Linear Predictive Speech Coding

10

10.1 OVERVIEW

The linear predictive method of speech analysis approximates the basic parameters of speech. This method is based on the assumption that a speech sample can be approximated as a linear combination of previous speech samples. The application of linear predictive analysis to estimate speech parameters is often called linear predictive coding (LPC).

The LPC method models the production of speech as shown in Figure 10.1. The time-varying digital filter has coefficients that represent the vocal tract parameters. This filter is driven by a function e(t). For voiced speech (sounds created by the vibration of the vocal folds), e(t) is a train of unit impulses at the pitch (fundamental) frequency. For unvoiced speech (sounds generated by the lips, tongue, etc., without vocal-fold vibration), e(t) is random noise with a flat spectrum.

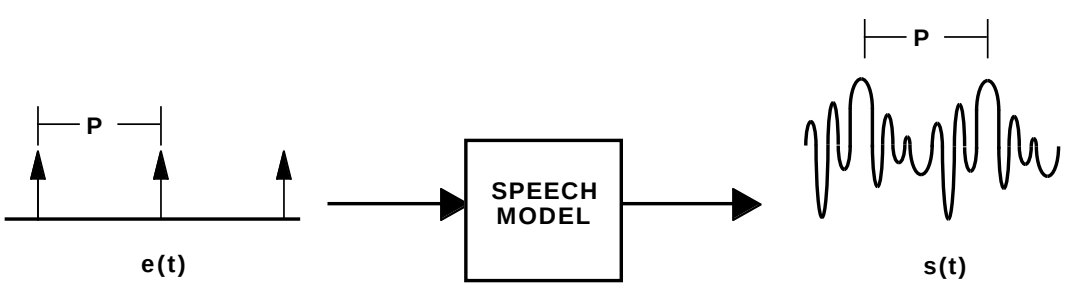

Figure 10.1 Model of Speech Production

The LPC method can be used to create a voice coding system for low-bit-rate transmission. Figure 10.2, on the next page, shows a block diagram of this system. The speech signal is input to the coding system, which derives a set of filter coefficients for the signal and determines whether the signal is voiced. For a voiced signal, the pitch is also calculated. The pitch and filter coefficients are transmitted to a receiving system. This system synthesizes the voice signal by creating a digital filter with the given coefficients and driving the filter with either a train of impulses at the given pitch (for voiced sounds) or a random noise sequence (for unvoiced sounds). The driving function is multiplied by a gain factor, G. For simplicity, in this example we assume unity gain.

10 Linear Predictive Coding

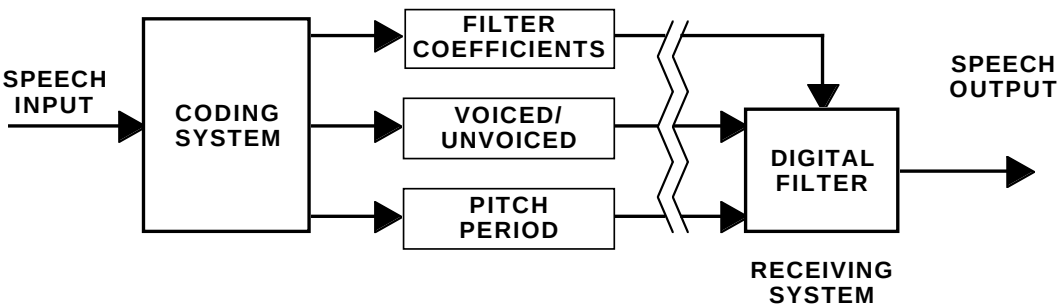

Figure 10.2 Simplified System For Speech Analysis and Synthesis

The coding system predicts the value of an input signal based on the weighted values of the previous P input samples; P is the number of filter coefficients (the order of the synthesis filter). The difference between the actual input and the predicted value is the prediction error. The problem of linear prediction is to determine a set of P coefficients that minimizes the average squared prediction error (E) over a short segment (window) of the input signal. The routines in this chapter are based on a 20-millisecond window, which at a 12-KHz sampling rate yields 240 input samples.

