
# ATIS-0100521.2005(S2020)

# Packet Loss Concealment for Use with ITU-T Recommendation G.711

AMERICAN NATIONAL STANDARD FOR TELECOMMUNICATIONS

As a leading technology and solutions development organization, the Alliance for Telecommunications Industry Solutions (ATIS) brings together the top global ICT companies to advance the industry's most pressing business priorities. ATIS' nearly 200 member companies are currently working to address the All-IP transition, 5G, network functions virtualization, big data analytics, cloud services, device solutions, emergency services, M2M, cyber security, network evolution, quality of service, billing support, operations, and much more. These priorities follow a fast-track development lifecycle — from design and innovation through standards, specifications, requirements, business use cases, software toolkits, open source solutions, and interoperability testing.

ATIS is accredited by the American National Standards Institute (ANSI). The organization is the North American Organizational Partner for the 3rd Generation Partnership Project (3GPP), a founding Partner of the oneM2M global initiative, a member of the International Telecommunication Union (ITU), as well as a member of the Inter-American Telecommunication Commission (CITEL). For more information, visit www.atis.org.

## AMERICAN NATIONAL STANDARD

Approval of an American National Standard requires review by ANSI that the requirements for due process, consensus, and other criteria for approval have been met by the standards developer.

Consensus is established when, in the judgment of the ANSI Board of Standards Review, substantial agreement has been reached by directly and materially affected interests. Substantial agreement means much more than a simple majority, but not necessarily unanimity. Consensus requires that all views and objections be considered, and that a concerted effort be made towards their resolution.

The use of American National Standards is completely voluntary; their existence does not in any respect preclude anyone, whether he has approved the standards or not, from manufacturing, marketing, purchasing, or using products, processes, or procedures not conforming to the standards.

The American National Standards Institute does not develop standards and will in no circumstances give an interpretation of any American National Standard. Moreover, no person shall have the right or authority to issue an interpretation of an American National Standard in the name of the American National Standards Institute. Requests for interpretations should be addressed to the secretariat or sponsor whose name appears on the title page of this standard.

**CAUTION NOTICE:** This American National Standard may be revised or withdrawn at any time. The procedures of the American National Standards Institute require that action be taken periodically to reaffirm, revise, or withdraw this standard. Purchasers of American National Standards may receive current information on all standards by calling or writing the American National Standards Institute.

## Notice of Disclaimer & Limitation of Liability

The information provided in this document is directed solely to professionals who have the appropriate degree of experience to understand and interpret its contents in accordance with generally accepted engineering or other professional standards and applicable regulations. No recommendation as to products or vendors is made or should be implied.

NO REPRESENTATION OR WARRANTY IS MADE THAT THE INFORMATION IS TECHNICALLY ACCURATE OR SUFFICIENT OR CONFORMS TO ANY STATUTE, GOVERNMENTAL RULE OR REGULATION, AND FURTHER, NO REPRESENTATION OR WARRANTY IS MADE OFMERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR AGAINST INFRINGEMENT OF INTELLECTUAL PROPERTY RIGHTS. ATIS SHALL NOT BE LIABLE, BEYOND THE AMOUNT OF ANY SUM RECEIVED IN PAYMENT BY ATIS FOR THIS DOCUMENT, AND IN NO EVENT SHALL ATIS BE LIABLE FOR LOST PROFITS OR OTHER INCIDENTAL OR CONSEQUENTIAL DAMAGES. ATIS EXPRESSLY ADVISES THAT ANY AND ALL USE OF OR RELIANCE UPON THE INFORMATION PROVIDED IN THIS DOCUMENT IS AT THE RISK OF THE USER.

NOTE - The user's attention is called to the possibility that compliance with this standard may require use of an invention covered by patent rights. By publication of this standard, no position is taken with respect to whether use of an invention covered by patent rights will be required, and if any such use is required no position is taken regarding the validity of this claim or any patent rights in connection therewith. Please refer to [http://www.atis.org/legal/patentinfo.asp] to determine if any statement has been filed by a patent holder indicating a willingness to grant a license either without compensation or on reasonable and non-discriminatory terms and conditions to applicants desiring to obtain a license.

ATIS-0100521.2005(S2020), *Packet Loss Concealment for Use with ITU-T Recommendation G.711*

Is an American National Standard developed by the **ATIS Network Performance, Reliability, and Quality of Service (PRQC)**.

*Published by*

**Alliance for Telecommunications Industry Solutions**
**1200 G Street, NW, Suite 500**
**Washington, DC 20005**

**ATIS-0100521.2005(S2020)**

(Revision of T1.521-1999 and T1.521a-2000)

American National Standard for Telecommunications

# Packet Loss Concealment for Use with ITU-T Recommendation G.711

Secretariat

**Alliance for Telecommunications Industry Solutions**

Approved March 11, 2005

**American National Standards Institute, Inc.**

**Abstract**

This standard describes Packet Loss Concealment algorithms for use in packetized speech transmission systems that use ITU-T Recommendation G.711 to code speech signals. These concealment algorithms enable high-quality speech transmission in operating environments where packet losses occur by providing high-quality packet loss recovery methods.

## FOREWORD

The information contained in this Foreword is not part of this American National Standard (ANS) and has not been processed in accordance with ANSI's requirements for an ANS. As such, this Foreword may contain material that has not been subjected to public review or a consensus process. In addition, it does not contain requirements necessary for conformance to the Standard.

In recent years, the Network Performance, Reliability, and Quality of Service Committee (PRQC)—formerly T1A1—has studied signal processing and performance aspects of speech packetization systems, issuing an ANSI standard (T1.509-1999) and a T1 Technical Report (Technical Report No. 45) on the subject. A shortcoming of those documents was the lack of clear information on high-quality methods for mitigating the effects of packets that are lost or destroyed during transmission.

The intent of this standard is to describe techniques and associated benefits of packet loss concealment (PLC) methods for use with ITU-T Recommendation G.711. The algorithms described in Annexes A & B are both single-ended; the implementation involves only the receiver. No modifications are required at the transmitting end. The Annexes provide example code; however, other implementations of the individual algorithms are possible. The performance of the methods defined in this standard has been demonstrated in subjective listening tests; the Annex A and Annex B methods are subjectively equivalent for the range of conditions tested.

Annex A describes a low-complexity, high-quality Packet Loss Concealment algorithm. In informal listening tests it performs well under a variety of input signal conditions: clean speech, noisy speech, music, and background noises, and compares favorably with the Packet Loss Concealment algorithms in several of the CELP-based speech coders standardized by the ITU-T.

Annex B describes a technique that employs linear prediction to estimate missing speech and uses the resultant vocal tract model output and excitation information to reconstruct the signal contained in missing packets. Formal subjective tests have been performed to evaluate this technique and unprotected G.711, using packet losses up to 10%, packet sizes up to 40 ms, and with clean and noisy speech. The results show significant benefit from the application of this technique compared to unprotected G.711.

The Alliance for Telecommunication Industry Solutions (ATIS) serves the public through improved understanding between carriers, customers, and manufacturers. The Alliance for Telecommunication Industry Solutions (ATIS) serves the public through improved understanding between carriers, customers, and manufacturers. The Network Performance, Reliability, and Quality of Service Committee (PRQC)—formerly T1A1—develops and recommends standards, requirements, and technical reports related to the performance, reliability, and associated security aspects of communications networks, as well as the processing of voice, audio, data, image, and video signals, and their multimedia integration. PRQC also develops and recommends positions on, and foster consistency with, standards and related subjects under consideration in other North American and international standards bodies.

ANSI guidelines specify two categories of requirements: mandatory and recommendation. The mandatory requirements are designated by the word *shall* and recommendations by the word *should*. Where both a mandatory requirement and a recommendation are specified for the same criterion, the recommendation represents a goal currently identifiable as having distinct compatibility or performance advantages.

Suggestions for improvement of this document are welcome. They should be sent to the Alliance for Telecommunications Industry Solutions, PRQC Secretariat, 1200 G Street NW, Suite 500, Washington, DC 20005.

At the time it approved this document, PRQC, which is responsible for the development of this Standard, had the following members:

R. Wohlert, PRQC Chair
N. Seitz, PRQC Vice-Chair
S. Carioti, ATIS Disciplines
S. Barclay, ATIS Secretariat
C. Underkoffler, ATIS Chief Editor
M. Perkins and L. Thorpe, PRQC Technical Editors

| Organization Represented | Name of Representative |
|---|---|
| Alcatel USA Inc. | Ken Biholar |
| ASTRI | Jacky Chow |
| AT&T | Percy Tarapore<br>Charles A. Dvorak (Alt.) |
| BellSouth Telecommunications | Archie McCain<br>David M. Brady (Alt.) |
| C.S.I Telecommunications | Michael S. Newman<br>Thomas G. Croda (Alt.) |
| Cingular Wireless LLC | Don Zelmer<br>Marc Grand (Alt.) |
| Defense Info. Systems Agency | Chris Fitzgerald |
| Ericsson Incorporated | Mustafa Kocaturk<br>Susana Sabater-Maroto (Alt.) |
| Harris Corporation | Marlis Humphrey |
| Intelsat | Mark T. Neibert |
| Lucent Technologies | Stuart O. Goldman |
| MCI | J. Martin Carroll<br>Robert Schafer (Alt.) |

| Organization Represented | Name of Representative |
|---|---|
| National Communications System | An Nguyen<br>Jean Trakinat (Alt.) |
| NTIA | Neal B. Seitz |
| Nortel Networks | Joseph A . Zebarth |
| Qwest | Steve Showell<br>Michael Fargano (Alt.) |
| SBC Communications, Inc. | Randolph Wohlert<br>Pierre Costa (Alt.) |
| Siemens Info & Comm Ntwks, Inc. | Suhas S. Gandhi<br>David E. Francisco (Alt.) |
| Sprint Corporation | Mark L. Jones |
| Telcordia Technologies | Spilios Makris<br>Cliff Halevi (Alt.) |
| Tellabs Operations, Inc. | William A. Walker<br>Kevin Stodola (Alt.) |
| Verizon Communications | John Colombo<br>Wendy Pugh (Alt.) |

# TABLE OF CONTENTS

## TABLE OF FIGURES



## TABLE OF TABLES

American National Standard for Telecommunications –

# Packet Loss Concealment for Use with ITU-T Recommendation G.711

## 1 PURPOSE, SCOPE, APPLICATION

### *1.1 Purpose*

In 1995, Working Group T1A1.7 (now, PRQC-QOS) published a Technical Report on speech packetization. That Report took note of the importance of packet loss recovery, but made no specific recommendations of how it should be achieved, choosing only to list a number of possible techniques. These included simple techniques such as repeating the previous packet, inserting white noise at the same power level as surrounding packets, or silence substitution.

