Digital Speech Compression

Putting the GSM 06.10 RPE-LTP algorithm to work

Jutta Degener

Jutta is a member of the Communications and Operating Systems research group at the Technical University of Berlin. She can be reached at jutta@cs.tu-berlin.de.

The good news is that new, sophisticated, real-time conferencing applications delivering speech, text, and images are coming online at an ever-increasing rate. The bad news is that at times, these applications can bog down the network to the point where it becomes unresponsive. Solutions to this problem range from increasing network bandwidth—a costly and time-consuming process—to compressing the data being transferred. I'll discuss the latter approach.

In 1992, my research group at the Technical University of Berlin needed a speech-compression algorithm to support our video-conferencing research. We found what we were looking for in GSM, the Global System for Mobile telecommunication protocol suite that's currently Europe's most popular protocol for digital cellular phones. GSM is a telephony standard defined by the European Telecommunications Standards Institute (ETSI). In this article, I'll present the speech-compression part of GSM, focusing on the GSM 06.10 RPE-LTP ("regular pulse excitation long-term predictor") full-rate speech transcoder. My colleague Dr. Carsten Bormann and I have implemented a GSM 06.10 coder/decoder (codec) in C. Its source is available via anonymous ftp from site ftp.cs.tu-berlin.de in the /pub/local/kbs/tubmik/gsm/ directory. It is also available electronically from DDJ; see "Availability," page 3. Although originally written for UNIX-like environments, others have ported the library to VMS and MS-DOS.

Human Speech

When you pronounce "voiced" speech, air is pushed out from your lungs, opening a gap between the two vocal folds, which is the glottis. Tension in the vocal folds (or cords) increases until—pulled by your muscles and a Bernoulli force from the stream of air—they
close. After the folds have closed, air from your lungs again forces the glottis open, and the cycle repeats--between 50 to 500 times per second, depending on the physical construction of your larynx and how strong you pull on your vocal cords.

For "voiceless" consonants, you blow air past some obstacle in your mouth, or let the air out with a sudden burst. Where you create the obstacle depends on which atomic speech sound ("phoneme") you want to make. During transitions, and for some "mixed" phonemes, you use the same airstream twice--first to make a low-frequency hum with your vocal cords, then to make a high-frequency, noisy hiss in your mouth. (Some languages have complicated clicks, bursts, and sounds for which air is pulled in rather than blown out. These instances are not described well by this model.)

You never really hear someone's vocal cords vibrate. Before vibrations from a person's glottis reach your ear, those vibrations pass through the throat, over the tongue, against the roof of the mouth, and out through the teeth and lips.

The space that a sound wave passes through changes it. Parts of one wave are reflected and mix with the next oncoming wave, changing the sound's frequency spectrum. Every vowel has three to five typical ("formant") frequencies that distinguish it from others. By changing the interior shape of your mouth, you create reflections that amplify the formant frequencies of the phoneme you're speaking.

Digital Signals and Filters

To digitally represent a sound wave, you have to sample and quantize it. The sampling frequency must be at least twice as high as the highest frequency in the wave. Speech signals, whose interesting frequencies go up to 4 kHz, are often sampled at 8 kHz and quantized on either a linear or logarithmic scale.

The input to GSM 06.10 consists of frames of 160 signed, 13-bit linear PCM values sampled at 8 kHz. One frame covers 20 ms, about one glottal period for someone with a very low voice, or ten for a very high voice. This is a very short time, during which a speech wave does not change too much. The processing time plus the frame size of an algorithm determine the "transcoding delay" of your communication. (In our work, the 125-ms frames of our input and output devices caused us more problems than the 20-ms frames of the GSM 06.10 algorithm.)

The encoder compresses an input frame of 160 samples to one frame of 260 bits. One second of speech turns into 1625 bytes; a megabyte of compressed data holds a little more than ten minutes of speech.

Central to signal processing is the concept of a filter. A filter's output can depend on more than just a single input value--it can also keep state. When a sequence of values is passed through a filter, the filter is "excited" by the sequence. The GSM 06.10 compressor models the human-speech system with two filters and an initial excitation. The linear-predictive short-term filter, which is the first stage of compression and the last during decompression, assumes the role of the vocal and nasal tract. It is excited by the output of a long-term predictive (LTP) filter that turns its input--the residual pulse excitation (RPE)--into a mixture of glottal wave and voiceless noise.

As Figure 1 illustrates, the encoder divides the speech signal into short-term predictable parts, long-term predictable parts, and the remaining residual pulse. It then quantizes and encodes
that pulse and parameters for the two predictors. The decoder (the synthesis part) reconstructs the speech by passing the residual pulse through the long-term prediction filter, and passes the output of that through the short-term predictor.

Figure 1.