The most efficient method for finding the coefficients is the Levinson-Durbin recursion, which is described by the four equations below. Note that the prime symbol (´) indicates the value to be used in the next pass of the recursion; e.g., E´ is the next value of E.

$$k_i' = [r_s(i) - \sum_{j=1}^{i-1} a_j r_s(i-j)] \div E$$

$$a_i' = k_i'$$

$$a_j' = a_j - k_i' a_{i-j} \qquad j = 1 \text{ to } i-1$$

$$E' = (1-(k_i')^2)\, E$$

where

k_i The negatives of the reflection coefficients used in the LPC synthesis filter (an all-pole lattice filter)

a_i The coefficients used to predict the value of the next input sample

$r_s(i)$ The autocorrelation function of the input signal

E The average squared error between the actual input and predicted input

The autocorrelation function, which is explained further in the next section, is defined as

$$r(k) = \sum_{m=-\infty}^{+\infty} s(m)\, s(m+k)$$

The autocorrelation function of the input signal s(n) is therefore

$$r_s(k) = \sum_{m=0}^{N-1-k} s(m)\, s(m+k)$$

To find the LPC coefficients (k-values), E is initialized to $r_s(0)$. Then the four equations are solved recursively for i = 1 to P. On each pass of the recursion, the first two equations yield another k-value and a-value. In the third equation, all previously calculated a-values are recalculated using the new k-value. The last equation produces a new value for E.

Once all of the k-values have been determined, we can determine whether the input sample is voiced, and if so, what the pitch is. We use the modified autocorrelation analysis algorithm to calculate the pitch using the autocorrelation sequence of the predicted input signal, $r_e(k)$, which can be expressed in terms of the autocorrelation sequence of the actual input and the autocorrelation sequence of the prediction coefficients, a_i.

$$r_e(k) = \sum_{j=1}^{P} r_a(j)\, r_s(k-j)$$

10 Linear Predictive Coding

The autocorrelation function for a_i is defined as

$$r_a(j) = \sum_{i=1}^{P} a_i a_{i+j}$$

The pitch is detected by finding the peak of the normalized autocorrelation sequence ($r_e(n)/r_e(0)$) in the time interval that corresponds to 3 to 15 milliseconds inside the 20-millisecond sampling window. If the value of this peak is at least 0.25, the window is considered voiced with a pitch equal to the value of n at the peak (the pitch period, n_p) divided by the sampling frequency (f_s). If the peak value is less than 0.25, the frame is considered unvoiced and the pitch is zero.

The values of the LPC coefficients (k-values) and the pitch period are transmitted from the coding system to the receiving system. The synthesizer is a lattice filter with coefficients that are the negatives of the calculated k-values. This filter is excited by a signal that is a train of impulses at the pitch frequency. If the pitch is zero, the excitation signal is random noise with a flat spectrum. The excitation function is scaled by the gain value, which is assumed to be one in this example.

The subroutines in this chapter implement linear predictive speech coding. We first present the *correlate* subroutine, which we use whenever a correlation operation (described in the next section) is needed. The *l_p_analysis* subroutine calculates the k-values and the pitch in three main steps. First, the *correlation* subroutine autocorrelates the input signal using the *correlate* subroutine. Next, the *levinson* subroutine finds the k-values using Levinson-Durbin recursion. Last, the *pitch_decision* subroutine determines whether the frame is voiced and computes the pitch period if it is. The *l_p_synthesis* routine, presented at the end of this chapter, generates speech output using the parameters calculated by the *l_p_analysis* routine and the all-pole lattice filter routine presented in Chapter 5.

10.2 CORRELATION

Correlation is an operation performed on two functions of the same variable that are both measures of the same property (voltage, for example). There are two types of correlation functions: cross-correlation and autocorrelation. The cross-correlation of two signals is the sum of the scalar products of the signals in which the signals are displaced in time

with respect to one another. The displacement is the independent variable of the cross-correlation function. Cross-correlation is a measure of the similarity between two signals; it is used to detect time-shifted or periodic similarities. Autocorrelation is the cross-correlation of a signal with a copy of the same signal. It compares the signal with itself, providing information about the time variation of the signal.

The cross-correlation of x(n) with y(n) is described by the equation below. L is the number of samples used for both inputs and L–k–1 is number of "overlapping" samples at the displacement k.

$$R(k) = \sum_{n=0}^{L-k-1} (x(n) \times y(n+k))$$

In autocorrelation, x(n) and y(n) are the same signal.