To provide high-quality speech transmission in packetized systems that use G.711 and in which packet loss may occur, high quality methods for recovering from packet loss are required.

### *1.2 Scope*

This standard describes Packet Loss Concealment algorithms for use in packetized speech transmission systems that use ITU-T Recommendation G.711 to code speech signals. Mechanisms for detecting packet loss are not defined in this standard and will depend on the application.

### *1.3 Application*

The methods for Packet Loss Concealment (i.e., recovery from packet loss) described here are applicable to packetized speech transmission systems that use ITU-T Recommendation G.711 as the coding mechanism.

### *1.4 Conformance*

Conformance with this standard will be achieved by use of either the algorithm defined in Annex A or that defined in Annex B.

## 2 NORMATIVE REFERENCE

The following standard contains provisions that, through reference in this text, constitute provisions of this American National Standard. At the time of publication, the editions indicated are valid. All ref-

erences are subject to revision; all users of this American National Standard are therefore encouraged to investigate the possibility of applying the most recent edition of the standards listed below.

ITU-T Recommendation G.711 (11/88), *Pulse code modulation (PCM) of voice frequencies*.[1]

## 3 ABBREVIATIONS

| | |
|---|---|
| CELP | Code-Excited Linear Prediction |
| OLA | Overlap-Add |
| PLC | Packet Loss Concealment |
| TD-PSOLA | Time Domain Pitch Synchronous Overlap Add |
| WSOLA | Waveform Similarity Overlap Add |

## 4 DEFINITIONS

**4.1 Packet Loss:** Loss or corruption of speech segments (packets) during transmission.

**4.2 Packet Loss Concealment:** A method for reconstructing speech segments that have been lost or corrupted during transmission.

## 5 CONVENTIONS

In this Standard, we use "'G.711" as shorthand for "ITU-T Recommendation G.711" to improve readability. For the same reason, "erasure" is used to refer to a sequence of one or more lost packets.

## 6 PACKET LOSS CONCEALMENT FOR G.711

This standard presumes an audio system design where the input signal is encoded and packetized at the transmitter and sent over a network to a receiver that decodes the packet and plays out the output. Packet Loss Concealment (PLC) algorithms are intended to reduce the impairment caused when speech packets are lost or damaged during transmission. Many of the standard CELP-based speech coders, such as those defined in ITU-T Recommendations G.723.1, G.728, and G.729, have frame erasure concealment algorithms that are either built-in or defined as added features in their standards. The objective of PLC is similar to that of frame erasure concealment, namely to generate a synthetic speech signal to replace the missing speech signal that was contained in the lost packets or frames. Ideally, the synthesized signal has the same timbre and spectral characteristics as the missing signal, and does not create unnatural artifacts. Since speech signals are often locally stationary, it is possible to use the past history of the signal to generate a reasonable approximation to the missing segment. If the duration of lost packets (i.e., an erasure) is short, and does not occur in a region where the signal is changing rapidly, the erasures may be inaudible after concealment.

[1] This document is available from the International Telecommunications Union. < http://www.itu.int/ITU-T/ >

The PLC technique described in Annex A to this standard recognizes and goes beyond earlier work performed on pitch waveform replication techniques designed to conceal lost packets [3, 4]. Generating synthesized speech to replace lost packets has similarities to time-scale expanding of speech. In both cases, the goal is to generate synthetic signals near a region of original speech. Over the last decade, several elegant, high-quality and low-complexity techniques, such as WSOLA [1] and TD-PSOLA [2], have been devised for time-scaling. The PLC method described in Annex A benefits from these well-known time-scaling techniques.

The PLC algorithm described in Annex B to this standard uses the well-known linear predictive model of speech production that is widely used in low bit-rate speech coding. The algorithm estimates the spectral characteristics of a missing segment, and then synthesizes a high-quality approximation to the missing segment using this production model.

Unlike CELP-based coders, G.711 has no model of speech production — i.e., there is nothing in the encoder to help the decoder with the concealment. Hence, the concealment algorithm for G.711 will be independent of the encoder. This implies that the PLC adds computational complexity, memory requirements, and delay to the decode process of G.711. This standard recognizes that G.711 is likely to operate in computationally sparse environments. Thus, the computational and memory requirements of PLC are minimized. On the other hand, G.711 has the advantage that the signal returns to the original signal at the first sample in the first good packet after an erasure. With CELP-based coders, the decoder's state variables take time to recover after an erasure, especially if the coder is backward adaptive. Thus, PLC in G.711 has the ability to recover rapidly after an erasure is over.

**Annex A**
(Normative)

# A REVERSE ORDER REPLICATED PITCH PERIODS (RORPP) ALGORITHM

The PLC technique defined in this Annex, called *Reverse Order Replicated Pitch Periods* (*RORPP*), is scalable with respect to the size of the packets. It is capable of concealing lost packets when the packets include up to 30 ms of speech, and is effective when packet loss rates are as high as 20%. Annoying impairments such as clicks, pops, and harshness are minimized, though intelligibility may suffer when packet loss rates are high.

## *A.1 Algorithm Description*

To add PLC to a G.711 system that currently does not conceal losses, changes are only required in the receiver. In the following discussion, it is assumed that the coder is G.711, the audio input signal is sampled at 8 kHz, and each packet contains 10 ms (80 samples) of audio. Any packet size or sampling rate can be accommodated by adjusting a few parameters.

### A.1.1 Good Packets

During normal operation (good packets), the receiver decodes the received packet and sends its output to the audio port. To support PLC, two operational changes are required at the receiver:

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 a sequence of lost packets (i.e., an erasure).
2. *The output is delayed by 3.75 ms (30 samples) before it is sent to the audio port.* This algorithm delay is used for an overlap add (OLA) at the start of an erasure and allows the PLC code to make a smooth transition between the real and synthesized signals.

### A.1.2 First Lost Packet

At the start of an erasure (when a packet loss is first detected), 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 *last_quarter buffer*, is made to allow for the case where the erasure lasts longer than 10 ms.

### A.1.3 Pitch Detection

The pitch 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), corresponding to frequencies of 200 to 66 Hz. The pitch range was chosen based on a range used

in the post-filter used in ITU-T Rec. G.728. 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 erasure. 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.

From WSOLA [1], 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 results will be obtained.

### A.1.4 Synthetic Signal Generation for First 10 ms

For the first 10 ms of lost speech, 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 ensure a smooth transition between the real and synthetic signal, and a smooth transition if the pitch period is repeated multiple times, an OLA is performed with a triangular window on 1/4 of the pitch period between the last and next to last pitch period. For 1/4 pitch period, 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 `last_quarter` buffer multiplied by a down-sloping ramp. If complexity is not an issue, the triangular windows may be replaced with Hanning windows in all 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 packet, replacing the original signal. This introduces the algorithm delay; the tail of the last packet cannot be output until it is known whether the next packet is lost. If a packet loss occurs, the signal in the tail of the last good packet is modified by the OLA to ensure 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 it 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 lost packet, good packet, lost packet sequence occurs.

### A.1.5 Synthetic Signal Generation after 10 ms

If the next packet is also lost, 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, on long erasures it introduces unnatural harmonic artifacts (i.e., “beeps”). This is especially noticeable if the erasure occurs 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 sounds quite 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. Beyond 20 ms no 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 lost packets, 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.

Updating the pitch buffer is accomplished in the same manner as during the first lost packet, except with an increased number of pitch periods. For example, at the start of the second lost packet, for 1/4 pitch period, 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 pitch period in the last_quarter buffer multiplied by a down-sloping ramp. The result of the OLA replaces the last 1/4 pitch period 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.

### A.1.6 Attenuation

As with other PLC algorithms, such as those defined for G.729 and for G.728, long sequence of lost packets necessitate attenuation of 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 synthesized signal is not attenuated. At the start of the second 10 ms, the synthesized signal is attenuated linearly with a ramp at the rate of 20% per 10 ms. After 60 ms the synthesized signal is zero.

### A.1.7 First Good Packet after an Erasure

At the first good packet after an erasure, a smooth transition is needed between the synthesized speech and the actual signal. To do this, the synthesized speech from the pitch buffer is continued beyond the end of the erasure, and is then mixed with the actual 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 pitch period window is used. For longer erasures the window is increased by 4 ms per 10 ms of erasure, up to a maximum of the packet size—i.e., 10 ms.

