.

While the erasure progresses, the history buffer is updated with the synthesized output. This way, the history buffer always has a smooth, continuous signal in it. This continuity is important if a "bad frame, good frame, bad frame" sequence occurs.

I.2.5 Synthetic signal generation after 10 ms

If the next frame is also erased, the erasure will be at least 20 ms long and further action is required. While repeating a single pitch period works well for short erasures (e.g. 10 ms), on long erasures it introduces unnatural harmonic artifacts (beeps). This is especially noticeable if the erasure lands in an unvoiced region of speech, or in a region of rapid transition such as a stop. It was discovered by experimentation that these artifacts are significantly reduced by increasing the number of pitch periods used to synthesize the signal as the erasure progresses. Playing more pitch periods increases the variation in the signal. Although the pitch periods are not played in the order they occurred in the original signal, the resulting output still sounds natural. At 10 ms into the erasure the number of pitch periods used to synthesize the speech is increased to two, and at 20 ms a third pitch period is added. For erasures longer than 20 ms no additional modifications to the pitch buffer are made.

When the number of pitch periods used in the pitch buffer increases, it is important that the transition in the synthesized signal be smooth. This is accomplished by continuing the output of the existing pitch buffer for 1/4 of a pitch period at the start of the second and third erased frame, updating the pitch buffer, keeping the buffer pointer synchronized with the correct phase, and then doing an OLA with the output from the new pitch buffer.

The pitch buffer is updated exactly as during the first erased frame, except that the number of pitch periods is increased. For example, at the start of the second erased frame, for 1/4 wavelength the signal starting at 2.25 pitch periods from the end of the pitch buffer is multiplied by an up-sloping ramp and is added to the 1/4 wavelength in the `lastq` buffer multiplied by a down-sloping ramp. The result of the OLA replaces the last 1/4 wavelength in the pitch buffer. To maintain the phase of the current output pointer, pitch periods are subtracted from the pointer until it is in the first pitch period used.

I.2.6 Attenuation

As with other PLC algorithms, such as G.729 and G.728 Annex I, with long erasures it is necessary to attenuate the signal as the erasure progresses. As the erasure gets longer, the synthesized signal is more likely to diverge from the real signal. Without attenuation strange artifacts are created by holding certain types of sounds too long, even if the synthesized signal segment sounds natural in isolation. For the first 10 ms of an erasure the signal is not attenuated. At the start of the second 10 ms, the synthesized signal is linearly attenuated with a ramp at the rate of 20% per 10 ms. After 60 ms, the synthesized signal is zero.

I.2.7 First good frame after an erasure

At the first good frame after an erasure, a smooth transition is needed between the synthesized erasure speech and the real signal. To do this, the synthesized speech from the pitch buffer is continued beyond the end of the erasure, and then mixed with the real signal using an OLA. The length of the OLA depends on both the pitch period and the length of the erasure. For short, 10 ms erasures, a 1/4 wavelength window is used. For longer erasures the window is increased by 4 ms per 10 ms of erasure, up to a maximum of the frame size, 10 ms.