Correlation is required three times in the computation of the linear prediction coefficients and pitch. First, the input signal must be autocorrelated to determine $r_s(k)$. The first P (= number of k-values) values of $r_s(k)$ are used in the Levinson-Durbin recursion and the rest are used in the pitch determination. Next, the a-values calculated in the Levinson-Durbin recursion are autocorrelated to yield $r_a(k)$. The $r_a(k)$ sequence is then cross-correlated with the autocorrelation sequence of the original input signal, $r_s(k)$, to yield $r_e(k)$, which is used to determine the pitch.

Listing 10.1 shows a correlation routine developed for the ADSP-2100. Before the routine is called, one of the input sequences must be stored in a program memory buffer whose starting address is in register I5. The other input sequence must be stored in a data memory buffer whose starting address is in I1. I6 should point to the start of the result buffer in program memory. I2, which is used as a down counter, must be initialized to the length of the input data buffer (both buffers have the same length), and M2 must be initialized to –1, to allow efficient counter manipulation. The CNTR register should be set with the number of correlation samples desired (N). The SE register, which controls output data scaling, must be set to an appropriate value to shift the products, if necessary, into the desired output format. (For example, if two 4.12 numbers are multiplied, the product is a 7.23 number. To obtain a product in 9.21 format, the SE register must be set to –2.) The modify registers M0, M4, M5, and M6 should all be set to one, and the circular buffer length registers must be set to zero.

The routine executes the *corr_loop* loop to produce the number of correlation samples specified by the CNTR register. Address registers I0 and I4 are set to the starting values of the input data buffers. Each time the loop is executed, I0 fetches the same input data, but the value of I4 is moved forward to fetch the next data sample in the program memory buffer. The CNTR register is then loaded with the length of the multiply/accumulate operation required to produce the current term of the correlation sequence; this length decreases each time the *corr_loop* loop is executed because N–k–1 decreases as k increases. The *data_loop* loop performs the multiply/accumulate operation. The result is then scaled to maintain a valid format. During the scaling operation, the routine takes advantage of multifunction instructions to update various pointers. I5, which points to the start of the program memory buffer, is incremented, and I2, which holds the length of the multiply/accumulate operation for the next loop, is decremented. The values in MX0 and MY0 are extraneous and are overwritten.

```
.MODULE           Correlation;

{                 Correlate Routine

                  Calling Parameters
                     I1 –> Data Memory Buffer              L1 = 0
                     I2 –> Length of Data Buffer           L2 = 0
                     I5 –> Program Memory Buffer           L5 = 0
                     I6 –> Program Memory Result Buffer    L6 = 0
                     M0,M4,M5,M6 = 1      M2 = -1
                     L0,L4 = 0
                     SE = scale value
                     CNTR = output buffer length

                  Return Values
                     Result Buffer Filled

                  Altered Registers
                     I0,I1,I2,I4,I5,I6,MX0,MY0,MR,SR

                  Computation Time
                     Output Length × (Input Length + 8 - ((Output Length - 1) ÷ 2)) + 2
}
```

```
.ENTRY            correlate;

correlate:        DO corr_loop UNTIL CE;
                     I0=I1;
                     I4=I5;
                     CNTR=I2;
                     MR=0, MY0=PM(I4,M4), MX0=DM(I0,M0);
                     DO data_loop UNTIL CE;
data_loop:              MR=MR+MX0*MY0(SS),MY0=PM(I4,M4),MX0=DM(I0,M0);
                     MY0=PM(I5,M5), SR=LSHIFT MR1 (HI);
                     MX0=DM(I2,M2), SR=SR OR LSHIFT MR0 (LO);
corr_loop:           PM(I6,M6)=SR1;
                     RTS;
.ENDMOD;
```

Listing 10.1 Correlation

10.3 LEVINSON-DURBIN RECURSION

The *l_p_analysis* routine, shown in Listing 10.2, calculates the coefficients of the LPC synthesis filter and determines the pitch in approximately 30,000 cycles. This routine calls three other subroutines. First, the *correlation* subroutine autocorrelates the input signal. The *levinson* subroutine uses this autocorrelation sequence to find the LPC coefficients. The *pitch_decision* routine determines the pitch by calling the *pitch_detect* routine, which is presented in the next section.