**Linear Prediction**

To model the effects of the vocal and nasal tracts, you need to design a filter that, when excited with an unknown mixture of glottal wave and noise, produces the speech you're trying to compress. If you restrict yourself to filters that predict their output as a weighted sum (or "linear combination") of their previous outputs, it becomes possible to determine the sum's optimal weights from the output alone. (Keep in mind that the filter's output---the speech---is your algorithm's input.)

For every frame of speech samples $s[n]$, you can compute an array of weights $lpc[P]$ so that $s[n]$ is as close as possible to $lpc[0] * s[n-1] + lpc[1] * s[n-2] + \ldots + lpc[P-1] * s[n-P]$ for all sample values $s[n]$. $P$ is usually between 8 and 14; GSM uses eight weights. The `levinson_durbin()` function in [Listing Two](#) calculates these linear-prediction coefficients.

The results of GSM's linear-predictive coding (LPC) are not the direct $lpc[]$ coefficients of the filter equation, but the closely related "reflection coefficients;" see Figure 2. This term refers to a physical model of the filter: a system of connected, hard-walled, lossless tubes through which a wave travels in one dimension.
When a wave arrives at the boundary between two tubes of different diameters, it does not simply pass through: Parts of it are reflected and interfere with waves approaching from the back. A reflection coefficient calculated from the cross-sectional areas of both tubes expresses the rate of reflection. The similarity between acoustic tubes and the human vocal tract is not accidental; a tube that sounds like a vowel has cross-sections similar to those of your vocal tract when you pronounce that vowel. But the walls of your mouth are soft, not lossless, so some wave energy turns to heat. The walls are not completely immobile either--they resonate with lower frequencies. If you open the connection to your nasal tract (which is normally closed), the model will not resemble the physical system very much.

**Short-Term and Long-Term Analysis**

The short-term analysis section of the algorithm calculates the short-term residual signal that will excite the short-term synthesis stage in the decoder. We pass the speech backwards through the vocal tract by almost running backwards the loop that simulates the reflections inside the sequence of lossless tubes; see Listing Three. The remainder of the algorithm works on 40-sample blocks of the short-term residual signal, producing 56 bits of the encoded GSM frame with each iteration.

The LTP analysis selects a sequence of 40 reconstructed short-term residual values that resemble the current values. LTP scales the values and subtracts them from the signal. As in the LPC section, the current output is predicted from the past output.

The prediction has two parameters: the LTP lag, which describes the source of the copy in time, and the LTP gain, the scaling factor. (The \[\text{lpc[]}\] array in the LPC section played a similar role as the LTP gain, but there was no equivalent to the LTP lag; the lags for the sum were always 1, 2, 8). To compute the LTP lag, the algorithm looks for the segment of the past that most resembles the present, regardless of scaling. How do we compute resemblance? By correlating the two sequences whose resemblance we want to know. The correlation of two sequences, \(x[n]\) and \(y[n]\), is the sum of the products \(x[n]*y[n-lag]\) for all \(n\). It is a function of the lag, of the time between every two samples that are multiplied. The lag between 40 and 120 with the maximum correlation becomes the LTP lag. In the ideal voiced case, that LTP lag will be the distance between two glottal waves, the inverse of the speech's pitch.
The second parameter, the LTP gain, is the maximum correlation divided by the energy of the reconstructed short-term residual signal. (The energy of a discrete signal is the sum of its squared values--its correlation with itself for a lag of zero.) We scale the old wave by this factor to get a signal that is not only similar in shape, but also in loudness. Listing Four shows an example of long-term prediction.

**Residual Signal**

To remove the long-term predictable signal from its input, the algorithm then subtracts the scaled 40 samples. We hope that the residual signal is either weak or random and consequently cheaper to encode and transmit. If the frame recorded a voiced phoneme, the long-term filter will have predicted most of the glottal wave, and the residual signal is weak. But if the phoneme was voiceless, the residual signal is noisy and doesn't need to be transmitted precisely. Because it cannot squeeze 40 samples into only 47 remaining GSM 06.10-encoded bits, the algorithm down-samples by a factor of three, discarding two out of three sample values. We have four evenly spaced 13-value subsequences to choose from, starting with samples 1, 2, 3, and 4. (The first and the last have everything but two values in common.)

The algorithm picks the sequence with the most energy--that is, with the highest sum of all squared sample values. A 2-bit "grid-selection" index transmits the choice to the decoder. That leaves us with 13 3-bit sample values and a 6-bit scaling factor that turns the PCM encoding into an APCM (Adaptive PCM; the algorithm adapts to the overall amplitude by increasing or decreasing the scaling factor).