I.2.8 Example

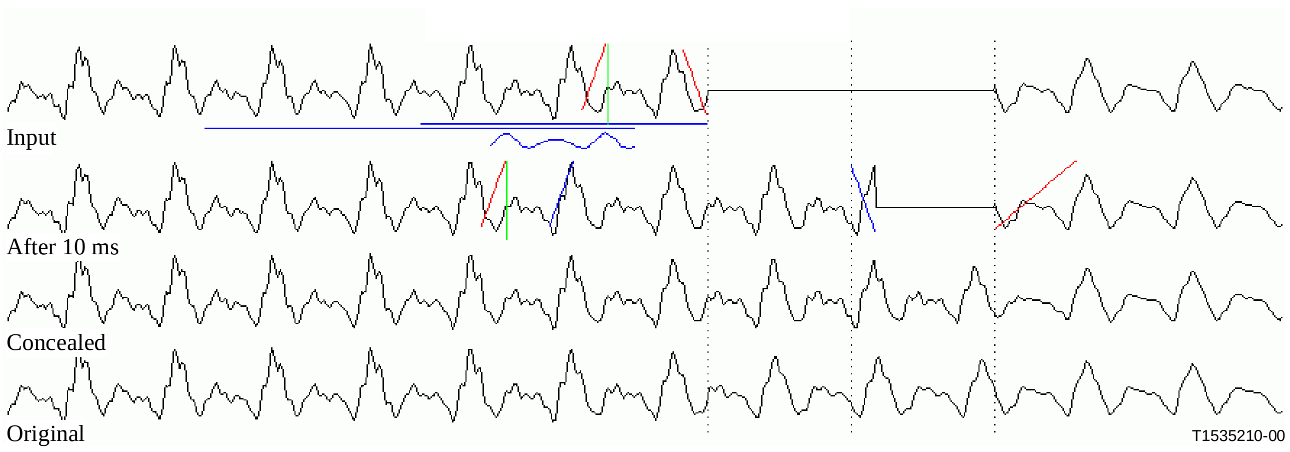

Figure I.1/G.711 – Frame erasure concealment algorithm for G.711

Figure I.1 shows a graphical example of how the algorithm operates with a 20 ms (i.e. two frames) erasure on a voiced segment of speech from a male speaker. The top waveform shows the input. The location of the erasure is delimited by the three dotted vertical lines at 10 ms intervals that cross all the waveforms. The speech before the erasure on the "Input" graph is the contents of the history buffer when the erasure begins. The speech after the erasure is 20 ms of uncorrupted speech that arrives after the erasure is over.

Below the "Input" signal are two horizontal lines that represent the windows used for pitch detection with the normalized cross-correlation. The short upper line shows the last 20 ms of speech before the erasure and is the reference signal. The lower line represents the 20 ms window that slides back at taps from 40 to 120 samples. The small graph below the horizontal lines is the output of the normalized cross-correlation. The peak of this graph shown by the vertical line is the pitch estimate. This vertical line is located one pitch period back from the start of the erasure.

The contents of the 1/4 wavelength before the erasure are saved in the `lastq` buffer for later use. To create the pitch buffer, the 1/4 wavelength from before the erasure is OLAed with a triangular window to the 1/4 wavelength from the previous pitch period. The weights and locations of the windows are shown by the two ramps in the "Input" signal. The results of this OLA replace the 1/4 wavelength of signal before the erasure. For the first 10 ms of the erasure, the synthetic signal is generated by repeating the single period pitch buffer, e.g. the region starting at the vertical line at the peak of the cross correlation and ending at the start of the erasure, as many times as needed.

The results of this are shown in the second waveform, "After 10 ms". In this waveform, the synthetic signal is extended 1/4 wavelength beyond the first 10 ms erasure boundary as this is needed for the next OLA. The offset into the single period pitch buffer at the end of 10 ms is saved in a variable called `poffset`. If the erasure ends after 10 ms, this 1/4 wavelength would just be OLAed with the input signal and the concealment algorithm is finished. Here the erasure is longer than 10 ms so the pitch buffer is lengthened to increase the signal variation.

The signal variation is increased by adding another pitch period to the pitch buffer. A vertical line has been drawn on the waveform two pitch periods back from the start of the erasure. The 1/4 wavelength before this line, shown by the up-sloping ramp, is OLAed with `lastq` buffer (the region under the down-sloping ramp in the "Input" waveform) and placed in the pitch buffer. For the second 10 ms the pitch buffer is thus the region between end of the first up-sloping ramp and the start of the erasure in the "After 10 ms" waveform. The 1/4 wavelength right before the erasure is different in the pitch buffer from what is shown in the waveform, since it is the result of an OLA.

To insure a smooth transition between the single and double period pitch buffers, an OLA is performed for 1/4 wavelength at the start of the second 10 ms in the erasure. The region under the down-sloping ramp at 10 ms into the erasure is combined with the region under the up-sloping ramp near the first signal peak in the two period pitch buffer with an OLA. The result replaces the region under the down-sloping ramp. The location of the up-sloping ramp is calculated from the saved `poffset` pointer by subtracting pitch periods until the pitch pointer is in the first wavelength of the currently used portion of the pitch buffer. This insures that the proper waveform phase is maintained as the number of periods in the pitch buffer is increased. For the duration of the second 10 ms erased frame, the synthetic waveform is generated by simply copying the signal from the pitch buffer.

As during the first 10 ms, the synthetic waveform is extended beyond the end of next 10 ms boundary for an OLA with the next segment. If the erasure continues, a 1/4 wavelength suffices, and the number of periods in the pitch buffer is increased again. If the erasure ends, the OLA window length is increased by 4 ms per additional 10 ms of erasure (up to a maximum of 10 ms) as phase mismatches between the synthetic and real signal are more likely. The OLA window for the end of the 20 ms erasure (1/4 wavelength + 4 ms) is shown as the up-sloping ramp at the at the end of the erasure in the "After 10 ms" waveform. During the second 10 ms of an erasure the waveform is also attenuated with a linear ramp. For long erasures, this attenuation drives the synthetic signal to 0 after 60 ms.

The "Concealed" waveform shows the final output from the algorithm. For comparison purposes the original waveform without any erasures, is also shown below it. The synthetic speech closely resembles the speech before the erasure, and is a good approximation to the original waveform. Close inspection of the synthetic waveform reveals that the first pitch period in the erasure comes from the last period before the erasure, the second period in the erasure comes from 2 periods before the erasure, and the third period in the erasure repeats the first period before the erasure. Also, due to a pitch change during the erasure the peak of the last period in the synthetic signal does not align exactly with the last pitch in the original signal. This is why the OLA window at the tail of the erasure must be widened, as the erasure gets longer.