The subroutines presented in this section calculate the LPC coefficients using the Levinson-Durbin recursion equations:

$$k_i' = [r_s(i) - \sum_{j=1}^{i-1} a_j r_s(i-j)] \div E$$

$$a_i' = k_i'$$

$$a_j' = a_j - k_i' a_{i-j} \qquad j = 1 \text{ to } i-1$$

$$E' = (1-(k_i')^2)\, E$$

The LPC coefficients are the negatives of the k-values.

The *correlation* routine, shown in Listing 10.2, calls the *correlate* routine presented in the previous section to compute the autocorrelation sequence of the input data. The length of the sequence (N) is given by the parameter

wndolength in the constant file, *lpcconst.h*. The autocorrelation of the original input signal takes up the vast majority of the computation time, almost 25,000 clock cycles.

The *levinson* subroutine, also shown in Listing 10.2, uses two data buffers (*a_ping*, *a_pong*) to store the a-values from the previous iteration of the recursion and the new a-values being computed in the current iteration, as necessitated by the third equation in the Levinson-Durbin recursion. The *in_a* pointer points to the start of the input buffer (old a-values) and the *out_a* pointer points to the start of the output buffer (new a-values). The locations of these pointers are swapped at the end of each iteration of the recursion. On the next pass, new values (a_j') become old values (a_j) and the previous old values are overwritten by the newly calculated values.

The *levinson* subroutine calls the *initialize* routine to set up various parameters for the recursion algorithm. The pointers to the a-value buffers (*in_a*, *out_a*) are initialized. The SE register is set to an appropriate scaling value. The last location of the output buffer is found by adding one less than the number of LPC coefficients that will be produced (P–1) to the starting location of the output data buffer; the resulting value is stored in I1. This location is needed because the LPC synthesis routine, presented later in this chapter, uses the coefficients in reverse order, and thus the routine stores the coefficients beginning with the last location of the output buffer. The location at which to store the pitch value is determined by adding P to the starting location of the buffer; this value is stored in SI. The 4.12 fixed-point representation for a one is stored in MF and AR; these values are used to adjust the result in some multiplications. The first term of the autocorrelation sequence of the input signal ($r_s(0)$) is stored as the startup value for the error (E).

The first pass of the recursion algorithm is executed outside of the *recursion* subroutine in the *pass_1* subroutine. Although it performs the same operations as other passes, this pass requires much less processing, since it computes only the first value. The *pass_1* subroutine uses an external *divide* routine (see Chapter 2) to divide the second term of the autocorrelation sequence of the input signal, $r_s(1)$, by the initial value of E, shifted left by three, to yield k_1 in 4.12 format. By Levinson-Durbin recursion, this is also a_1. In this pass, there are no old a-values to recalculate, so all that remains is to determine the new value of E. First, the MR register is loaded with a 7.25 representation of a one (by multiplying registers AR and MF, which were initialized to the necessary values). The squared k_1 value just calculated is subtracted from the MR register value. This difference ($1–k_i^2$) is then multiplied by the old E value

at the same time that a_1 is stored at the location in I0 and the first filter coefficient (negated k_1) is stored at the location in I1. The subroutine finishes by storing the new E value and swapping the *in_a* and *out_a* pointers to the a-value buffers.