Finally, the encoder prepares the next LTP analysis by updating its remembered "past output," the reconstructed short-term residual. To make sure that the encoder and decoder work with the same residual, the encoder simulates the decoder's steps until just before the short-term stage; it deliberately uses the decoder's grainy approximation of the past, rather than its own more accurate version.

**The Decoder**

Decoding starts when the algorithm multiplies the 13 3-bit samples by the scaling factor and expands them back into 40 samples, zero-padding the gaps. The resulting residual pulse is fed to the long-term synthesis filter: The algorithm cuts out a 40-sample segment from the old estimated short-term residual signal, scales it by the LTP gain, and adds it to the incoming pulse. The resulting new estimated short-term residual becomes part of the source for the next three predictions.

Finally, the estimated short-term residual signal passes through the short-term synthesis filter whose reflection coefficients the LPC module calculated. The noise or glottal wave from the excited long-term synthesis filter passes through the tubes of the simulated vocal tract--and emerges as speech.

**The Implementation**

Our GSM 06.10 implementation consists of a C library and stand-alone program. While both were originally designed to be compiled and used on UNIX-like environments with at least 32-bit integers, the library has been ported to VMS and MS-DOS. GSM 06.10 is faster than
code-book lookup algorithms such as CELP, but by no means cheap: To use it for real-time communication you'll need at least a medium-scale workstation.

When using the library, you create a `gsm` object that holds the state necessary to either encode frames of 160 16-bit PCM samples into 264-bit GSM frames, or to decode GSM frames into linear PCM frames. (The "native" frame size of GSM, 260 bits, does not fit into an integral number of 8-bit bytes.) If you want to examine and change the individual parts of the GSM frame, you can "explode" it into an array of 70 parameters, change them, and "implode" them back into a packed frame. You can also print an entire GSM frame to a file in human-readable format with a single function call.

We also wrote some throwaway tools to generate the bit-packing and unpacking code for the GSM frames. You can easily change the library to handle new frame formats. We verified our implementation with test patterns from the ETSI. However, since ETSI patterns are not freely available, we aren't distributing them. Nevertheless, we are providing test programs that understand ETSI formats.

The front-end program called "toast" is modeled after the UNIX compress program. Running `toast myspeech`, for example, will compress the file `myspeech`, remove it, and collect the result of the compression in a new file called `myspeech.gsm`; `untoast myspeech` will revert the process, though not exactly--unlike compress, toast loses information with each compression cycle. (After about ten iterations, you can hear high-pitched chirps that I initially mistook for birds outside my office window.)

Listing One is `params.h`, which defines `P_MAX` and `WINDOW` and declares the three functions in Listing Two--`schur()`, `levinson_durbin()`, and `autocorrelation()`--that relate to LPC. Listing Three uses the functions from Listing Two in a short-term transcoder that makes you sound like a "speak-and-spell" machine. Finally, Listing Four offers a plug-in LTP that adds pitch to Listing Three's robotic voice.

For More Information


"Frequently Asked Questions" posting in the USENET comp.speech news group.


Figure 1 Overview of the GSM 6.10 architecture. Figure 2 Reflection coefficients.

Listing One

```c
/* params.h -- common definitions for the speech processing listings. */
#define P_MAX   8   /* order p of LPC analysis, typically 8..14 */
#define WINDOW  160 /* window size for short-term processing   */

double levinson_durbin(double const * ac, double * ref, double * lpc);
double schur(double const * ac, double * ref);
void autocorrelation(int n, double const * x, int lag, double * ac);
```
Listing Two

```c
/* LPC- and Reflection Coefficients
* The next two functions calculate linear prediction coefficients
* and/or the related reflection coefficients from the first P_MAX+1
* values of the autocorrelation function.
*/
#include "params.h"     /* for P_MAX */

/* The Levinson-Durbin algorithm was invented by N. Levinson in 1947
* and modified by J. Durbin in 1959.
*/
double                  /* returns minimum mean square error        */
levinson_durbin(
    double const * ac,  /* in: [0...p] autocorrelation values      */
    double       * ref, /* out: [0...p-1] reflection coefficients   */
    double       * lpc) /*      [0...p-1] LPC coefficients          */
{
    int i, j;  double r, error = ac[0];
    if (ac[0] == 0) {
        for (i = 0; i < P_MAX; i++) ref[i] = 0;
        return 0;
    }
    for (i = 0; i < P_MAX; i++) {
        /* Sum up this iteration's reflection coefficient. */
        r = -ac[i + 1];
        for (j = 0; j < i; j++) r -= lpc[j] * ac[i - j];
        ref[i] = r /= error;
        /* Update LPC coefficients and total error. */
        lpc[i] = r;
        for (j = 0; j < i / 2; j++) {
            double tmp      = lpc[j];
            lpc[j]          = r * lpc[i - 1 - j];
            lpc[i - 1 - j]  += r * tmp;
        }
        if (i % 2) lpc[j] += lpc[j] * r;
        error *= 1 - r * r;
    }
    return error;
}

/* I. Schur's recursion from 1917 is related to the Levinson-Durbin method,
* but faster on parallel architectures; where Levinson-Durbin would take
* time proportional to p * log(p), Schur only requires time proportional to p.
The* GSM coder uses an integer version of the Schur recursion.
*/
double                  /* returns the minimum mean square error        */
schur(
    double const * ac,  /* in: [0...p] autocorrelation values      */
    double       * ref) /* out: [0...p-1] reflection coefficients   */
{
    int i, j;  double r, error = ac[0];
    if (ac[0] == 0) {
        for (i = 0; i < P_MAX; i++) ref[i] = 0;
        return 0;
    }
    for (i = 0; i < P_MAX; i++) {
        /* Sum up this iteration's reflection coefficient. */
        r = -ac[i + 1];
        for (j = 0; j < i; j++) r -= lpc[j] * ac[i - j];
        ref[i] = r /= error;
        /* Update LPC coefficients and total error. */
        lpc[i] = r;
        for (j = 0; j < i / 2; j++) {
            double tmp      = lpc[j];
            lpc[j]          = r * lpc[i - 1 - j];
            lpc[i - 1 - j]  += r * tmp;
        }
        if (i % 2) lpc[j] += lpc[j] * r;
        error *= 1 - r * r;
    }
    return error;
}
```
int i, m; double r, error = ac[0], G[2][P_MAX];