I.3 Algorithm description with annotated C++ code

The PLC algorithm has been implemented in floating-point with a C++ class[1]. The code, along with an explanation of how it works is presented in this subclause. The complete implementation is about 360 line of C++ code, including comments. The C++ code contained in this annex is included for illustrative purposes only.

I.3.1 Typedefs and constants

To allow switching between double-precision and single-precision floating-point arithmetic, the following type is defined.

```
typedef float Float;
```

To switch to double-precision mode, change the "float" to a "double". The code defines the following preprocessor constants:

```
#define PITCH_MIN       40              /* minimum allowed pitch, 200 Hz */
#define PITCH_MAX       120             /* maximum allowed pitch, 66 Hz */
#define PITCHDIFF       (PITCH_MAX - PITCH_MIN)
#define POVERLAPMAX     (PITCH_MAX >> 2)/* maximum pitch OLA window */
#define HISTORYLEN      (PITCH_MAX * 3 + POVERLAPMAX) /* history buffer length*/
#define NDEC            2               /* 2:1 decimation */
#define CORRLEN         160             /* 20 ms correlation length */
#define CORRBUFLEN      (CORRLEN + PITCH_MAX) /* correlation buffer length */
#define CORRMINPOWER    ((Float)250.)   /* minimum power */
#define EOVERLAPINCR    32              /* end OLA increment per frame, 4 ms */
#define FRAMESZ         80              /* 10 ms at 8 KHz */
#define ATTENFAC((Float).2)     /* attenuation factor per 10 ms frame */
#define ATTENINCR       (ATTENFAC/FRAMESZ) /* attenuation per sample */
```

I.3.2 Class declaration

```
class LowcFE {
public:
             LowcFE();
     void    dofe(short *s);         /* synthesize speech for erasure */
     void    addtohistory(short *s); /* add a good frame to history buffer */
```

[1] Alternatively, an ANSI-C code is available in the G.711 module of the ITU-T Software Tools Library (ITU-T Rec. G.191).

```
 6 protected:
 7      int     erasecnt;              /* consecutive erased frames */
 8      int     poverlap;              /* overlap based on pitch */
 9      int     poffset;               /* offset into pitch period */
10      int     pitch;                 /* pitch estimate */
11      int     pitchblen;             /* current pitch buffer length */
12      Float   *pitchbufend;          /* end of pitch buffer */
13      Float   *pitchbufstart;        /* start of pitch buffer */
14      Float   pitchbuf[HISTORYLEN];  /* buffer for cycles of speech */
15      Float   lastq[POVERLAPMAX];    /* saved last quarter wavelength */
16      short   history[HISTORYLEN];   /* history buffer */
17
18      void    scalespeech(short *out);
19      void    getfespeech(short *out, int sz);
20      void    savespeech(short *s);
21      int     findpitch();
22      void    overlapadd(Float *l, Float *r, Float *o, int cnt);
23      void    overlapadd(short *l, short *r, short *o, int cnt);
24      void    overlapaddatend(short *s, short *f, int cnt);
25      void    convertsf(short *f, Float *t, int cnt);
26      void    convertfs(Float *f, short *t, int cnt);
27      void    copyf(Float *f, Float *t, int cnt);
28      void    copys(short *f, short *t, int cnt);
29      void    zeros(short *s, int cnt);
30 };
```

The class is called `LowcFE`, for Low Complexity Frame Erasure concealment. It's interface is defined by three public functions. The constructor on line 3 initializes the internal variables. The `addtohistory()` function on line 5 handles good frames. Its argument, a pointer to an array of shorts of length FRAMESZ, should contain one frame of data that has been decoded. The code assumes that a short contains 16-bit signed data and the output of the decoder is uniform 16-bit PCM data. The `dofe()` function generates the synthetic speech during an erasure.

The rest of the class is protected, meaning its members and functions cannot be accessed by the user of the class. The variable `erasecnt` tracks the number of consecutive erased frames during an erasure. Its value starts at 0 and is incremented by 1 every 10 ms during an erasure and is reset to 0 at the first good frame after an erasure.

The variable `poverlap` is the length of the OLA window, and corresponds to 1/4 of the current pitch. The variable `poffset`, on line 9, is the offset into the current pitch buffer and is used for synthetic waveform generation. The variable `pitch`, on line 10, is the current pitch estimate. The variable `pitchblen` is the length of the portion of the pitch buffer that is currently in use. This length changes if the erasure lasts more than one frame. The variable `pitchbufend` points to the end of the pitch buffer. The variable `pitchbufstart` is a pointer to the start of the currently used portion of the pitch buffer. The variable `pitchbuf`, on line 14, contains the history buffer at the start of an erasure. Other than the last 1/4 wavelength, this buffer remains static for the duration of the erasure. The `lastq` buffer contains a cached copy of the last quarter wavelength of signal before the erasure begins and is used in OLA operations. The `history` buffer on line 16 contains the recent history of the signal. It is updated during good and erased frames.