The *recursion* subroutine produces the remaining P–1 filter coefficients, one for each iteration of the *durbin* loop. To keep track of loop iterations, the CNTR register is loaded with P–1 and the counting variable *i*, in DM(I), is initialized to a one. Inside the *durbin* loop, pointers to the buffers that contain the a-values and the r_s values are set up. The *loop_1* loop finds the product of a previous a-value and the corresponding $r_s(i–j)$ value and subtracts this product from $r_s(i)$; this operation is performed until all previous i–1 a-values have been used. The result is divided by E to produce the new k_i. The *loop_2* loop recalculates i–1 a-values, one per iteration. It multiplies the old $a_{i–j}$ value from the DM(*in_a*) buffer (pointer in I6) by k_i and subtracts this product from the old a_j value (pointer in I4). The resulting new a-value is stored in the DM(*out_a*) buffer (pointer in I5). The new value of E is calculated in the same way as in the *pass_1* routine. The quantity $(1–k_i^2)$ in MR is multiplied by the old value of E fetched from data memory to produce the new E while a_i and the filter coefficient are being stored. The loop finishes by incrementing the counting variable *i* and swapping the *in_a* and *out_a* pointers to the a-value buffers.

```
.MODULE Predictor;

{  This routine computes the LPC coefficients for the input data.

   Calling Parameters
        I0 –> Input Buffer                                  L0 = 0
        I1 –> Output Buffer                                 L1 = 0
        L2,L3,L4,L5,L6,L7 = 0

   Return Values
        Output buffer filled
        k[10].....k[1], PITCH

   Altered Registers
        I0,I1,I2,I4,I5,I6,M0,M1,M2,M4,M5,M6,M7
        AX0,AY0,AX1,AY1,AR,AF
        MX0,MY0,MX1,MY1,MR,MF
        SI,SE,SR

   Computation Time
        34,000 cycles (approximately)
}
```

(listing continues on next page)

10 Linear Predictive Coding

```
.INCLUDE          <divide.mac>;
.INCLUDE          <lpcconst.h>;

.VAR/DM/RAM       a_ping[p], a_pong[p], dmhold[p];
.VAR/DM/RAM       e, i, in_a, out_a;

.VAR/PM/RAM       hold[length], zeroed[p], r[ptchlength];
.VAR/PM/RAM       ra[p], re[ptchlength];

.EXTERNAL         correlate, pitch_detect;

.INIT             zeroed : <zero.dat>;

.ENTRY            l_p_analysis;

l_p_analysis:   CALL correlation;
                CALL levinson;
                CALL pitch_decision;
                RTS;

correlation:    AY0=I1;I1=I0;M1=1;
                I4=^hold;M5=1;
                CNTR=length;
                DO trans UNTIL CE;        {copy signal into PM}
                   AX0=DM(I0,M1);
trans:             PM(I4,M5)=AX0;
                SE=5;CNTR=ptchlength;     {set parameters for correlate}
                M6=1;M2=-1;M4=1;M0=1;
                I5=^hold;I2=length;I6=^r;
                CALL correlate;
                RTS;

levinson:       CALL initialize;
                CALL pass_1;
                CALL recursion;
                RTS;

pitch_decision: AY0=^r; I0=^a_ping;
                CALL pitch_detect;
                RTS;

initialize:     M0=0;M4=0;M6=-1;          {set up pointers for recursion}
                AX0=^a_ping;DM(in_a)=AX0;
                AX0=^a_pong;DM(out_a)=AX0;
                SE=3;
                AX0=p-1;
                AR=AX0+AY0;
                I1=AR;                    {point to k buffer}
                AX0=p;
                AR=AX0+AY0;
                SI=AR;                    {save pitch pointer}
```

```
                AR=H#1000;MY1=H#8000;
                MF=AR*MY1 (SU);           {MF = formatted one}
                I4=^r;AX0=PM(I4,M5);
                DM(e)=AX0;                {E = r(0)}
                RTS;

pass_1:         AY1=PM(I4,M6);            {compute k}
                SR1=DM(e);
                SR=LSHIFT SR1 (HI);
                AX0=SR1;AY0=SR0;
                divide(AX0,AY1);
                MY0=AY0;
                I0=DM(out_a);
                MX0=AY0, MR=AR*MF (SS);
                MY1=AX0, MR=MR-MX0*MY0 (SS);
                AR=-AY0;
                DM(I0,M0)=AY0, SR=LSHIFT MR1 (HI);    {compute next E}
                DM(I1,M2)=AR, MR=SR1*MY1 (SS);
                DM(e)=MR1;                            {store next E}
                SR1=DM(in_a);
                SR0=DM(out_a);
                DM(out_a)=SR1;
                DM(in_a)=SR0;
                RTS;

recursion:      CNTR=p-1;
                AX1=1;
                DM(i)=AX1;
                AX0=H#1000;
                DO durbin UNTIL CE;
                   I2=DM(in_a);I6=DM(in_a);
                   I4=^r+1;I5=DM(out_a);M7=DM(i);
                   MX0=PM(I4,M7), AR=PASS AX0;
                   MY1=PM(I4,M6);
                   MR=AR*MY1 (SS), MY0=PM(I4,M6), MX0=DM(I2,M1);
                   CNTR=DM(i);
                   DO loop_1 UNTIL CE;          {compute k values}
loop_1:               MR=MR-MX0*MY0 (SS),MX0=DM(I2,M1),MY0=PM(I4,M6);
                   SR=LSHIFT MR1 (HI);
                   AY1=SR1;
                   SR1=DM(e);
                   AY0=SR0, SR=LSHIFT SR1 (HI);
                   AX1=SR1;
                   divide(AX1,AY1);           {divide by E}
                   I4=DM(in_a);
                   MX1=DM(I4,M5);
                   MODIFY(I6,M7);
                   MODIFY(I6,M6);
                   CNTR=DM(i);
                   MY0=AY0;
                   DO loop_2 UNTIL CE;          {compute new a values}
```