if (ac[0] == 0.0) {
    for (i = 0; i < P_MAX; i++) ref[i] = 0;
    return 0;
}

/* Initialize the rows of the generator matrix G to ac[1...p]. */
for (i = 0; i < P_MAX; i++) G[0][i] = G[1][i] = ac[i + 1];

for (i = 0; ;) {
    /* Calculate this iteration's reflection coefficient and error. */
    ref[i] = r = -G[1][0] / error;
    error += G[1][0] * r;

    if (++i >= P_MAX) return error;

    /* Update the generator matrix. Unlike Levinson-Durbin's summing of */
    /* reflection coefficients, this loop could be executed in parallel */
    /* by p processors in constant time. */
    for (m = 0; m < P_MAX - i; m++) {
        G[1][m] = G[1][m + 1] + r * G[0][m];
        G[0][m] = G[1][m + 1] * r + G[0][m];
    }
}

/* Compute the autocorrelation */
void autocorrelation(
    int   n, double const * x,  /* in: [0...n-1] samples x */
    int lag, double       * ac) /* out: [0...lag-1] autocorrelation */
{
    double d; int i;
    while (lag--) {
        for (i = lag, d = 0; i < n; i++) d += x[i] * x[i-lag];
        ac[lag] = d;
    }
}

Listing Three

/* Short-Term Linear Prediction */
/* To show which parts of speech are picked up by short-term linear */
/* prediction, this program replaces everything but the short-term */
/* predictable parts of its input with a fixed periodic pulse. (You */
/* may want to try other excitations.) The result lacks pitch */
/* information, but is still discernible. */
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include "params.h"    /* See Listing One; #defines WINDOW and P_MAX */
#include "params.h"    /* See Listing One; #defines WINDOW and P_MAX */
/* Default period for a pulse that will feed the short-term processing. The
* length of the period is the inverse of the pitch of the program's
* "robot voice"; the smaller the period, the higher the voice.
*/
#define PERIOD 100          /* human speech: between 16 and 160  */

/* The short-term synthesis and analysis functions below filter and
inverse-
filter their input according to reflection coefficients from Listing
Two.
*/
static void short_term_analysis(
    double const * ref, /* in: [0...p-1] reflection coefficients */
    int n, /* # of samples */
    double const * in, /* [0...n-1] input samples */
    double * out /* out: [0...n-1] short-term residual */
) {
    double sav, s, ui; int i;
    static double u[P_MAX];
    while (n--) {
        sav = s = *in++;
        for (i = 0; i < P_MAX; i++) {
            ui = u[i];
            u[i] = sav;
            sav = ui + ref[i] * s;
            s = s + ref[i] * ui;
        }
        *out++ = s;
    }
}
static void short_term_synthesis(
    double const * ref, /* in: [0...p-1] reflection coefficients */
    int n, /* # of samples */
    double const * in, /* [0...n-1] residual input */
    double * out /* out: [0...n-1] short-term signal */
) {
    double s; int i;
    static double u[P_MAX+1];
    while (n--) {
        s = *in++;
        for (i = P_MAX; i--;) {
            s -= ref[i] * u[i];
            u[i+1] = ref[i] * s + u[i];
        }
        *out++ = u[0] = s;
    }
}