The protected functions on lines 18 to 29 implement the core of the algorithm and are described later.

I.3.3 Main loop

The main processing loop of a program that uses the `LowcFE` class with G.711 is shown below. It is assumed that the `receiveframe()` function returns true if a 10 ms G.711 bit-stream frame has been received without errors and false if an erasure occurs. The function `g711dec()` converts the G.711 bitstream into 16-bit uniform PCM and `output()` outputs the audio with the frame erasures concealed. The implementation of these functions is beyond the scope of this appendix and is not presented here.

```
 1 void process()
 2 {
 3      char    bitstream[FRAMESZ];
 4      short   speech[FRAMESZ];
 5      LowcFE  fec;
 6      bool    frameisgood;
 7
 8      for(;;) {
 9              frameisgood = receiveframe(bitstream);
10              if (frameisgood) {
11                      g711dec(bitstream, speech);
12                      fec.addtohistory(speech);
13              } else
14                      fec.dofe(speech);
15              output(speech);
16      }
17 }
```

On line 10, a test is made to see if the input frame is erased. If the frame is good it is sent to the decoder on line 11, and the output of the decoder is given to the PLC algorithm with the member function `addtohistory()`. If the frame is erased the `dofe()` member function is called to generate the synthetic speech.

It should be noted the `addtohistory` function on line 12 does more than just copy the frame of speech to the history buffer. It also modifies the contents of the speech array. The output signal is delayed by `POVERLAPMAX` samples. In addition, on the first good frame after an erasure the input signal is OLAed with the synthesized speech before returning.

I.3.4 Utility member functions

The utility functions on lines 25 to 29 of the class declaration simply convert shorts to Floats `(convertsf)` and vice-versa `(convertfs())`, copy Float `(copyf())` and short `(copys())` arrays, and zero a short array `(zeros())`. They are presented first as they are used by the other routines.

```
 1 void LowcFE::convertsf(short *f, Float *t, int cnt)
 2 {
 3      for (int i = 0; i < cnt; i++)
```

```
                t[i] = (Float)f[i];
}

void LowcFE::convertfs(Float *f, short *t, int cnt)
{
     for (int i = 0; i < cnt; i++)
                t[i] = (short)f[i];
}

void LowcFE::copyf(Float *f, Float *t, int cnt)
{
     for (int i = 0; i < cnt; i++)
                t[i] = f[i];
}

void LowcFE::copys(short *f, short *t, int cnt)
{
     for (int i = 0; i < cnt; i++)
                t[i] = f[i];
}

void LowcFE::zeros(short *s, int cnt)
{
     for (int i = 0; i < cnt; i++)
                s[i] = 0;
}
```

There is no saturation or rounding in the `convertfs` routine.

I.3.5 Constructor

The constructor initializes the internal members of the class.

```
 1 LowcFE::LowcFE()
 2 {
 3      erasecnt = 0;
 4      pitchbufend = &pitchbuf[HISTORYLEN];
5       zeros(history, HISTORYLEN);
6 }
```

On line 3, the `erasecnt` is set to 0 so the code knows it is not currently in an erasure. Next, `pitchbufend` is set to point to the end of the pitch buffer. Then the history buffer is cleared so artifacts will not occur if an erasure occurs at the start of the signal.

I.3.6 Addtohistory and savespeech

Next we present the public interface function `addtohistory()`, along with a protected function, `savespeech`, called internally by `addtohistory()`. `addtohistory()` is called by the application after decoding a good frame, but before the signal is output.

```
/*
 * Save a frames worth of new speech in the history buffer.
 * Return the output speech delayed by POVERLAPMAX.
 */
void LowcFE::savespeech(short *s)
{
     /* make room for new signal */
     copys(&history[FRAMESZ], history, HISTORYLEN - FRAMESZ);
     /* copy in the new frame */
     copys(s, &history[HISTORYLEN - FRAMESZ], FRAMESZ);
     /* copy out the delayed frame */
     copys(&history[HISTORYLEN - FRAMESZ - POVERLAPMAX], s, FRAMESZ);
}

/*
 * A good frame was received and decoded.
 * If right after an erasure, do an overlap add with the synthetic signal.
 * Add the frame to history buffer.
 */
void LowcFE::addtohistory(short *s)
{
     if (erasecnt) {
             short overlapbuf[FRAMESZ];
             /*
              * longer erasures require longer overlaps
              * to smooth the transition between the synthetic
              * and real signal.
              */
             int olen = poverlap + (erasecnt - 1) * EOVERLAPINCR;
             if (olen > FRAMESZ)
                     olen = FRAMESZ;
             getfespeech(overlapbuf, olen);
             overlapaddatend(s, overlapbuf, olen);
             erasecnt = 0;
     }
     savespeech(s);
}
```

On line 22, addtohistory() checks to see if this is the first good frame after an erasure. If an erasure occurred in the previous frame, erasecnt will contain the number of 10 ms frames in the erasure. If the previous frame was not erased, erasecnt is 0.