## *A.2 Application to Other Speech Coders*

The algorithm defined in this Annex may be adapted for use with other coders that do not provide PLC, such as ITU-T Rec. G.726. When using this algorithm with other coders, it is important to maintain any state information present in the coder. G.711 does not maintain state information between samples, but almost all other speech coders have such information. When the coder maintains state information, the PLC algorithm defined in this Annex may be used as it is to generate the synthetic speech during the erasure, but care must be taken to ensure that the coder's internal state variables track the synthetic speech. Otherwise, after the erasure is over, artifacts and discontinuities will appear

as the decoder restarts using its erroneous state information. To some degree, this is mitigated by the OLA window at the first good packet after an erasure, but more needs to be done to ensure proper operation of the encoder/decoder pair. Better results can be obtained by having the decoder's state variables track the synthesized speech during the erasure—that is, convert the decoder into an encoder for the duration of the erasure using the synthesized output of the concealment algorithm as the input to the encoder. With many coders, the encoder has higher complexity than the decoder, so the added complexity may be an issue. However, it should be noted that unlike a typical encoder, this encoder is only being run to maintain state information and its output will never be used. Thus, shortcuts may be taken that significantly decrease the encoder complexity but allow the state variables to track the output sufficiently well that artifacts are avoided when the erasure is over.

## *A.3 Example*

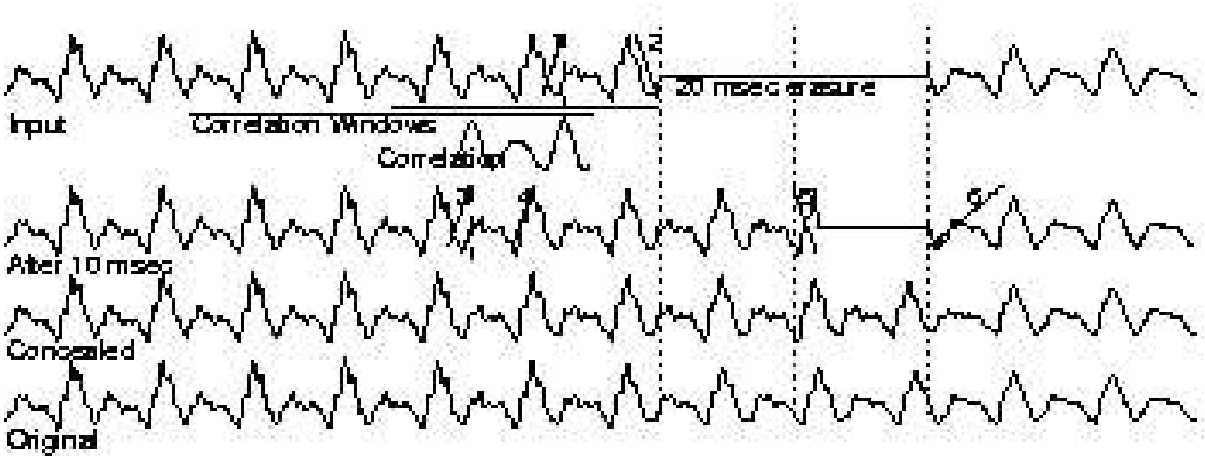


**Figure A.1 - RORPP Packet Loss Concealment Algorithm for G.711**

Figure A.1 shows a graphical example of how the algorithm operates when a 20 ms erasure (i.e., a sequence of two 10 ms packets) occurs in a voiced segment of speech from a male speaker. The top waveform ("Input") shows the input. The location of the erasure is delimited at 10 ms intervals by the three *dotted* vertical lines that cross all waveforms. The speech before the erasure 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. The algorithm steps are:

1. *The pitch is estimated using normalized autocorrelation.* The horizontal lines labeled "Correlation Windows" show the windows. The upper line corresponds to the 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 graph labeled "Correlation" under the windows is the normalized autocorrelation. The peak of this graph, shown by the *dashed* vertical line at "1" on the input waveform, is the pitch estimate. The time interval between this line and the start of the erasure is one pitch period.
2. *The contents of the 1/4 pitch period before the erasure, under the window labeled "2", are saved in a buffer* (`last_quarter`) in case the erasure lasts more than 10 ms.

3. *A single period pitch buffer is created.* To ensure a smooth transition between the real and synthetic signal, and in case the pitch buffer is repeated more than once during the first 10 ms, an OLA with a triangular window is performed on the 1/4 pitch period before the erasure (window 2) with the 1/4 pitch period from the previous pitch period (window 1). The result of this OLA replaces the region under window 2 in both the output signal and in the pitch buffer.

4. *For the first 10 ms, the synthesized signal is generated by repeating the single period pitch buffer as many times as needed.* The results of this are shown in the "After 10 msec" waveform. The offset into the pitch buffer at the end of 10 ms is saved in a variable called pitch_offset. The signal is extrapolated an extra 1/4 pitch period beyond the 10 ms erasure boundary to ensure a smooth transition to the next packet. If the erasure ends after 10 ms, an OLA is performed on this 1/4 pitch period and the input signal; and the concealment algorithm is finished. In this example, the erasure is longer than 10 ms so the pitch buffer is lengthened to increase the signal variation.

5. *Another pitch period is added to the pitch buffer.* A dashed vertical line has been drawn on the "After 10 ms" waveform two pitch periods back from the start of the erasure. An OLA is performed on the 1/4 pitch period before this line (window 3) and the last_quarter buffer saved in step 2 (window 2). The result is placed in the tail of the pitch buffer. For the second 10 ms, the pitch buffer is thus the region between the dashed vertical line and first dotted vertical line in the "After 10 ms" waveform, with the exception that the last 1/4 pitch period is the result of the above OLA.

6. *To ensure a smooth transition between the single and double period pitch buffers, an OLA is performed for 1/4 pitch period at the start of the second 10 ms in the erasure.* An OLA is performed on the region under window 5 and the region under window 4. The result replaces the signal located at window 5. The location of window 4 is calculated from the pitch_offset pointer saved in step 4 by subtracting pitch periods until the pitch_pointer is in the first pitch period of the currently used portion of the pitch buffer. This ensures that the proper waveform phase is maintained as the number of periods in the pitch buffer is increased.

7. *For the duration of the second 10 ms lost packet, 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 packet. If the erasure continues beyond 20 ms, a 1/4 pitch period 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. 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.

8. *At the start of the first good packet, an OLA is performed on the extrapolated synthetic signal and the beginning of the real signal.* The OLA window for the real signal for this 20 ms erasure (1/4 pitch period + 4 ms) is shown by window 6. The result of this OLA replaces the signal under window 6.

The "Concealed" waveform shows the final output from the algorithm. For comparison purposes the "Original" waveform without any erasures is also shown. 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 two 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.

## A.4 Complexity and Delay

The algorithm complexity is estimated to have a peak rate of approximately 0.5 DSP MIPS. 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 packet of an erasure. For all the other packets, 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 packet, 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. Two routines, `findpitch` and `overlapadd`, are defined and have the following operator counts:

**Table A.1 - Frequency of operator occurrence in findpitch and overlapadd routines**

| Operator | MAC | Compare | Divide | Sqrt |
|---|---|---|---|---|
| `findpitch()` | 3764 | 86 | 44 | 44 |
| `overlapadd()` | 121 | | 1 | |
| **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 packet, 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 (at a slight cost in memory), the PLC algorithm can be executed after every good packet, before it is known if the next packet is lost. If the next packet is lost, the synthetic signal is available immediately. If the next packet is not lost, the synthetic signal is simply discarded.

Reference C++ code to implement this algorithm is found in the next clause.

## A.5 Annotated C++ Code

The PLC algorithm described in this Annex has been implemented in floating-point with a C++ class. The code, along with an explanation of how it works is presented in this section. The complete implementation is about 360 lines of C++ code, including comments.

### A.5.1 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 `float` to `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 buff length*/
#define NDEC           2              /* 2:1 decimation */
#define CORRLEN        160            /* 20 msec 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 msec at 8 KHz */
#define ATTENFAC       ((Float).2)    /* attenu. factor per 10 ms frame */
#define ATTENINCR      (ATTENFAC/FRAMESZ) /* attenuation per sample */
```

#### A.5.1.1 Class Declaration

```
 1 class LowcFE {
 2 public:
 3             LowcFE();
 4    void     dofe(short *s);              /* synthesize speech for erasure
*/
 5    void     addtohistory(short *s);      /* add a good frame to history
buf */
 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. The interface to the class is defined by three public functions. The constructor on line 3 initializes the internal variables. The function `addtohistory` on line 5 handles good packets. Its argument is a pointer to an array of `short` values of length `FRAMESZ`. This array should contain a single packet of speech 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 packets during an erasure. Its value starts at zero and is incremented by one every 10 ms during an erasure. It is reset to zero at the first good packet after an erasure.

Each of the following variables serve as indicated:

- `poverlap` is the length of the OLA window, which corresponds to 1/4 of the current pitch.
- `poffset` is the offset into the current pitch buffer and is used for synthetic waveform generation
- `pitch` is the current pitch estimate
- `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 packet
- `pitchbufend` points to the end of the pitch buffer
- `pitchbufstart` is a pointer to the start of the currently used portion of the pitch buffer
- `pitchbuf` contains the history buffer at the start of an erasure. Other than the last 1/4 pitch period, this buffer remains static for the duration of the erasure
- `lastq` is a buffer containing a cached copy of the last quarter pitch period of signal before the erasure begins. It is used in OLA operations
- `history` is a buffer that contains the recent history of the signal. It is updated during good and erased packets.

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

### A.5.1.2 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 function `receiveframe` returns `TRUE` if a 10 ms G.711 bit-stream packet has been received without errors and `FALSE` if an erasure occurs. The function `g711dec` converts the G.711 bit-

stream into 16-bit uniform PCM and `output` outputs the audio with lost packets concealed. Implementations of these functions are beyond the scope of this standard.

```
 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 packet is lost. If the packet 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 packet is erased the `dofe` member function is called to generate the synthetic speech.

It should be noted that `addtohistory` (line 12) does more than just copy the packet 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 packet after an erasure an OLA is performed on the input signal and the synthesized speech before returning.

### A.5.1.3 Utility Member Functions

The utility functions on lines 25 to 29 of the class declaration do conversions from `short` to `Float` (`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++)
 4      t[i] = (Float)f[i];
 5 }
 6
 7 void LowcFE::convertfs(Float *f, short *t, int cnt)
 8 {
 9    for (int i = 0; i < cnt; i++)
10      t[i] = (short)f[i];
11 }
12
13 void LowcFE::copyf(Float *f, Float *t, int cnt)
```

```
14 {
15   for (int i = 0; i < cnt; i++)
16     t[i] = f[i];
17 }
18
19 void LowcFE::copys(short *f, short *t, int cnt)
20 {
21   for (int i = 0; i < cnt; i++)
22     t[i] = f[i];
23 }
24
25 void LowcFE::zeros(short *s, int cnt)
26 {
27   for (int i = 0; i < cnt; i++)
28     s[i] = 0;
29 }
```

There is no saturation or rounding in `convertfs`.