/* This fake long-term processing section implements the "robotic" voice:
* it replaces the short-term residual by a fixed periodic pulse.
*/
static void long_term(double * d) {
    int i; static int r;
    for (i = 0; i < WINDOW; i++) d[i] = 0;
    for (; r < WINDOW; r += PERIOD) d[r] = 10000.0;
    r -= WINDOW;
}

/* Read signed short PCM values from stdin, process them as double,
int main(int argc, char ** argv)
{
    short s[WINDOW]; double d[WINDOW]; int i, n;
    double ac[P_MAX + 1], ref[P_MAX];

    while ((n = fread(s, sizeof(*s), WINDOW, stdin)) > 0) {
        for (i = 0; i < n; i++) d[i] = s[i];
        for (; i < WINDOW; i++) d[i] = 0;

        /* Split input into short-term predictable part and residual. */
        autocorrelation(WINDOW, d, P_MAX + 1, ac);
        schur(ac, ref);
        short_term_analysis(ref, WINDOW, d, d);

        /* Process that residual, and synthesize the speech again. */
        long_term(d);
        short_term_synthesis(ref, WINDOW, d, d);

        /* Convert back to short, and write. */
        for (i = 0; i < n; i++)
            s[i] = d[i] > SHRT_MAX ? SHRT_MAX
                   : d[i] < SHRT_MIN ? SHRT_MIN
                   : d[i];
        if (fwrite(s, sizeof(*s), n, stdout) != n) {
            fprintf(stderr, "%s: write failed\n", *argv);
            exit(1);
        }
        if (feof(stdin)) break;
    }
    if (ferror(stdin)) {
        fprintf(stderr, "%s: read failed\n", *argv); exit(1);
    }
    return 0;
}

Listing Four

/* Long-Term Prediction
 * Here's a replacement for the long_term() function of Listing Three:
 * A "voice" that is based on the two long-term prediction parameters,
 * the gain and the lag. By transmitting very little information,
 * the final output can be made to sound much more natural.
 */
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include "params.h"     /* see Listing One; #defines WINDOW */
#define SUBWIN  40      /* LTP window size, WINDOW % SUBWIN == 0 */

/* Compute n
 * cc(l) = > x(i) * y(i-l)
 * i=0
 * for lags 1 from 0 to lag-1.
 */
static void crosscorrelation(
    int n,        /*  in: # of sample values */
    double const * x,        /* [0...n-1] samples x */
    double const * y,        /* [-lag+1...n-1] samples y */
    int lag,      /*      maximum lag+1 */
    double       * c)        /* out: [0...lag-1] cc values */
{
    while (lag--) {
        int i; double d = 0;
        for (i = 0; i < n; i++)
            d += x[i] * y[i - lag];
        c[lag] = d;
    }
}

/* Calculate long-term prediction lag and gain. */
static void long_term_parameters(
    double const * d,        /*  in: [0.....SUBWIN-1] samples */
    double const * prev,     /*      [-3*SUBWIN+1...0] past signal */
    int          * lag_out,  /* out: LTP lag */
    double       * gain_out) /*      LTP gain */
{
    int      i, lag;
    double   cc[2 * SUBWIN], maxcc, energy;

    /* Find the maximum correlation with lags SUBWIN...3*SUBWIN-1 */
    /* between this frame and the previous ones. */
    crosscorrelation(SUBWIN, d, prev - SUBWIN, 2 * SUBWIN, cc);
    maxcc = cc[lag = 0];

    for (i = 1; i < 2 * SUBWIN; i++)
        if (cc[i] > maxcc) maxcc = cc[lag = i];

    *lag_out = lag + SUBWIN;

    /* Calculate the gain from the maximum correlation and */
    /* the energy of the selected SUBWIN past samples. */
    autocorrelation(SUBWIN, prev - *lag_out, 1, &energy);
    *gain_out = energy ? maxcc / energy : 1.0;
}

/* The "reduce" function simulates the effect of quantizing, */
/* encoding, transmitting, decoding, and inverse quantizing the */
/* residual signal by losing most of its information. For this */
/* experiment, we simply throw away everything but the first sample value. */
static void reduce(double *d) {
    int i;
    for (i = 1; i < SUBWIN; i++) d[i] = 0;
}

void long_term(double * d)
{
    static double     prev[3*SUBWIN];
    double            gain;
    int               n, i, lag;

    for (n = 0; n < WINDOW/SUBWIN; n++, d += SUBWIN) {
        long_term_parameters(d, prev + 3*SUBWIN, &lag, &gain);
    }
The transmission of speech from one point to another over GSM mobile phone network is something that most of us take for granted. The complexity is usually perceived to be associated with the network infrastructure and management required in order to create the end-to-end connection, and not with the transmission of the payload itself. The real complexity, however, lies in the codec scheme used to encode voice traffic for transmission.