(listing continues on next page)

```
                        MR=MX1*MF(SS), MX0=DM(I6,M6);
                        MR=MR-MX0*MY0 (SS);
                        SR=LSHIFT MR1 (HI), MX1=DM(I4,M5);
loop_2:                 DM(I5,M5)=SR1;
                    MY1=DM(e);
                    I6=DM(out_a);
                    MX0=MY0, MR=AR*MF (SS);             {MR = 1}
                    SR0=DM(I6,M7), MR=MR-MX0*MY0 (SS); {MR = 1-k²}
                    AR=-AY0;
                    DM(I1,M2)=AR, SR=LSHIFT MR1 (HI);
                    DM(I6,M6)=AY0, MR=SR1*MY1 (SS);
                    SR=LSHIFT MR1 (HI);
                    DM(e)=SR1;
                    AY0=DM(i);
                    AR=AY0+1;
                    DM(i)=AR;
                    SR1=DM(in_a);
                    SR0=DM(out_a);
                    DM(in_a)=SR0;
durbin:             DM(out_a)=SR1;
                RTS;
.ENDMOD;
```

Listing 10.2 LPC Coefficient Calculation

10.4 PITCH DETECTION

The pitch detection routine is shown in Listing 10.3. Two separate correlation operations (calls to the *correlate* subroutine shown earlier in this chapter) are performed. The first call occurs in the *coeff_corr* routine to autocorrelate the sequence of a-values, which were computed by the Levinson-Durbin recursion routine in the previous section. The second call occurs in the *error_corr* routine which cross-correlates the autocorrelation sequence of the a-values with the autocorrelation of the original input data to calculate the value of $r_e(k)$, as given by the equation:

$$r_e(k) = \sum_{j=1}^{P} r_a(j)\, r_s(j-k) \qquad k = 0 \text{ to } wndolength$$

Because this equation is not a true cross-correlation, the calculation requires a few variations from the normal execution of the *correlate* routine. M4 is set to –1, not the usual 1, to scan the sequence $r_s(n)$ backward instead of forward. To eliminate the possibility of generating errors by using values of $r_s(n)$ in which n is less than zero, P zeros are

appended to the beginning of the $r_s(n)$ data buffer. M2 is set to zero, so that the number of multiplies remains the same instead of decreasing for each execution of the loop.

In the *pitch_period* routine, the sequence $r_e(n)$ is searched over the interval from 3 to 15 milliseconds for the peak value. The starting point of the search is given by *ptchstrt*, which has been set to the sample number that corresponds to 3 milliseconds. The routine first determines whether $r_e(0)$ is positive or negative. If it is negative, the routine jumps to the *nomaxabs* loop, which finds the negative value with the greatest magnitude. If it is positive, the routine jumps to the *nomax* loop, which finds the positive value with the greatest magnitude. After the peak value is found, if its magnitude is greater than $r_e(0)$, then we know that $r_e(\text{peak})/r_e(0)$ must be greater than one; thus, the routine jumps to the *compute* label to compute the pitch. Otherwise, $r_e(\text{peak})$ is divided by $r_e(0)$ to determine if this value is greater than 0.25. Only the first three bits are calculated because more precision is not required. If $r_e(\text{peak})/r_e(0) > 0.25$, the window is considered voiced and the pitch is calculated by multiplying the peak position time value by the sampling period (in *iperiodh* and *iperiodl*), which requires fewer cycles than dividing by the sampling frequency. If $r_e(\text{peak})/r_e(0) \leq 0.25$, the routine returns with a pitch of zero to indicated an unvoiced window.

```
.MODULE     pitching;

{           This routine computes the pitch period for a speech sample.
            It is used in conjunction with a Linear Predictive Coder.

            Calling Parameters
               AY0 —> Autocorrelation buffer
               I0 —> LPC Coefficient buffer     L0 = 0
               SI —> Pitch buffer
               L1,L2,L3,L4,L5,L6,L7 = 0

            Return Values
               Pitch buffer filled

            Altered Registers
               I0,I1,I2,I4,I5,I6,M1,M2,M4,M5,M6,AX0,AX1,AY0,AY1,AR,AF
               MX0,MX1,MY0,MY1,MR,MF,SI,SE,SR

            Computation Time
               approximately 1800 cycles
}
```