#### A.5.1.4 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, `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.

### A.5.2 ADDTOHISTORY and SAVESPEECH

Next we present the public interface function `addtohistory`, and a protected function, `savespeech`. `savespeech` is called internally by `addtohistory`. `addtohistory` is called by the application after decoding a good packet, but before the signal is output.

```
1 /*
2  * Save a frames worth of new speech in the history buffer.
3  * Return the output speech delayed by POVERLAPMAX.
4  */
5 void LowcFE::savespeech(short *s)
6 {
7   /* make room for new signal */
8   copys(&history[FRAMESZ], history, HISTORYLEN - FRAMESZ);
```

```
 9    /* copy in the new frame */
10    copys(s, &history[HISTORYLEN - FRAMESZ], FRAMESZ);
11    /* copy out the delayed frame */
12    copys(&history[HISTORYLEN - FRAMESZ - POVERLAPMAX], s, FRAMESZ);
13  }
14
15  /*
16   * A good frame was received and decoded.
17   * If right after an erasure, do an overlap add with the synthetic sig-
nal.
18   * Add the frame to history buffer.
19   */
20  void LowcFE::addtohistory(short *s)
21  {
22    if (erasecnt) {
23      short overlapbuf[FRAMESZ];
24      /*
25       * longer erasures require longer overlaps
26       * to smooth the transition between the synthetic
27       * and real signal.
28       */
29      int olen = poverlap + (erasecnt - 1) * EOVERLAPINCR;
30      if (olen > FRAMESZ)
31              olen = FRAMESZ;
32      getfespeech(overlapbuf, olen);
33      overlapaddatend(s, overlapbuf, olen);
34      erasecnt = 0;
35    }
36    savespeech(s);
37  }
```

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

If the previous packet 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. `poverlap` contains the number of samples in 1/4 of the current estimated pitch period. If the erasure is only 10 ms (i.e., `erasecnt == 1`), the OLA window length is set to `poverlap`. If multiple packets were lost, the length of the OLA window is increased by 4 ms for every 10 ms of erasure. Lines 30-31 ensure 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, an OLA is performed on the synthetic signal and 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 packet after an erasure, the original speech is saved. Otherwise, it's the result of the OLA on line 33.

`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 packet. The new packet is copied to the tail of the buffer on line 10, and line 12 re-

places the contents of the input array with the signal delayed by `POVERLAPMAX` (30) samples. This introduces the algorithm delay of 3.75 ms.

### A.5.3 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 zero. If so, this is the first packet 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 `Float` 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. `Findpitch` returns a value between `PITCH_MIN` (40) and `PITCH_MAX` (120). `findpitch` dominates the complexity of the algorithm and is only called on the first packet 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 packet. 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, an OLA is performed on the contents of `lastq` and the 1/4 period starting 1.25 periods back in the pitch buffer. This ensures a smooth transition between the original speech and the synthetic speech at the start of erasure, as well as ensuring a smooth transition if the single period pitch buffer is repeated during the first packet, 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 ensures the speech resulting from the OLA 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 packet by calling `getfespeech`, which 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. `savespeech` also delays the signal, so that 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 packets 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 packet is continued for 1/4 period and placed in a temporary buffer. An OLA is performed on this signal and the new expanded pitch period buffer. This ensures 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 ensures 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 packet, this ensures 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 packet, and an OLA is performed on the 1/4 period at the start of this data and 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 lost packet. Note that the synthetic signal is not attenuated during the first packet 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 zero on line 46.

#### A.5.3.1 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. A fine search is then 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 msec of speech.
 4  * r - points to the sample PITCH_MAX before l
 5  */
 6 int LowcFE::findpitch()
 7 {
 8   int   i, j, k;
 9   int   bestmatch;
10   Float bestcorr;
11   Float corr;      /* correlation */
12   Float energy;          /* running energy */
13   Float scale;           /* scale correlation by average power */
14   Float *rp;       /* segment to match */
15   Float *l = pitchbufend - CORRLEN;
16   Float *r = pitchbufend - CORRBUFLEN;
17
18   /* coarse search */
19   rp = r;
20   energy = 0.f;
21   corr = 0.f;
22   for (i = 0; i < CORRLEN; i += NDEC) {
23        energy += rp[i] * rp[i];
24        corr += rp[i] * l[i];
25   }
26   scale = energy;
27   if (scale < CORRMINPOWER)
28        scale = CORRMINPOWER;
29   corr = corr / (Float)sqrt(scale);
30   bestcorr = corr;
```

```
31    bestmatch = 0;
32    for (j = NDEC; j <= PITCHDIFF; j += NDEC) {
33          energy -= rp[0] * rp[0];
34          energy += rp[CORRLEN] * rp[CORRLEN];
35          rp += NDEC;
36          corr = 0.f;
37          for (i = 0; i < CORRLEN; i += NDEC)
38                corr += rp[i] * l[i];
39          scale = energy;
40          if (scale < CORRMINPOWER)
41                scale = CORRMINPOWER;
42          corr /= (Float)sqrt(scale);
43          if (corr >= bestcorr) {
44                bestcorr = corr;
45                bestmatch = j;
46          }
47    }
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, 1, to the sample 20 ms before the start of the erasure. Line 16 sets the lag signal to PITCH_MAX samples before that. Lines 19-29 compute the normalized cross-correlation at lag PITCH_MAX 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.

### A.5.3.2 Synthetic Signal Generation and Attenuation

`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. `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.

`scalespeech` on lines 20-27 applies the attenuation ramp to one packet of synthetic speech. It is not called during the first packet of an erasure. On the second packet, `erasecnt` will be one, as it has not been incremented. Thus, the gain will start at 1.0 and ramp down to 0.8. Attenuation continues at 20% per packet for subsequent erased packets.

#### A.5.3.3 Overlap Add Operators

The overlap add operators complete the source code. Three routines are provided. Two of them are identical other than 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.

```
 1 /*
 2  * Overlap add left and right sides
 3  */
 4 void LowcFE::overlapadd(Float *l, Float *r, Float *o, int cnt)
 5 {
 6   Float incr = (Float)1. / cnt;
 7   Float lw = (Float)1. - incr;
 8   Float rw = incr;
 9   for (int i = 0; i < cnt; i++) {
10         Float t = lw * l[i] + rw * r[i];
11         if (t > 32767.)
12               t = 32767.;
13         else if (t < -32768.)
14               t = -32768.;
15         o[i] = t;
16         lw -= incr;
17         rw += incr;
18   }
19 }
20
21 void LowcFE::overlapadd(short *l, short *r, short *o, int cnt)
22 {
23   Float incr = (Float)1. / cnt;
24   Float lw = (Float)1. - incr;
25   Float rw = incr;
26   for (int i = 0; i < cnt; i++) {
27         Float t = lw * l[i] + rw * r[i];
28         if (t > 32767.)
29               t = 32767.;
```

```
30          else if (t < -32768.)
31               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 packet 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.