The GSM standard supports four different but similar compression technologies to analyse and compress speech. These include full-rate, enhanced full-rate (EFR), adaptive multi-rate (AMR), and half-rate. Despite all being lossy (i.e. some data is lost during the compression), these codecs have been optimized to accurately regenerate speech at the output of a wireless link.

In order to provide toll-quality voice over a GSM network, designers must understand how and when to implement these codecs. To help out, this article provides a look inside how each of these codecs works. We'll also examine how the codecs need to evolve in order to meet the demands of 2.5 and 3G wireless networks.

**Speech Transmission Overview**

When you speak into the microphone on a GSM phone, the speech is converted to a digital signal with a resolution of 13 bits, sampled at a rate of 8 kHz—this 104,000 b/s forms the input signal to all the GSM speech codecs. The codec analyses the voice, and builds up a bit-stream composed of a number of parameters that describe aspects of the voice. The output rate of the codec is dependent on its type (see Table 1), with a range of between 4.75 kbit/s and 13 kbit/s.
After coding, the bits are re-arranged, convoluted, interleaved, and built into bursts for transmission over the air interface. Under extreme error conditions a frame erasure occurs and the data is lost, otherwise the original data is re-assembled, potentially with some errors to the less significant bits. The bits are arranged back into their parametric representation, and fed into the decoder, which uses the data to synthesise the original speech information.

The Full-Rate Codec

The full-rate codec is a regular pulse excitation, long-term prediction (RPE-LTP) linear predictive coder that operates on a 20-ms frame composed of one hundred sixty 13-bit samples.

The vocoder model consists of a tone generator (which models the vocal chords), and a filter that modifies the tone (which models the mouth and nasal cavity shape) [Figure 1]. The short-term analysis and filtering determines the filter coefficients and an error measurement, the long-term analysis quantifies the harmonics of the speech.

<table>
<thead>
<tr>
<th>Codec</th>
<th>Bit Rate (k/sec)</th>
<th>Compression</th>
<th>Codec Type (see text)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full Rate</td>
<td>13</td>
<td>8</td>
<td>RPE-LTP</td>
</tr>
<tr>
<td>Enhanced Full Rate (EFR)</td>
<td>12.2</td>
<td>8.5</td>
<td>ACELP</td>
</tr>
<tr>
<td>Half Rate</td>
<td>5.6</td>
<td>18.4</td>
<td>VSELP</td>
</tr>
<tr>
<td>AMR 12.2</td>
<td>12.2</td>
<td>8</td>
<td>ACELP</td>
</tr>
<tr>
<td>AMR 10.2</td>
<td>10.2</td>
<td>10.2</td>
<td>ACELP</td>
</tr>
<tr>
<td>AMR 7.95</td>
<td>7.95</td>
<td>13.1</td>
<td>ACELP</td>
</tr>
<tr>
<td>AMR 7.4</td>
<td>7.4</td>
<td>14.1</td>
<td>ACELP</td>
</tr>
<tr>
<td>AMR 6.7</td>
<td>6.7</td>
<td>15.5</td>
<td>ACELP</td>
</tr>
<tr>
<td>AMR 5.9</td>
<td>5.9</td>
<td>17.6</td>
<td>ACELP</td>
</tr>
<tr>
<td>AMR 5.15</td>
<td>5.15</td>
<td>20.2</td>
<td>ACELP</td>
</tr>
<tr>
<td>AMR 4.75</td>
<td>4.75</td>
<td>21.9</td>
<td>ACELP</td>
</tr>
</tbody>
</table>

As the mathematical model for speech generation in a full-rate codec shows a gradual decay in power for an increase in frequency, the samples are fed through a pre-emphasis filter.

Figure 1: Diagram of a full-rate vocoder model.
filter that enhances the higher frequencies, resulting in better transmission efficiency. An equivalent de-emphasis filter at the remote end restores the sound.

The short-term analysis (linear prediction) performs autocorrelation and Schur recursion on the input signal to determine the filter ("reflection") coefficients. The reflection coefficients, which are transmitted over the air as eight parameters totalling 36 bits of information, are converted into log area ratios (LARs) as they offer more favourable companding characteristics. The reflection coefficients are then used to apply short term filtering to the input signal, resulting in 160 samples of residual signal.

The residual signal from the short-term filtering is segmented into four sub-frames of 40 samples each. The long-term prediction (LTP) filter models the fine harmonics of the speech using a combination of current and previous sub-frames. The gain and lag (delay) parameters for the LTP filter are determined by cross-correlating the current sub-frame with previous residual sub-frames.