(listing continues on next page)

10 Linear Predictive Coding

```
.INCLUDE        <lpcconst.h>;
.VAR/DM/RAM     rahold[p];
.VAR/PM/RAM     hold[p], ra[p], re[ptchlength];
.GLOBAL         pitch_detect;
.EXTERNAL       correlate;

pitch_detect:   CALL coeff_corr;
                CALL error_corr;
                CALL pitch_period;
                RTS;

coeff_corr:     M1=1;M5=1;M2=-1;
                I1=I0;I4=^hold;
                CNTR=p;
                DO move_coeff UNTIL CE;          {copy coeff. to PM}
                   AX0=DM(I0,M1);
move_coeff:        PM(I4,M5)=AX0;
                I5=^hold;CNTR=p;
                M0=1;M4=1;M6=1;I2=p;I6=^ra;
                CALL correlate;
                RTS;

error_corr:     I0=^rahold;I4=^ra;
                CNTR=p;
                DO move_ra UNTIL CE;             {copy r_a to DM}
                   AX0=PM(I4,M5);
move_ra:           DM(I0,M1)=AX0;
                SE=5;I1=^rahold;I5=AY0;CNTR=ptchlength;
                M4=-1;M2=0;I2=P;I6=^re;
                CALL correlate;
                RTS;
pitch_period:   I5=^re;                          {point to r_e}
                M7=ptchstrt;
                MX0=M7;
                AX0=PM(I5,M7);
                AX1=PM(I5,M5), AR=ABS AX0;
                CNTR=wndolength;
                AY0=PM(I5,M5);
                IF POS JUMP max;
                DO nomaxabs UNTIL CE;            {find largest neg. number}
                   AR=AX1-AY0;
                   IF LT JUMP nomaxabs;
                   AX1=AY0;                      {find peak value}
                   AX0=I5;
                   AY0=^re;
                   AF=AX0-AY0;
                   AR=AF-1;
                   MX0=AR;                       {MX0 = peak value}
```

```
nomaxabs:         AY0=PM(I5,M5);
                JUMP pitch_compute;
max:            DO nomax UNTIL CE;
                  AR=AX1-AY0;                    {find peak value}
                  IF GT JUMP nomax;
                  AX1=AY0;
                  AX0=I5;
                  AY0=^re;
                  AF=AX0-AY0;
                  AR=AF-1;
                  MX0=AR;                        {MX0 = peak value}
nomax:            AY0=PM(I5,M5);
pitch_compute:  M0=0;
                I5=^re;
                I1=SI;
                AX0=PM(I5,M5), AR=PASS 0;
                DM(I1,M0)=AR;
                AY0=AR, AR=ABS AX0;
                SR0=AR, AF=ABS AX1;
                AR=SR0-AF;                       {r(0) < r(peak)?}
                IF LE JUMP compute;              {yes, compute pitch}
                AF=PASS AX1;                     {no, find r(peak)÷r(0)}
                AX1=3;
                DIVS AF,AX0;
                DIVQ AX0;
                DIVQ AX0;
                AR=AX1 AND AY0;                  {r(peak)÷r(0) < 0.25?}
                IF EQ JUMP done_compute;         {yes, pitch = 0}
compute:        MY0=iperiodh;                    {no, compute pitch}
                MR=MX0*MY0 (ss);
                MR1=MR0;
                MR0=0;
                MY0=iperiodl;
                MR=MR+MX0*MY0 (su);
                SR=LSHIFT MR1 BY -1 (LO);
                DM(I1,M0)=SR0;
done_compute:   RTS;

.ENDMOD;
```