```
 1 /*
 2  * Overlap add the erasure tail with the start of the first good frame
 3  * Scale the synthetic speech by the gain factor before the OLA.
 4  */
 5 void LowcFE::overlapaddatend(short *s, short *f, int cnt)
 6 {
 7    Float incr = (Float)1. / cnt;
 8    Float gain = (Float)1. - (erasecnt - 1) * ATTENFAC;
 9    if (gain < 0.)
10         gain = (Float)0.;
11    Float incrg = incr * gain;
12    Float lw = ((Float)1. - incr) * gain;
13    Float rw = incr;
14    for (int i = 0; i < cnt; i++) {
15         Float t = lw * f[i] + rw * s[i];
16         if (t > 32767.)
17              t = 32767.;
18         else if (t < -32768.)
19              t = -32768.;
20         s[i] = (short)t;
21         lw -= incrg;
22         rw += incr;
23    }
24 }
```

**Annex B**
(Normative)

# B LP-BASED PACKET LOSS CONCEALMENT ALGORITHM

This Annex describes a packet loss concealment (PLC) algorithm that uses well-known linear prediction techniques. It works with packet sizes of 5 ms or larger, including the most commonly used range of 10 to 30 ms. The default sampling rate is 8 kHz, but the algorithm can support any sampling rate with a slight modification of some parameters. Subjective test results demonstrate that the performance of the algorithm is comparable to that of the algorithm in Annex A.

## *B.1 Algorithm*

The LP-based PLC algorithm is implemented entirely at the receiver side of a transmission channel. It operates well with typical packet sizes, and requires no extra action on the part of the transmitter. The algorithm uses the well-known speech production model, where speech is generated by passing an excitation signal through an LP filter. In this model, a speech signal is composed of two components: a predictable signal that contains the vocal tract information, and a residual signal that contains the excitation information. The basic operation of the algorithm is to estimate these two components for a missing speech segment, based on the components extracted from the previously received speech. The components are then combined to generate an approximation of the missing segment.

### B.1.1 State Variables

The algorithm uses the following state variables and buffers:

- *Speech buffer*. The most recent 30 ms of speech is kept in the speech buffer. The first 25 ms of this buffer contains samples that have already been played and the last 5 ms contains samples that have been received but not yet played.
- *Overlap buffer*. This buffer contains a 5-ms extension of the generated signal. It is used for overlap-and-adding with the first good packet that comes after the lost segment.
- *LP coefficients*. The LP coefficients that are calculated at the first packet of a lost speech segment are stored and used with consecutive lost packets.
- *Excitation buffer*. The excitation signal for a lost packet is stored in the excitation buffer. It is used if the following packet is also lost.
- *Pitch period*. The pitch period estimate for the first packet of a lost speech segment is stored and used with consecutive lost packets.
- *Previous loss indicator*. A binary flag that shows whether the previous packet was lost or not.
- *Scale*. Current value of the envelope that scales the signal to be played.

### B.1.2 First Lost Packet

A lost speech segment contains one or more lost packets. The majority of the computations in the algorithm are done at the first packet of a lost speech segment. Figure B.1 shows a block diagram of these computations. Whenever the current packet is missing and the previous loss indicator is not set, the algorithm sets the previous loss indicator and executes the blocks shown in Figure B.1. First, the autocorrelation function and the LP coefficients of the previous speech samples are computed. The LP coefficients are used by both the LP filter and the inverse LP filter. The previous speech samples are filtered by the LP filter to extract the vocal tract information. The resulting residual signal is processed by a pitch detector to estimate the pitch period of the previous speech. The pitch period and the residual signal are then used to produce an excitation signal for the lost packet. The LP coefficients extracted from the previous speech are used to approximate the inverse LP filter for the lost packet, and the excitation signal is passed through this inverse LP filter to add the vocal tract information. The filter output is combined with the last 5-msec segment of the previous speech samples by an overlap-and-add operation, generating the output frame. Finally, this output frame is scaled by a decreasing envelope and then played. Each of these blocks is described in detail in the following sections.

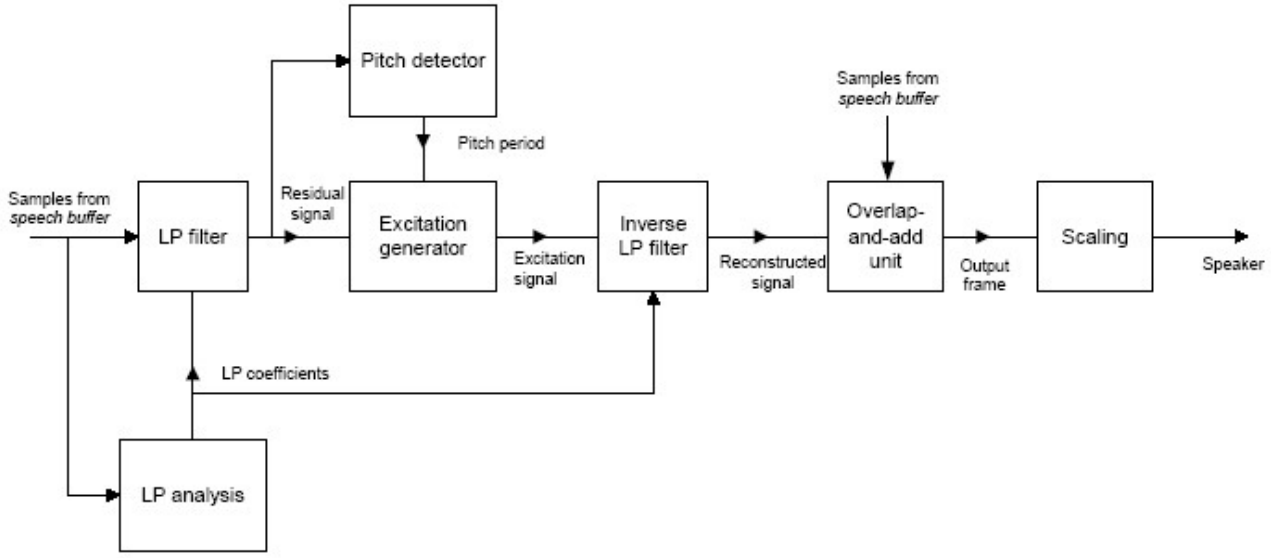


**Figure B.1 - Block diagram of the algorithm for the first lost packet**

#### B.1.2.1 LP Analysis

The LP analysis block computes the LP coefficients of the previous speech using the samples in the second half of the *speech buffer*. An LP order of 20 is used. The last 15 ms of the *speech buffer* is first windowed by an asymmetric Hamming window. Then the autocorrelation function of the windowed speech is computed, and checked for stability purposes. If the check fails, or if the signal energy is too low, then LP coefficients of all zeros are passed to the LP filter and the inverse LP filter. Otherwise, white noise correction and a 60-Hz bandwidth expansion are applied by windowing the autocorrelation function by an exponential lag window, and the LP coefficients are computed by using the Durbin algorithm with the windowed autocorrelation function. The LP coefficients are passed to the LP filter and the inverse LP filter. They are also stored, and used at consecutive lost packets.

### B.1.2.2 LP Filter

The entire *speech buffer* is filtered by the LP filter to extract the vocal tract characteristics. The resulting residual signal is passed to the pitch detector and the excitation generator.

### B.1.2.3 Pitch Detector

The pitch period of the previous speech is estimated by computing the normalized autocorrelation function of the residual signal and then searching for the peak location. Pitch periods ranging from 2.5 ms to 15 ms are searched at intervals of 0.125 ms.

The pitch detector also performs a voiced/unvoiced decision for better performance during unvoiced speech sections. Using a small pitch period during these sections of speech introduces unpleasant harmonics. Therefore, a sufficiently large multiple of the estimated pitch period is used when the detector determines that the packet contains unvoiced speech.

The estimated pitch period (or a multiple of the estimated pitch period) is passed to the excitation generator. It is also stored, and used at subsequent and consecutive lost packets.

### B.1.2.4 Excitation Generator (first lost packet)

The residual signal of the previous speech and the estimated pitch period are used to generate an excitation for the lost packet and the two 5-msec segments just before and after the lost packet. This process is illustrated in Figure B.2 for a pitch period of *P* samples. First, the 5-msec segment just before the last *P* samples is copied from the residual signal into the beginning of the new excitation signal. Then, the last *P* samples of the residual signal are appended to the excitation signal as many times as necessary to fill the remaining portion. The entire excitation signal (packet size plus 10 ms) is passed to the inverse LP filter. It is also stored at the *excitation buffer*, and used if the next packet is also lost.

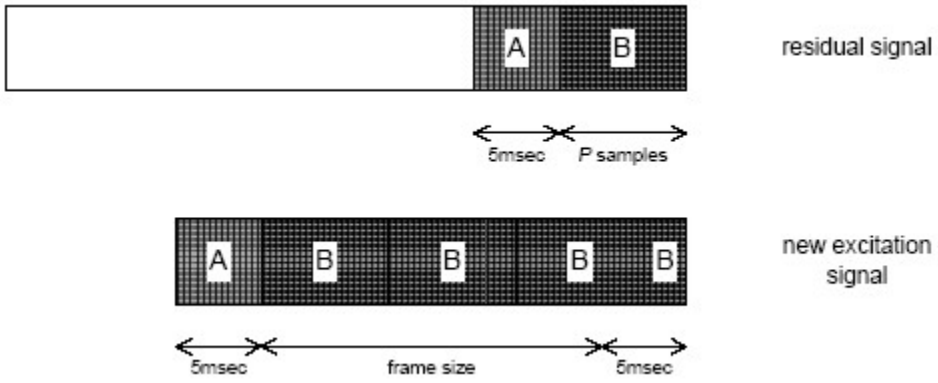


**Figure B.2 - Generating the new excitation from the residual signal**

#### B.1.2.5 Inverse LP Filter

The excitation signal is filtered by the inverse LP filter to add the vocal tract information. The 20 samples just before the last 5-msec segment of the *speech buffer* are used as the initial conditions of the filter. The reconstructed signal at the output of the filter is passed to the overlap-and-add unit.

#### B.1.2.6 Overlap-and-Add Unit

The overlap-and-add unit uses the reconstructed signal and the samples from the *speech buffer* to generate the output frame. For a packet size of $T$ ms, the two 5-msec segments at the beginning and end of the reconstructed signal are used for overlap-and-add operations, while the $T$-ms middle portion replaces the lost packet. The 5-msec segment at the beginning of the reconstructed signal and the 5-msec segment at the end of the *speech buffer* are weighted by a triangular window and summed. The resulting signal replaces the 5-msec segment at the end of the *speech buffer*. It is also copied into the first 5 ms of the output frame. If the packet size, $T$, is less than 30 ms, then the entire *speech buffer* is shifted by $T$ ms, and the middle portion of the reconstructed signal is copied to the end of the *speech buffer*. Otherwise, the contents of the entire *speech buffer* are replaced by the last 30-msec segment in the middle portion of the reconstructed signal. The ($T$ – 5)-ms segment at the beginning of the middle portion of the reconstructed signal is also copied into the remaining locations of the output frame. Finally, the 5-msec segment at the end of the reconstructed signal is copied into the *overlap buffer*. It used at the next packet if the next packet is not lost.

#### B.1.2.7 Scaling (lost packets)

The output frame is scaled down before it is played out to the speaker, by multiplying each sample by the current value of the *scale*. The *scale* is set to 1.0 when the algorithm is initialized. Starting from its value at the beginning of the current output frame, it is decreased at each sample with a slope of 0.054 per 10-ms packet, and multiplied by that sample. This process continues until the last sample of the output frame or for 20 ms, whichever is first. After 20 ms, the slope is increased to 0.222 per 10-ms packet. The rest of the output frame is completely muted if the *scale* is zero. The value of *scale* at the end of the output frame is retained, and used at the following packets.

### B.1.3 Consecutive Packet Losses

If the current packet is lost and the *previous loss indicator* is also set, then the LP analysis, LP filter, and the pitch detector blocks are skipped. The LP coefficients and the pitch period that were stored previously are used instead. The operation of the remaining blocks are the same as the first lost packet case except the excitation generator, which is explained below.

#### B.1.3.1 Excitation Generator (consecutive packet losses)

If the previous packet was also lost, then the excitation generator uses the previously stored pitch period and the excitation signal to generate the new excitation. It starts by copying the last 10-msec segment from the *excitation buffer* into the beginning of the new excitation signal. Then, it continues as in the first lost packet case, i.e., it appends the last $P$ samples of the *excitation buffer* to the new excitation

signal as many times as necessary to fill the remaining portion, where *P* is the stored pitch period. The generated excitation signal (packet size plus 10 ms) is passed to the inverse LP filter. It also replaces the old excitation in the *excitation buffer*. The operation of the excitation generator for consecutive packet losses is illustrated in Figure B.3.

### B.1.4 Good Packets

If the current packet is not lost, then the algorithm first decodes the current packet. If the *previous loss indicator* is set, then the current packet is the first good packet after a lost segment. In this case, the algorithm resets the *previous loss indicator*, and the current packet is modified by an overlap-and-add operation, as explained below. In all cases, the output frame is constructed from the last 5-msec segment in the *speech buffer* and the entire current packet except the last 5-msec segment. If the packet size, *T*, is less than 30 ms, then the entire *speech buffer* is shifted by *T* ms, and the current packet is copied to the end of the *speech buffer*. Otherwise, the contents of the entire *speech buffer* are replaced by the last 30-msec segment of the current packet. The new output frame is scaled, if necessary, as explained below and then played.

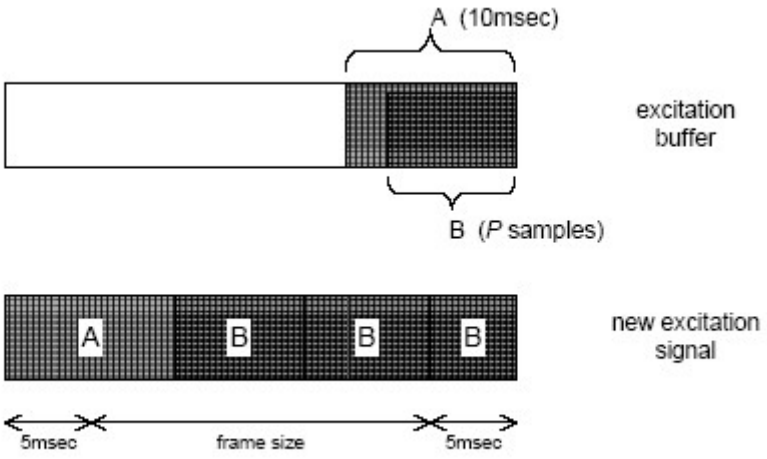


**Figure B.3 - Generating the new excitation for consecutive packet losses**

#### B.1.4.1 Overlap-and-Add Unit (first good packet only)

The 5-msec segment at the beginning of the current packet and the samples in the *overlap buffer* are weighted by a triangular window and summed. The resulting signal replaces the 5-msec segment at the beginning of the current packet.

#### B.1.4.2 Scaling (good packets)

If the value of the *scale* is less than 1.0, then the output frame is scaled up before it is played out to the speaker. For the first good packet received after a lost packet, the current value of the *scale* is retained for the duration of the overlap window, and each sample of the output frame is multiplied by the *scale*. For the remaining samples of the output frame, the sample is multiplied by the *scale* and then the *scale*

is increased with a slope of 0.498 per 10-msec packet. This process continues until the last sample of the output frame, or until *scale* reaches to 1.0. The final value of the *scale* is retained, and used at the following packets.

## B.2 Delay and complexity

The overlap-and-add operation between a reconstructed frame and the following good packet requires an additional delay. Therefore, the *speech buffer* introduces a delay of 5 ms, always keeping a portion of the received speech that has not been played yet.

The computational complexity of the LP-based PLC algorithm depends on the actual packet size that is used. The computational requirements obtained by counting the number of MAC, compare, square root, and divide operations in the worst-case branch are displayed in Table B.1 for the most commonly used packet sizes. In the calculation, it was assumed that a MAC operation requires 1 cycle, a compare operation requires 2 cycles, and a square root or a divide operation requires 10 cycles. It can be seen that the algorithm is not computationally intense, and most applications can easily accommodate these requirements.

**Table B.1 - Computational requirements of the LP-based PLC algorithm**

| Packet size | Computational complexity |
|---|---|
| 10 ms | 2.3 mips |
| 20 ms | 1.3 mips |
| 30 ms | 0.9 mips |

## B.3 Annotated C code

A floating-point implementation of the LP-based PLC algorithm is presented in this section.

### B.3.1 Constants and type definitions