If the previous frame was erased, lines 23-34 continue the synthetic signal generation, OLA the synthetic signal with the input signal, and then clear erasecnt. Line 29 determines the length of OLA window. The variable poverlap contains the number of samples in 1/4 of the current estimated pitch period. If the erasure is only 10 ms (erasecnt == 1), the OLA window length is set to poverlap. If multiple frames were erased, the length of the OLA window is increased by 4 ms for every 10 ms of erasure. Lines 30-31 insure the OLA window does not exceed 10 ms for long erasures. Line 32 generates the synthetic speech in the temporary buffer, overlapbuf. On line 33 the synthetic signal is OLAed with the input signal. The output is placed back in s, replacing the original input signal.

Line 36 calls savespeech() to save the signal in the history buffer and add the algorithm delay. If this is not the first frame after an erasure, the original speech is saved. Otherwise, it is the result of the OLA on line 33.

Function savespeech(), on lines 5 to 13, saves the speech in the history buffer. On a DSP this could be a circular buffer, but for simulation purposes it is simpler to shift the contents. Line 8 shifts the buffer to make room for the new frame. The new frame is copied to the tail of the buffer on line 10, and line 12 replaces the contents of the input array with the signal delayed by POVERLAPMAX (30) samples. This introduces the algorithm delay of 3.75 ms.

I.3.7 Dofe

The public member function dofe generates the synthetic signal during an erasure and contains the bulk of the PLC algorithm.

```
 1 /*
 2  * Generate the synthetic signal.
 3  * At the beginning of an erasure determine the pitch, and extract
 4  * one pitch period from the tail of the signal. Do an OLA for 1/4
 5  * of the pitch to smooth the signal. Then repeat the extracted signal
 6  * for the length of the erasure. If the erasure continues for more than
 7  * 10 ms, increase the number of periods in the pitchbuffer. At the end
 8  * of an erasure, do an OLA with the start of the first good frame.
 9  * The gain decays as the erasure gets longer.
10  */
11 void LowcFE::dofe(short *out)
12 {
13      if (erasecnt == 0) {
14              convertsf(history, pitchbuf, HISTORYLEN); /* get history */
15              pitch = findpitch();            /* find pitch */
16              poverlap = pitch >> 2;          /* OLA 1/4 wavelength */
17              /* save original last poverlap samples */
18              copyf(pitchbufend - poverlap, lastq, poverlap);
19              poffset = 0;            /* create pitch buffer with 1 period */
20              pitchblen = pitch;
21              pitchbufstart = pitchbufend - pitchblen;
22              overlapadd(lastq, pitchbufstart - poverlap,
23                      pitchbufend - poverlap, poverlap);
24              /* update last 1/4 wavelength in history buffer */
25              convertfs(pitchbufend - poverlap, &history[HISTORYLEN-poverlap],
26                      poverlap);
27              getfespeech(out, FRAMESZ);      /* get synthesized speech */
28      } else if (erasecnt == 1 || erasecnt == 2) {
29              /* tail of previous pitch estimate */
30              short tmp[POVERLAPMAX];
31              int saveoffset = poffset;       /* save offset for OLA */
32              getfespeech(tmp, poverlap);     /* continue with old pitchbuf */
33              /* add periods to the pitch buffer */
34              poffset = saveoffset;
35              while (poffset > pitch)
36                      poffset -= pitch;
37              pitchblen += pitch;             /* add a period */
38              pitchbufstart = pitchbufend - pitchblen;
39              overlapadd(lastq, pitchbufstart - poverlap,
40                      pitchbufend - poverlap, poverlap);
41              /* overlap add old pitchbuffer with new */
42              getfespeech(out, FRAMESZ);
43              overlapadd(tmp, out, out, poverlap);
44              scalespeech(out);
45      } else if (erasecnt > 5) {
46              zeros(out, FRAMESZ);
47      } else {
48              getfespeech(out, FRAMESZ);
49              scalespeech(out);
50      }
51      erasecnt++;
52      savespeech(out);
53 }
```