The peak of the cross-correlation determines the signal lag, and the gain is calculated by normalising the cross-correlation coefficients. The parameters are applied to the long-term filter, and a prediction of the current short-term residual is made. The error between the estimate and the real short-term residual signal—the long-term residual signal—is applied to the RPE analysis, which performs the data compression.

The Regular Pulse Excitation (RPE) stage involves reducing the 40 long-term residual samples down to four sets of 13-bit sub-sequences through a combination of interleaving and sub-sampling. The optimum sub-sequence is determined as having the least error, and is coded using APCM (adaptive PCM) into 45 bits.

The resulting signal is fed back through an RPE decoder and mixed with the short-term residual estimate in order to source the long-term analysis filter for the next frame, thereby completing the feedback loop (Table 2).

### Table 2 - Output Parameters from the Full Rate Codec

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Number of parameters</th>
<th>Total bits per frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>LARs</td>
<td>1 per frame</td>
<td>36 bits</td>
</tr>
<tr>
<td>LTP lag</td>
<td>1 per subframe (7 bits)</td>
<td>28 bits</td>
</tr>
<tr>
<td>LTP gain</td>
<td>1 per subframe (2 bits)</td>
<td>8 bits</td>
</tr>
<tr>
<td>RPE grid position</td>
<td>1 per subframe (2 bits)</td>
<td>8 bits</td>
</tr>
<tr>
<td>Block amplitude</td>
<td>1 per subframe (6 bits)</td>
<td>24 bits</td>
</tr>
<tr>
<td>RPE Pulses</td>
<td>13 per subframe (3 bits each)</td>
<td>156 bits</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td><strong>260 bits per frame</strong></td>
</tr>
</tbody>
</table>

**The Enhanced Full-Rate Codec**

As processing power improved and power consumption decreased in digital signal processors (DSPs), more complex codecs could be used to give a better quality of speech. The EFR codec is capable of conveying more subtle detail in the speech, even though the output bit rate is lower than full rate.

The EFR codec is an algebraic code excitation linear prediction (ACELP) codec, which uses a set of similar principles to the RPE-LTP codec, but also has some significant differences. The EFR codec uses a 10th-order linear-predictive (short-term) filter and a long-term filter implemented using a combination of adaptive and fixed codebooks (sets of excitation vectors).
The pre-processing stage for EFR consists of an 80 Hz high-pass filter, and some downscaling to reduce implementation complexity. Short-term analysis, on the other hand, occurs twice per frame and consists of autocorrelation with two different asymmetric windows of 30mS in length concentrated around different sub-frames. The results are converted to short-term filter coefficients, then to line spectral pairs (for better transmission efficiency) and quantized to 38 bits.

In the EFR codec, the adaptive codebook contains excitation vectors that model the long-term speech structure. Open-loop pitch analysis is performed on half a frame, and this gives two estimates of the pitch lag (delay) for each frame.

The open-loop result is used to seed a closed-loop search for speed and reduced computation requirements. The pitch lag is applied to a synthesiser, and the results compared against the non-synthesised input (analysis-by-synthesis), and the minimum perceptually weighted error is found. The results are coded into 34 bits.

The residual signal remaining after quantization of the adaptive codebook search is modelled by the algebraic (fixed) codebook, again using an analysis-by-synthesis approach. The resulting lag is coded as 35 bits per sub-frame, and the gain as 5 bits per sub-frame.

The final stage for the encoder is to update the appropriate memory ready for the next frame.

**Going Adaptive**

The principle of the AMR codec is to use very similar computations for a set of codecs, to create outputs of different rates. In GSM, the quality of the received air-interface signal is monitored and the coding rate of speech can be modified. In this way, more protection is applied to poorer signal areas by reducing the coding rate and increasing the redundancy, and in areas of good signal quality, the quality of the speech is improved.

In terms of implementation, an ACELP coder is used. In fact, the 12.2 kbit/s AMR codec is computationally the same as the EFR codec. For rates lower than 12.2 kbit/s, the
short-term analysis is performed only once per frame. For 5.15 kbit/s and lower, the
open-loop pitch lag is estimated only once per frame. The result is that at lower output
bit rates, there are a smaller number of parameters to transmit, and fewer bits are used
to represent them.

The Half-Rate Codec
The air transmission specification for GSM allows the splitting of a voice channel into two
sub-channels that can maintain separate calls. A voice coder that uses half the channel
capacity would allow the network operators to double the capacity on a cell for very little
investment.

The half-rate codec is a vector sum excitation linear prediction (VSELP) codec that
operates on an analysis-by-synthesis approach similar to the EFR and AMR codecs. The
resulting output is 5.7 kb/s, which includes 100 b/s of mode indicator bits specifying
whether the frames are thought to contain voice or no voice. The mode indicator allows
the codec to operated slightly differently to obtain the best quality.