```
1     /*                                                                  */
2     /* constants */
3     #define MAX_FR_SIZE     480
4     #define DEFAULT_FR_SIZE 80
5     #define MIN_ENR         (double)64577 // energy threshold (-63dBov)
6     #define PL              20    // min acceptable pitch period (samples)
7     #define PH              119   // max acceptable pitch period (samples)
8     #define OVERLAP_SIZE    40
9     #define SCALE_DEC1      22
10    #define SCALE_DEC2      91
11    #define SLOPE_THR       29247 // 1db attenuation level
12    #define SCALE_INC       204
13    #define BUFFER_SIZE     240
14    #define LPC_SIZE        20
15    #define COR_WINDOW_SIZE 100
16    #define AUTOCORR_SIZE   120
17    #define VOICED_THR      (double)1300 // voiced/unvoiced threshold
18    #define MIN_UV_PER      80           // min. period for unvoiced case
19
20    /* type definitions */
```

```
21    typedef int                   S16;
22    typedef unsigned int          U16;
23    typedef long int              S32;
24    typedef unsigned long int     U32;
25
26    typedef struct{
27          S16         speech[BUFFER_SIZE];
28          S16         overlap_buffer[OVERLAP_SIZE];
29          double      lp_coefs[LPC_SIZE];
30          double      excitation[MAX_FR_SIZE+2*OVERLAP_SIZE];
31          S16         pitch_period;
32          S16         prev_loss_ind;
33          S16         scale;
34    } STATE_DEF;
```

### B.3.2 Global variables and tables

```
1     STATE_DEF state_var;    // state variables and buffers
2     S16 frame_size;
3     S16 wgt_window[OVERLAP_SIZE] =      // for overlap-and-add operations
4           {31969, 31170, 30370, 29571, 28772, 27973, 27173, 26374, 25575,
5            24776, 23977, 23177, 22378, 21579, 20780, 19980, 19181, 18382,
6            17583, 16784, 15984, 15186, 14386, 13587, 12789, 11989, 11190,
7            10390,  9591,  8791,  7992,  7193,  6394,  5595,  4795,  3996,
8             3197,  2398,  1598,   799};
9     double hamming[AUTOCORR_SIZE];      // for autocorrelation computation
10    double lag_window[LPC_SIZE];  // for bandwidth expansion
```

The Hamming and bandwidth expansion tables are loaded into the memory by calling the function, `load_tables()`.

```
1     void load_tables(void)
2     {
3           S16 i;
4           double pi=4.0*atan(1.0);
5
6           for (i=0; i<80; ++i)
7                 hamming[i] = 0.54-0.46*cos((pi*i)/(double)79.5);
8           for (i=0; i<40; ++i)
9                 hamming[AUTOCORR_SIZE-1-i] = 0.54-0.46*cos((pi*i)/(double)39.5);
10          for (i=1; i<=LPC_SIZE; ++i)
11                lag_window[i-1] = exp(-pi*pi*i*i*72/(double)640000.0)/1.0001;
12          return;
13    }
```

### B.3.3 Initialization and the main loop

The initialization function, `lp_plc_init()`, is called at call setup with the address of the state variables that will be used by the algorithm. It sets the state variables to their initial values. The main algorithm is then called with each received packet together with a packet loss indicator. An example usage is shown in the `main()` program below. The program first calls the initialization function, and then enters the main loop. It calls a function, `get_frame()`, to get the samples of the current packet and a binary loss indicator. For good packets, `get_frame()` returns the received samples and a loss indicator equal to zero. For lost packets, it sets the loss indicator to one. The main loop calls the main algorithm, `lp_plc()`, for each packet. It passes the addresses of the received samples, output buffer and

the state variables, and the value of the packet loss indicator to the algorithm. The samples returned in the output buffer by the algorithm are then sent to the output.

```
1     /* Initializes the state variables of the algorithm, pointed by stptr. */
2     void lp_plc_init(STATE_DEF *stptr)
3     {
4           S16 i;
5
6           stptr->prev_loss_ind = 0;
7           stptr->scale = 32767;
8           for (i=0; i<BUFFER_SIZE; ++i)
9                 stptr->speech[i] = 0;
10          for (i=0; i<frame_size+2*OVERLAP_SIZE; ++i)
11                stptr->excitation[i] = 0.0;
12          return;
13    }
14
15    main()
16    {
17          S16 current_frame[MAX_FR_SIZE]; // Input buffer
18          S16 output_frame[MAX_FR_SIZE];  // Output buffer
19          S16 current_loss_ind;         // Current packet loss indicator
20          STATE_DEF *stptr; // Pointer to state variables and buffers
21
22          /* Load tables and initialize the state variables */
23          load_tables();
24          stptr = &state_var;
25          frame_size = DEFAULT_FR_SIZE;
26          lp_plc_init(stptr);
27
28          /* Main loop */
29          while( get_frame(current_frame, &current_loss_ind) )
30          {     lp_plc(current_frame, output_frame, stptr, current_loss_ind);
31                write_frame(output_frame);
32          }
33    }
```

### B.3.4 Main algorithm

The main algorithm receives a binary packet loss indicator and the addresses of the input samples, output buffer, and the state variables. If the loss indicator is zero, then the input samples are expanded to linear domain and then used to construct the output and to update buffers. Otherwise, the output is generated using the buffered samples, as explained in the previous sections.