Listing 10.3 Pitch Detection

10.5 LINEAR PREDICTIVE CODING SYNTHESIZER

The receiving end of the LPC system uses the recursive IIR lattice filter routine presented in Chapter 5 to synthesize an approximation of the original voice signal according to the the LPC voice model. The coefficients of this filter are the negatives of the reflection coefficients (k-values); the k-values were negated and stored in memory in the *levinson* routine. The filter is driven by an excitation function based on the pitch calculated by the *pitch_detect* routine. The gain factor is assumed to be one in this example.

The *l_p_synthesis* routine shown in Listing 10.4 multiplies the pitch by the sampling frequency to yield the pitch period, n_p. It fills the driving function buffer with impulses at the pitch period, and zeros elsewhere. If the window is unvoiced (pitch is zero), the synthesizer uses a driving function of random data in a buffer called *white_noise*. You must initialize this buffer before executing the routine. This can be done using the uniform random number generator presented in Chapter 4.

Various parameters are then set to call the *lattice_filter* subroutine. Note that the length registers L1 and L4 must be set to P, the number of k-values; they are set to zeros after the routine has been executed to ensure that their values do not interfere with any subsequent routines.

```
.MODULE     Synthesizer;

{           Lattice Filter LPC synthesizer

            Calling Parameters
               I1 —> Coefficient Buffer   L0,L1,L2,L3=0
               I2 —> Output Buffer        L4,L5,L6,L7=0

            Return Values
               Output Buffer Filled

            Altered Registers
               I0,I1,I2,I4,M0,M1,M4,M5,M6,M7,AX0,AY0,AX1,AY1,AR,AF
               MX0,MY0,MY1,MR,L1,L4,SE

            Computation Time
               16,000 cycles (approximately)
}

.INCLUDE          <lpcconst.h>;

.VAR/DM           white_noise[length];
.VAR/DM/RAM       pitch_driver[length];
.VAR/PM/RAM/CIRC  delay[p];
```

```
.EXTERNAL         p_latt;
.INIT             delay: <zero.dat>;
.INIT             white_noise: <random.dat>;

.ENTRY            l_p_synthesis;

l_p_synthesis: CALL set_driving;
               CALL synthesis;
               RTS;

set_driving:   M0=1;
               AX0=I1;
               AY0=p;
               AR=AX0+AY0;
               I0=AR;
               AX0=DM(I0,M0);
               I0=^white_noise;          {Point to random data buffer}
               MX0=AX0, AR=PASS AX0;
               IF EQ RTS;                {If pitch = 0, return}
               I0=^pitch_driver;         {Compute pitch period}
               MY0=samplefreq;
               MR=MX0*MY0 (SS);
               CNTR=length;
               AY1=0;
               AY0=MR1, AF=AY1+1;
               AX1=impulse;
               DO fill_buffer UNTIL CE;  {Fill buffer with impulses at}
                  AX0=AY1, AF=AF-1;      {the pitch period}
                  IF NE JUMP fill_buffer;
                  AX0=AX1, AF=AY0+1;
fill_buffer:      DM(I0,M0)=AX0;
               I0=^pitch_driver;
               RTS;

synthesis:     CNTR=length;I4=^delay;          {Set parameters for lattice}
               L1=p;L4=p;M1=-1;M4=1;           {filter routine}
               M5=-1;M6=3;M7=-2;AR=H#1000;
               AX0=p-1;SE=3;
               CAll p_latt;
               L1=0;L4=0;
               RTS;
.ENDMOD;
```

Listing 10.4 LPC Synthesizer

10 Linear Predictive Coding

10.6 REFERENCES

Levinson-Durbin Recursion:
Rabiner, L. R. and Schafer, R. W. 1978. *Digital Processing of Speech Signals*. Englewood Cliffs, N.J.: Prentice-Hall, Inc.

Pitch Detection:
Markel, J. D., and Gray, A. H., Jr. 1980. *Linear Prediction of Speech*. New York: Springer-Verlag.