On line 13 `erasecnt` is tested to see if it is 0. If true, this is the first frame of an erasure and the code on lines 14 to 27 is executed. Line 14 copies the contents of the history buffer to the pitch buffer, converting it to Floats in the process. Except for the last 1/4 wavelength, the contents of the pitch buffer do not change for the duration of the erasure. Line 15 calls `findpitch` to perform a normalized cross correlation that estimates the pitch. The function `findpitch()` returns a value between `MIN_PITCH`(40) and `MAX_PITCH`(120). `findpitch()` dominates the complexity of the algorithm and is only called on the first frame of an erasure. Line 16 sets the OLA window length to 1/4 of the pitch period. On line 18 the last 1/4 wavelength of the pitch buffer is saved in `lastq`, in case the erasure lasts more than one frame. Lines 19-21 then set up the pitch buffer so only the last period in the buffer is used to generate the synthetic speech. On line 22, the contents of `lastq` are OLAed with the 1/4 period starting 1.25 periods back in the pitch buffer. This insures a smooth transition between the original speech and the synthetic speech at the start of erasure, as well as insuring a smooth transition if the single period pitch buffer is repeated during the first frame, e.g. the period is less than 80 samples (10 ms).

The result of this OLA is placed in the tail of the pitch buffer, and replaces the last 1/4 period in the history buffer on line 25. Updating the history buffer insures the OLAed speech is output when `savespeech` is called on line 52, and that the history buffer does not contain any discontinuities. Line 27 then generates the synthetic speech for the frame by calling `getfespeech()`.The function `getfespeech()`simply copies the last period in the pitch buffer to the output array, repeating the period as many times as needed to fill the 80 samples. After line 27, the code jumps to line 51 which increments `erasecnt`. Then line 52 updates the history buffer with the synthetic speech. The function `savespeech()` also delays the signal, so when `savespeech()` returns the result of the OLA in line 22 will be present in the first 3.75 ms of out.

On the second and third frames of an erasure, the branch on lines 29-44 is taken. This branch increases the number of periods in the pitch buffer used to synthesize the pitch. Increasing the number of pitch periods increases the variation in the signal, which in turn decreases the number of unnatural harmonic artifacts (beeps) that occur if only a single pitch period is used. On lines 30-32 the output of the pitch buffer used in the previous erased frame is continued for 1/4 period and placed in a temporary buffer. This signal is OLAed with the new expanded pitch period buffer to insure a smooth transition in the output signal as the number of pitch periods in the pitch buffer is increased.

On line 31 the offset into the pitch period buffer is saved in `saveoffset`. After the old pitch buffer generates the waveform on line 32, `poffset` is restored to `saveoffset` on line 34. This insures the phase of the synthetic signal is maintained. Lines 35-36 subtract pitch periods from `poffset` until it points into the first period.

Line 37 adds a period to the pitch buffer. Line 38 updates `pitchbufstart`, the pointer to the start of the used portion of the pitch buffer. Then line 39 does an OLA between the saved data in `lastq` and the 1/4 period before `pitchbufstart`, placing the results in the last 1/4 period at the tail of the pitch buffer. As in the first frame, this insures the pitch buffer is smooth if it is repeated multiple times in the output signal.

On line 42 the synthetic signal from the new pitch buffer is extracted for the duration of the frame, and the 1/4 period at the start of this data is OLAed with the temporary buffer created on line 32. On line 44, the synthetic signal is attenuated with a linear ramp by a call to `scalespeech`. This attenuation is at the rate of 20% per 10 ms and starts at the beginning of the second erased frame. Note that the synthetic signal is not attenuated during the first frame of an erasure.

During the fourth, fifth and sixth frames of an erasure the processing is simpler since the pitch buffer remains static, as shown on lines 48-49. The synthetic signal is generated with a call to `getfespeech()`, and then attenuated with `scalespeech()`. Beyond 60 ms, the synthetic signal is set to 0 on line 46.