```
1  /* Main LP-based PLC algorithm. */
2  void lp_plc(S16 *current_frame, S16 *output_frame,
3                                  STATE_DEF *stptr, S16 current_loss_ind)
4  {
5     double r[LPC_SIZE+1];               // autocorrelation function
6     double residual_signal[BUFFER_SIZE-LPC_SIZE];
7     S16 reconstructed_signal[MAX_FR_SIZE+2*OVERLAP_SIZE];
8     S16 i;
9
10    if (current_loss_ind == 0)
11    {     /* decode current packet */
```

```
12          ulaw_expand(frame_size, current_frame, current_frame);
13          if (stptr->prev_loss_ind != 0)
14                /* first good packet, do overlap-and-add */
15                overlap_and_add(current_frame, stptr->overlap_buffer, 1);
16          construct_output(output_frame, current_frame, stptr->speech);
17          if (stptr->scale < 32767)
18                stptr->scale = scale_up(output_frame, stptr->scale,
19                                              stptr->prev_loss_ind);
20          stptr->prev_loss_ind = 0;
21    }
22    else
23    {     if (stptr->prev_loss_ind == 0)
24          {     /* first lost packet */
25                stptr->prev_loss_ind = 1;
26                auto_corr(stptr->speech, r);
27                /* check for a proper autocorrelation function */
28                for (i=1; i<=LPC_SIZE; ++i)
29                {     if (r[0] <= fabs(r[i]))
30                      {     r[0] = 0.0;
31                            break;
32                      }
33                }
34                /* proceed with all-zero coefficients if the check fails
35                      or the energy is below -63dBov */
36                if (r[0] < MIN_ENR)
37                {     for (i=0; i<LPC_SIZE; ++i)
38                            stptr->lp_coefs[i] = 0.0;
39                }
40                else
41                {     /* apply white-noise correction and bandwidth expansion,
42                            and compute the LP coefficients */
43                      for (i=1; i<=LPC_SIZE; ++i)
44                            r[i] = r[i] * lag_window[i-1];
45                      durbin(r, stptr->lp_coefs);
46                }
47                lp_filter(stptr->speech, stptr->lp_coefs, residual_signal);
48                stptr->pitch_period = find_pitch(residual_signal);
49                construct_exc(residual_signal, stptr->excitation,
50                                              stptr->pitch_period);
51          }
52          else
53          {  /* previous packet was also lost, use the stored variables */
54                construct_exc_mult(stptr->excitation, stptr->pitch_period);
55          }
56          inverse_lp_filter(stptr->excitation, stptr->lp_coefs,
57                                        stptr->speech, reconstructed_signal);
58          overlap_and_add(&(stptr->speech[BUFFER_SIZE-OVERLAP_SIZE]),
59                                        reconstructed_signal, 0);
60          construct_output(output_frame, &reconstructed_signal[OVERLAP_SIZE],
61                                        stptr->speech);
62          for (i=0; i<OVERLAP_SIZE; ++i)      // modify the overlap buffer
63                stptr->overlap_buffer[i]=
64                      reconstructed_signal[frame_size+OVERLAP_SIZE+i];
65          stptr->scale = scale_down(output_frame, stptr->scale);
66    }
67    return;
68 }
```

The functions that are called by the main algorithm are shown in the following sections.

### B.3.5 LP analysis and filtering

```
1     /* Computes the autocorrelation function of the last AUTOCORR_SIZE samples
2     in speech. Returns the resulting LPC_SIZE coefficients in r. */
3     void auto_corr(S16 *speech, double *r)
4     {
5           double win_speech[AUTOCORR_SIZE];
6           S16 i, j;
7           double sum;
8
9           for(i=0; i<AUTOCORR_SIZE; ++i) // scaling by a hamming window
10                win_speech[i] =
11                      (double)speech[BUFFER_SIZE-AUTOCORR_SIZE+i] * hamming[i];
12          for(i=0; i<=LPC_SIZE; ++i)
13          {     sum = 0.0;
14                for(j=0; j<AUTOCORR_SIZE-i; ++j)
15                      sum = sum + win_speech[j] * win_speech[j+i];
16                r[i] = sum;
17          }
18          return;
19    }
20
21    /* Converts the autocorrelation coefficients in r into LP coefficients,
22       and returns the result in lp_coefs */
23    void durbin(double *r, double *lp_coefs)
24    {
25          double next_coefs[LPC_SIZE];
26          S16 i, j;
27          double K;
28          double energy;
29          double sum;
30
31          /* initialize */
32          for (i=0; i<LPC_SIZE; ++i)
33                lp_coefs[i] = 0.0;
34          K = -r[1] / r[0];
35          lp_coefs[0] = K;
36          energy = r[0] * (1.0 - K*K);
37          /* main loop */
38          for(i=1; i<LPC_SIZE; ++i)
39          {     sum = r[i+1];
40                for(j=1; j<=i; ++j)
41                      sum = sum + r[j] * lp_coefs[i-j];
42                if (fabs(sum) > energy)
43                      break;
44                K = -sum / energy;
45                for(j=0; j<i; ++j)
46                      next_coefs[j] = lp_coefs[j] + K * lp_coefs[i-j-1];
47                next_coefs[i] = K;
48                energy = energy * (1.0 - K*K);
49                for(j=0; j<=i; ++j)
50                      lp_coefs[j] = next_coefs[j];
51          }
52          return;
```

```
53    }
54
55    /* LP filters the samples in speech using the coefficients in lp_coefs.
56       Returns the result in res_signal. */
57    void lp_filter(S16 *speech, double *lp_coefs, double *res_signal)
58    {
59          S16 i,j;
60
61          for (i=0; i<BUFFER_SIZE-LPC_SIZE; ++i)
62          {     res_signal[i] = (double)speech[i+LPC_SIZE];
63                for (j=1; j<=LPC_SIZE; ++j)
64                      res_signal[i] = res_signal[i] +
65                            lp_coefs[j-1]*(double)speech[i+LPC_SIZE-j];
66          }
67    }
68
69    /* Passes the samples in excitation through the inverse LP filter, whose
70       coefficients are in lp_coefs. Uses samples from speech as the initial
71       conditions. Returns the result in rec_signal. */
72    void inverse_lp_filter(double *excitation, double *lp_coefs,
73                                        S16 *speech, S16 *rec_signal)
74    {
75          S16 i, j;
76          S16 exc_size;
77          double sum;
78
79          exc_size = frame_size+2*OVERLAP_SIZE;
80          for (i=0; i<LPC_SIZE; ++i)
81          {     sum = excitation[i];
82                for (j=1; j<=i; ++j)
83                      sum = sum - lp_coefs[j-1]*(double)rec_signal[i-j];
84                for (j=i+1; j<=LPC_SIZE; ++j)
85                      sum = sum - lp_coefs[j-1] *
86                            (double)speech[BUFFER_SIZE-OVERLAP_SIZE+i-j];
87                rec_signal[i] = double_to_int_r(sum);
88          }
89          for (i=LPC_SIZE; i<exc_size; ++i)
90          {     sum = excitation[i];
91                for (j=1; j<=LPC_SIZE; ++j)
92                      sum = sum - lp_coefs[j-1]*(double)rec_signal[i-j];
93                rec_signal[i] = double_to_int_r(sum);
94          }
95          return;
96    }
```

### B.3.6 Pitch prediction

```
1     /* Computes and returns the pitch period of the signal in res_signal. The
2        pitch period is searched in the range from PL to PH samples. */
3     S16 find_pitch(double *res_signal)
4     {
5           double r_res[PH-PL+1];        // autocorrelation of residual
6           double mx_num, mx_den, new_den, sum;
7           S16 mx_ind;
8           S16 final_est;
9           S16 i, j;
10          double signal_power;
11
12          /* compute autocorrelation */
```

```
13          for (i=PL; i<=PH; ++i)
14          {     sum = 0.0;
15                for (j=BUFFER_SIZE-LPC_SIZE-COR_WINDOW_SIZE-i;
16                                          j<BUFFER_SIZE-LPC_SIZE-i; ++j)
17                      sum = sum + res_signal[j] * res_signal[j+i];
18                r_res[i-PL] = sum;
19          }
20          signal_power = 0.0;
21          for (j=BUFFER_SIZE-LPC_SIZE-PH-COR_WINDOW_SIZE;
22                                          j<BUFFER_SIZE-LPC_SIZE-PH; ++j)
23                signal_power = signal_power + res_signal[j] * res_signal[j];
24          /* search for maximum */
25          mx_num = r_res[PH-PL];        // numerator of norm. autocorrelation
26          mx_den = sqrt(signal_power);  // denominator of norm. autocorrelation
27          mx_ind = PH-PL;
28          for (i=PH-PL-1; i>=0; --i)
29          {     signal_power = signal_power -
30                      res_signal[BUFFER_SIZE-LPC_SIZE-i-PL-COR_WINDOW_SIZE-1] *
31                      res_signal[BUFFER_SIZE-LPC_SIZE-i-PL-COR_WINDOW_SIZE-1];
32                signal_power = signal_power +
33                      res_signal[BUFFER_SIZE-LPC_SIZE-i-PL-1] *
34                      res_signal[BUFFER_SIZE-LPC_SIZE-i-PL-1];
35                new_den = sqrt(signal_power);
36                if ( (r_res[i] * mx_den) > (mx_num * new_den) )
37                {     mx_num = r_res[i];
38                      mx_den = new_den;
39                      mx_ind = i;
40                }
41          }
42          final_est = mx_ind + PL;
43          if (mx_den == 0.0  ||  mx_num < VOICED_THR*mx_den)
44          {     i = final_est;                // unvoiced
45                while (final_est < MIN_UV_PER)
46                      final_est += i;
47          }
48          return (final_est);
49    }
```

### B.3.7 Excitation generation