Half-rate speech coding was first introduced in the mid 1990’s, but the public perception
of speech quality was so poor that it is not generally used today. However, due to the
variable bit-rate output, AMR lends itself nicely to transmission over a half-rate channel.
By limiting the output to the lowest 6 coding rates (4.75 -- 7.95kbps), the user can still
experience the quality benefits of adaptive speech coding, and the network operator
benefits from increased capacity. It is thought that with the introduction of AMR, use of
the half-rate air-channel will start to become much more widespread.

Computational Complexity
Table 3 shows the time taken to encode and decode a random stream of speech-like
data, and the speed of the operations relative to the GSM full-rate codec.

<table>
<thead>
<tr>
<th>Codec</th>
<th>Encoding (mS)</th>
<th>Decoding (mS)</th>
<th>Relative Encode</th>
<th>Relative Decode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full Rate</td>
<td>0.41</td>
<td>0.15</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>Enhanced Full Rate (EFR)</td>
<td>9.02</td>
<td>0.85</td>
<td>22.0</td>
<td>5.4</td>
</tr>
<tr>
<td>Half Rate</td>
<td>8.31</td>
<td>1.30</td>
<td>20.3</td>
<td>8.1</td>
</tr>
<tr>
<td>AMR 12.2</td>
<td>8.99</td>
<td>1.11</td>
<td>21.9</td>
<td>6.9</td>
</tr>
<tr>
<td>AMR 10.2</td>
<td>8.31</td>
<td>1.12</td>
<td>20.3</td>
<td>7.0</td>
</tr>
<tr>
<td>AMR 7.95</td>
<td>8.70</td>
<td>1.07</td>
<td>21.2</td>
<td>6.7</td>
</tr>
<tr>
<td>AMR 7.4</td>
<td>8.10</td>
<td>1.07</td>
<td>19.8</td>
<td>6.7</td>
</tr>
<tr>
<td>AMR 6.7</td>
<td>8.53</td>
<td>1.08</td>
<td>20.8</td>
<td>6.8</td>
</tr>
<tr>
<td>AMR 5.9</td>
<td>7.19</td>
<td>1.44</td>
<td>17.5</td>
<td>9.0</td>
</tr>
<tr>
<td>AMR 5.15</td>
<td>6.40</td>
<td>1.38</td>
<td>15.6</td>
<td>8.6</td>
</tr>
<tr>
<td>AMR 4.75</td>
<td>7.71</td>
<td>1.08</td>
<td>18.8</td>
<td>6.8</td>
</tr>
</tbody>
</table>

The full-rate encoder operates on a non-iterative analysis and filtering, which results in
fast encoding and decoding. By comparison, the analysis-by-synthesis approach
employed in the CELP codecs involves repetitive computation of synthesised speech
parameters. The computational complexity of the EFR/AMR/half-rate codecs is therefore
far greater than the full-rate codec, and is reflected in the time taken to compress and
decompress a frame.

The output of the speech codecs is grouped into parameters (e.g. LARs) as they are
generated (Figure 3). For transmission over the air interface, the bits are rearranged so
the more important bits are grouped together. Extra protection can then be applied to
the most significant bits of the parameters that will have biggest effect on the speech
quality if they are erroneous.
The process of building the air transmission bursts involves adding redundancy to the data by convolution. During this process, the most important bits (Class 1a) are protected most while the least important bits (Class 2) have no protection applied.

This frame building process ensures that many errors occurring on the air interface will be either correctable (using the redundancy), or will have only a small impact on the speech quality.

**Future Outlook**
The current focus for speech codecs is to produce a result that has a perceptually high quality at very low data rates by attempting to mathematically simulate the mechanics of human voice generation. With the introduction of 2.5G and 3G systems, it is likely that two different applications of speech coding will be developed.

The first will be comparatively low bandwidth speech coding, most likely based on the current generation of CELP codecs. Wideband AMR codecs have already been standardised for use with 2G and 2.5G technologies and these will utilise the capacity gains from EDGE deployment.

The second will make more use of the wide bandwidth employing a range of different techniques which will probably be based on current psychoacoustic coding, a technique which is in widespread use today for MP3 audio compression.

There is no doubt that speech quality over mobile networks will improve, but it may be some time before wideband codecs are standardised and integrated with fixed wire-line networks, leading to potentially CD-quality speech communications worldwide.

**About the Authors**
*Richard Meston* is a software engineer at Racal Instruments, working with GSM/GPRS/EDGE and CDMA test equipment. He primarily works GSM mobile and base-station measurement and protocol testers, as well as speech coding and coverage analysis applications. Richard has an Electrical Engineering degree from the University of Sussex and can be reached at richard.meston@racalinstruments.com.