I.3.8 Pitch detection

The pitch detector is the only part of the algorithm that has significant complexity. To keep the complexity low, a coarse search is first performed on a decimated signal. Then a fine search is performed near the peak of the coarse search. The last 20 ms of signal is cross-correlated with the earlier signal at lags from PITCH_MIN to PITCH_MAX.

```
1 /*
2  * Estimate the pitch.
 3  * l - pointer to first sample in last 20 ms of speech.
 4  * r - points to the sample PITCH_MAX before l
 5  */
```

```
int LowcFE::findpitch()
{
     int     i, j, k;
     int     bestmatch;
     Float   bestcorr;
     Float   corr;           /* correlation */
     Float   energy;         /* running energy */
     Float   scale;          /* scale correlation by average power */
     Float   *rp;            /* segment to match */
     Float   *l = pitchbufend - CORRLEN;
     Float   *r = pitchbufend - CORRBUFLEN;

     /* coarse search */
     rp = r;
     energy = 0.f;
     corr = 0.f;
     for (i = 0; i < CORRLEN; i += NDEC) {
             energy += rp[i] * rp[i];
             corr += rp[i] * l[i];
     }
     scale = energy;
     if (scale < CORRMINPOWER)
             scale = CORRMINPOWER;
     corr = corr / (Float)sqrt(scale);
     bestcorr = corr;
     bestmatch = 0;
     for (j = NDEC; j <= PITCHDIFF; j += NDEC) {
             energy -= rp[0] * rp[0];
             energy += rp[CORRLEN] * rp[CORRLEN];
             rp += NDEC;
             corr = 0.f;
             for (i = 0; i < CORRLEN; i += NDEC)
                     corr += rp[i] * l[i];
             scale = energy;
             if (scale < CORRMINPOWER)
                     scale = CORRMINPOWER;
             corr /= (Float)sqrt(scale);
             if (corr >= bestcorr) {
                     bestcorr = corr;
                     bestmatch = j;
             }
     }
48      /* fine search */
49      j = bestmatch - (NDEC - 1);
50      if (j < 0)
51              j = 0;
52      k = bestmatch + (NDEC - 1);
53      if (k > PITCHDIFF)
54              k = PITCHDIFF;
55      rp = &r[j];
56      energy = 0.f;
57      corr = 0.f;
58      for (i = 0; i < CORRLEN; i++) {
59              energy += rp[i] * rp[i];
60              corr += rp[i] * l[i];
61      }
62      scale = energy;
63      if (scale < CORRMINPOWER)
64              scale = CORRMINPOWER;
65      corr = corr / (Float)sqrt(scale);
66      bestcorr = corr;
67      bestmatch = j;
68      for (j++; j <= k; j++) {
69              energy -= rp[0] * rp[0];
70              energy += rp[CORRLEN] * rp[CORRLEN];
71              rp++;
72              corr = 0.f;
73              for (i = 0; i < CORRLEN; i++)
74                      corr += rp[i] * l[i];
75              scale = energy;
76              if (scale < CORRMINPOWER)
77                      scale = CORRMINPOWER;
78              corr = corr / (Float)sqrt(scale);
79              if (corr > bestcorr) {
80                      bestcorr = corr;
81                      bestmatch = j;
82              }
83      }
84      return PITCH_MAX - bestmatch;
85 }
```

When `findpitch()` is called, the contents of the pitch buffer match the contents of the history buffer. Line 15 sets the reference signal, l, to the sample 20 ms before the start of the erasure. Line 16 sets the lag signal to `MAX_PITCH` samples before that. Lines 19-29 compute the normalized cross-correlation at lag `MAX_PITCH` on a 2:1 decimated signal. Lines 26-28 clamp the energy at a minimum level, to avoid a divide by zero and de-emphasize regions with very low energy.

Lines 32-47 repeat the normalized cross-correlation calculation for every other lag. Only the even lags are examined. A running sum of the energy is kept so only two multiply accumulates (MACs) are required to update it per iteration. The peak of the correlation is held in `bestcorr`, and the corresponding lag is held in `bestmatch`.

Lines 48-54 compute the region used for the fine search. If the coarse peak is not at the minimum or maximum lag, 3 lags are searched in the fine search. Lines 55-83 repeat the calculations performed in the coarse search, but without decimation. The peak of the fine search is returned as the pitch estimate on line 84.

I.3.9 Synthetic signal generation and attenuation

The function `getfespeech()` extracts the synthesized waveform from the pitch buffer, while `scalespeech` applies the attenuation ramp to the synthesized speech.

```
 1 /*
 2  * Get samples from the circular pitch buffer. Update poffset so
 3  * when subsequent frames are erased the signal continues.
 4  */
 5 void LowcFE::getfespeech(short *out, int sz)
 6 {
 7      while (sz) {
 8              int cnt = pitchblen - poffset;
 9              if (cnt > sz)
10                      cnt = sz;
11              convertfs(&pitchbufstart[poffset], out, cnt);
12              poffset += cnt;
13              if (poffset == pitchblen)
14                      poffset = 0;
15              out += cnt;
16              sz -= cnt;
17      }
18 }
19
20 void LowcFE::scalespeech(short *out)
21 {
22      Float g = (Float)1. - (erasecnt - 1) * ATTENFAC;
23      for (int i = 0; i < FRAMESZ; i++) {
24              out[i] = (short)(out[i] * g);
25              g -= ATTENINCR;
26      }
27 }
```

`getfespeech()` on lines 5-18 does little more than copy the waveform from the pitch buffer to the output array. If the requested size is larger than the pitch buffer, the output pointer `poffset` rolls back to the start of the buffer and continues from there. The pitch buffer starts at `pitchbufstart` and is `pitchblen` samples long. The variable `poffset` is the current position in the buffer. `poffset` is updated at each iteration through the loop, and on return points to the next sample that should be output. This way synthetic signal generation can be continued if `getfespeech()` is called again.