```
1   /* Constructs the excitation signal using the residual signal and the
2      estimated pitch period. Used if the previous packet was not lost. */
3   void construct_exc(double *res_signal, double *excitation, S16 period)
4   {
5      S16 i, i_limit;
6      S16 res_size, exc_size;
7
8      res_size = BUFFER_SIZE-LPC_SIZE;
9      exc_size = frame_size+2*OVERLAP_SIZE;
10     for (i=0; i<OVERLAP_SIZE; ++i)
11           excitation[i] = res_signal[res_size-period-OVERLAP_SIZE+i];
12     i_limit = period;
13     if (i_limit > exc_size-OVERLAP_SIZE)
14           i_limit = exc_size-OVERLAP_SIZE;
15     for (i=0; i<i_limit; ++i)
16           excitation[OVERLAP_SIZE+i] = res_signal[res_size-period+i];
17     i = OVERLAP_SIZE + i_limit;
18     while (i<exc_size)
19     {     excitation[i] = excitation[i-period];
```

```
20          ++i;
21    }
22    return;
23 }
24
25 /* Constructs the excitation signal using the previous excitation and the
26    pitch period. The result replaces the previous excitation. Used if the
27    previous packet was also lost. */
28 void construct_exc_mult(double *excitation, S16 period)
29 {
30    double temp_buffer[2*MIN_UV_PER-2];
31    S16 exc_size;
32    S16 i;
33
34    exc_size = frame_size + 2*OVERLAP_SIZE;
35    for (i=0; i<2*OVERLAP_SIZE; ++i)
36          temp_buffer[i] = excitation[frame_size+i];
37    for (i=2*OVERLAP_SIZE; i<period; ++i)
38          temp_buffer[i] = excitation[frame_size-period+i];
39    for (i=0; i<period; ++i)
40          excitation[i] = temp_buffer[i];
41    while (i<exc_size)
42    {     excitation[i] = excitation[i-period];
43          ++i;
44    }
45    return;
46 }
```

### B.3.8 Overlap-and-add operations

```
1     /* If direction is 0, then multiplies data1 by wgt_window, multiplies
2        data2 by reversed wgt_window, and adds the results into data1. If
3        direction is not 0, then uses wgt_window for data2 and reversed
4        wgt_window for data1, storing the result again in data1. */
5     void overlap_and_add (S16 *data1, S16 *data2, S16 direction)
6     {
7           S32 prd1, prd2, longsum;
8           S16 i;
9
10          if (direction == 0)
11          {     for (i=0; i<OVERLAP_SIZE; ++i)
12                {     prd1 = (S32)data1[i] * (S32)wgt_window[i];
13                      prd2 = (S32)data2[i] * (S32)wgt_window[OVERLAP_SIZE-1-i];
14                      longsum = prd1 + prd2 + (S32)0x00004000;
15                      if (prd1>0 && prd2>0 && longsum<=0)
16                            data1[i] = 0x7FFF;
17                      else if (prd1<0 && prd2<0 && longsum>=0)
18                            data1[i] = 0x8000;
19                      else
20                            data1[i] = (S16)(longsum >> 15);
21                }
22          }
23          else
24          {     for (i=0; i<OVERLAP_SIZE; ++i)
25                {     prd1 = (S32)data1[i] * (S32)wgt_window[OVERLAP_SIZE-1-i];
26                      prd2 = (S32)data2[i] * (S32)wgt_window[i];
27                      longsum = prd1 + prd2 + (S32)0x00004000;
28                      if (prd1>0 && prd2>0 && longsum<=0)
29                            data1[i] = 0x7FFF;
```

```
30                      else if (prd1<0 && prd2<0 && longsum>=0)
31                            data1[i] = 0x8000;
32                      else
33                            data1[i] = (S16)(longsum >> 15);
34                }
35          }
36          return;
37    }
```

### B.3.9 Constructing the output frame and the buffer updates

```
1     /* Constructs the output frame using the last OVERLAP_SIZE samples in
2        speech and the frame_size-OVERLAP_SIZE samples in new_data. Also
3        updates speech by the samples in new_data. */
4     void construct_output(S16 *output_fr, S16 *new_data, S16 *speech)
5     {
6           S16 i;
7
8           for (i=0; i<OVERLAP_SIZE; ++i)
9                 output_fr[i] = speech[BUFFER_SIZE-OVERLAP_SIZE+i];
10          for (i=OVERLAP_SIZE; i<frame_size; ++i)
11                output_fr[i] = new_data[i-OVERLAP_SIZE];
12          if (frame_size < BUFFER_SIZE)
13          {     for (i=0; i<BUFFER_SIZE-frame_size; ++i)
14                      speech[i] = speech[frame_size+i];
15                for (i=0; i<frame_size; ++i)
16                      speech[BUFFER_SIZE-frame_size+i] = new_data[i];
17          }
18          else
19          {     for (i=0; i<BUFFER_SIZE; ++i)
20                      speech[i] = new_data[frame_size-BUFFER_SIZE+i];
21          }
22          return;
23    }
```

### B.3.10 Scaling functions

```
1   /* Scales the output frame by an envelope starting at the value of scale.
2      The envelope is constant for the first 5ms of the first good frame.
3      Then it is increased by SCALE_INC at each sample, with a maximum
4      of 32767 (1 in Q15). The final value of the envelope is returned. */
5   S16 scale_up(S16 *output_fr, S16 scale, S16 prev_loss_ind)
6   {
7      S16 i;
8
9      i = 0;
10     if (prev_loss_ind != 0)
11     {
12          for ( ; i<OVERLAP_SIZE; ++i)
13                output_fr[i] = mpyr(output_fr[i], scale);
14     }
15     while (scale < 32767  &&  i < frame_size)
16     {     output_fr[i] = mpyr(output_fr[i], scale);
17          scale += SCALE_INC;
18          if (scale < 0)
19                scale = 32767;
20          i++;
21     }
22     return(scale);
```

```
}

/* Scales the output frame by an envelope starting at the value of scale.
   At each sample the envelope is decreased by SCALE_DEC1 until 1dB
   attenuation, and by SCALE_DEC2 afterwards, with a minimum of zero. The
   final value of the envelope is returned. */
S16 scale_down(S16 *output_fr, S16 scale)
{
   S16 i;

   i = 0;
   while (scale > SLOPE_THR  &&  i < frame_size)
   {     scale -= SCALE_DEC1;
         output_fr[i] = mpyr(output_fr[i], scale);
         i++;
   }
   while (scale > 0  &&  i < frame_size)
   {     scale -= SCALE_DEC2;
         if (scale < 0)
               scale = 0;
         output_fr[i] = mpyr(output_fr[i], scale);
         i++;
   }
   if (scale == 0)
   {     while (i < frame_size)
               output_fr[i++] = 0;
   }
   return(scale);
}
```

### B.3.11 Utility functions

```
   /* Returns the product of two Q15 numbers after rounding. The result is
      also in Q15. */
   S16 mpyr(S16 x, S16 y)
   {
         S32 long_x, long_y, long_p;
         S16 p;

         long_x = (S32)x;
         long_y = (S32)y;
         long_p = (long_x * long_y) + (S32)0x00004000;
         p = (S16)(long_p >> 15);
         return(p);
   }

   /* Converts a double to 16-bit integer after rounding. */
   S16 double_to_int_r(double x)
   {
         S16 y;

         if (x>32766.5)
               y = 0x7FFF;
         else if (x<-32768.0)
               y = 0x8000;
         else
               y = (S16)(x+0.5);
         return(y);
   }
```

**Annex C**
(Informative)

# C BIBLIOGRAPHY

[1] W. Verhelst and M. Roelands, "An Overlap-Add Technique Based on Waveform Similarity (WSOLA) for High Quality Time-Scale Modification of Speech," *Proc. IEE ICASSP-93*, pp. 554-557, 1993.

[2] E. Moulines and F. Charpentier, "Pitch-Synchronous Waveform Processing Techniques for Text-to-Speech Synthesis Using Diphones", *Speech Communication* Vol. 9 (5/6) pp. 453-467, 1990.

[3] D. J. Goodman, G. B. Lockhart, O. J. Wasem, and W-C. Wong, "Waveform Substitution Techniques for Recovering Missing Speech Segments in Packet Voice Communications", *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. 34, No 6: pp. 1440-1448, December 1986.

[4] O. J. Wasem, D. J. Goodman, C.A. Dvorak, and H. G. Page, "The Effect of Waveform Substitution on the Quality of PCM Packet Communications", *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. 36, No 3: pp. 342-348, March 1988.

[5] T1.TR.45-1995, *A Technical Report on Speech Packetization*.[2]

[6] ITU-T Recommendation G.723.1 (03/96), *Dual rate speech coder for multimedia communications transmitting at 5.3 and 6.3 kbit/s.*[1]

[7] ITU Recommendation G.726 (12/90), *40, 32, 24, 16 kbit/s Adaptive Differential Pulse Code Modulation (ADPCM).*[1]

[8] ITU-T Recommendation G.728 (09/92), *Coding of speech at 16 kbit/s using low-delay code excited linear prediction.*[1]

[9] ITU-T Recommendation G.729 (03/96), *Coding of speech at 8 kbit/s using Conjugate-Structure Algebraic-Code-Excited Linear-Prediction (CS-ACELP).*[1]

[10] L.R. Rabiner and R.W. Schafer, Chapter 8 in *Digital Processing of Speech Signals,* Upper Saddle River, NJ: Prentice Hall, 1978, pp. 396–461.

[2] This document is available from the Alliance for Telecommunications Industry Solutions (ATIS), 1200 G Street N.W., Suite 500, Washington, DC 20005. <http://www.atis.org>