The function `scalespeech()` on lines 20-27 applies the attenuation ramp to one frame's worth of synthetic speech. It is not called during the first frame of an erasure. On the second frame, `erasecnt` will be 1 as it hasn't been incremented yet, so the gain will start at 1. and ramp down to 0.8. Attenuation continues at 20% per frame for subsequent erased frames.

I.3.10 Overlap add operators

The overlap add operators complete the source code. Three routines are provided. Two of them are identical other then their argument types and overload the same member function name, `overlapadd`. The Float version is called for OLAs on the pitch buffer, while the short version is called for OLAs on the output signal.

```
/*
 * Overlap add left and right sides
 */
void LowcFE::overlapadd(Float *l, Float *r, Float *o, int cnt)
{
        Float incr = (Float)1. / cnt;
        Float lw = (Float)1. - incr;
        Float rw = incr;
        for (int i = 0; i < cnt; i++) {
                Float t = lw * l[i] + rw * r[i];
                if (t > 32767.)
                        t = 32767.;
                else if (t < -32768.)
                        t = -32768.;
                o[i] = t;
                lw -= incr;
                rw += incr;
        }
}

void LowcFE::overlapadd(short *l, short *r, short *o, int cnt)
{
        Float incr = (Float)1. / cnt;
        Float lw = (Float)1. - incr;
        Float rw = incr;
        for (int i = 0; i < cnt; i++) {
                Float t = lw * l[i] + rw * r[i];
                if (t > 32767.)
                        t = 32767.;
                else if (t < -32768.)
                        t = -32768.;
32              o[i] = (short)t;
33              lw -= incr;
34              rw += incr;
35      }
36 }
```

The OLA uses triangular windows that are computed in the routine based on the length of the `cnt` argument. Line 6 computes the window increment per sample, while lines 7 and 8 initialize the left and right side window weights. Lines 10-17 apply the weights to each sample, saturate the value to a 16-bit integer, output the result, and then update the weights for the next iteration.

An additional OLA routine, `overlapaddatend()`, is called at the first good frame after an erasure. This routine differs from those above by scaling the synthetic speech by the attenuation factor before it is combined with the input signal.

```
/*
 * Overlap add the end of the erasure with the start of the first good frame
 * Scale the synthetic speech by the gain factor before the OLA.
 */
void LowcFE::overlapaddatend(short *s, short *f, int cnt)
{
        Float incr = (Float)1. / cnt;
        Float gain = (Float)1. - (erasecnt - 1) * ATTENFAC;
        if (gain < 0.)
                gain = (Float)0.;
        Float incrg = incr * gain;
        Float lw = ((Float)1. - incr) * gain;
        Float rw = incr;
        for (int i = 0; i < cnt; i++) {
                Float t = lw * f[i] + rw * s[i];
                if (t > 32767.)
                        t = 32767.;
                else if (t < -32768.)
                        t = -32768.;
                s[i] = (short)t;
                lw -= incrg;
                rw += incr;
        }
}
```

I.4 Complexity and delay

The algorithm complexity is estimated to have a peak rate of approximately 1/2 of a DSP MIP. The average is much lower. The complexity is dominated by the calculation of the cross-correlation term in the pitch detection routine. This calculation only occurs during the first frame of an erasure. For all the other frames the complexity is very low – on the order of a few multiply accumulate (MAC) instructions per sample.

The complexity is estimated as follows. At the first erased frame in the erasure, the algorithm must estimate the pitch and perform an OLA for 1/4 of a pitch period. The maximum value of the pitch is 120 samples, so 1/4 of this is 30. The `findpitch()` and `overlapadd()` routine have the following operator counts:

Routine	**MAC**	**compare**	**div**	**sqrt**
`findpitch()`	3764	86	44	44
`overlapadd()`	121	0	1	0
Total	3885	86	45	44

Assuming 2 cycles for a compare, and 10 for a divide or sqrt this leads to 3885 + 86 * 2 + (45 + 44) * 10 = 4947 cycles. Since this occurs in a 10 ms frame, we multiply by 100 to yield 0.5 MIPS.

As mentioned previously the algorithm has an algorithm delay of 3.75 ms. To minimize the computational delay the PLC algorithm can be executed after every good frame, before it is known if the next frame is erased at a slight cost in memory. If the next frame is erased the synthetic signal is available immediately. If the next frame is not erased the synthetic signal is just discarded.

Bibliography

[1] ITU-T Recommendation G.723.1 (1996), *Speech coders: Dual rate speech coder for multimedia communications transmitting at 5.3 and 6.3 kbit/s*.

[2] CCITT Recommendation G.728 (1992), *Coding of speech at 16 kbit/s using low-delay code excited linear prediction*.

[3] ITU-T Recommendation G.729 (1996), *Coding of speech at 8 kbit/s using Conjugate-Structure Algebraic-Code-Excited Linear-Prediction (CS-ACELP).*

Printed in Switzerland

Geneva, 2000