```
/* SOFT-DECISION VITERBI DECODER                                        */
/* Copyright (c) 1999, 2001 Spectrum Applications, Derwood, MD, USA */
/* All rights reserved                                                  */
/* Version 2.2 Last Modified 2001.11.28                                 */

#include <alloc.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <values.h>

#include "vdsim.h"

#undef SLOWACS
#define FASTACS
#undef NORM
#define MAXMETRIC 128

void deci2bin(int d, int size, int *b);
int bin2deci(int *b, int size);
int nxt_stat(int current_state, int input, int *memory_contents);
void init_quantizer(void);
void init_adaptive_quant(float es_ovr_n0);
int soft_quant(float channel_symbol);
int soft_metric(int data, int guess);

int quantizer_table[256];

void sdvd(int g[2][K], float es_ovr_n0, long channel_length,
            float *channel_output_vector, int *decoder_output_matrix) {

    int i, j, l, ll;                          /* loop variables */
    long t;                                   /* time */
    int memory_contents[K];                   /* input + conv. encoder sr */
    int input[TWOTOTHEM][TWOTOTHEM];          /* maps current/nxt sts to input */
    int output[TWOTOTHEM][2];                 /* gives conv. encoder output */
    int nextstate[TWOTOTHEM][2];              /* for current st, gives nxt given
input */
    int accum_err_metric[TWOTOTHEM][2];       /* accumulated error metrics */
    int state_history[TWOTOTHEM][K * 5 + 1];  /* state history table */
    int state_sequence[K * 5 + 1];            /* state sequence list */

    int *channel_output_matrix;               /* ptr to input matrix */

    int binary_output[2];                     /* vector to store binary enc output */
    int branch_output[2];                     /* vector to store trial enc output */

    int m, n, number_of_states, depth_of_trellis, step, branch_metric,
        sh_ptr, sh_col, x, xx, h, hh, next_state, last_stop; /* misc variables */

/* ******************************************************************** */
```

```
   /* n is 2^1 = 2 for rate 1/2 */
   n = 2;

   /* m (memory length) = K - 1 */
   m = K - 1;

   /* number of states = 2^(K - 1) = 2^m for k = 1 */
   number_of_states = (int) pow(2, m);

   /* little degradation in performance achieved by limiting trellis depth
      to K * 5--interesting to experiment with smaller values and measure
      the resulting degradation. */
   depth_of_trellis = K * 5;

   /* initialize data structures */
   for (i = 0; i < number_of_states; i++) {
       for (j = 0; j < number_of_states; j++)
           input[i][j] = 0;

       for (j = 0; j < n; j++) {
           nextstate[i][j] = 0;
           output[i][j] = 0;
       }

       for (j = 0; j <= depth_of_trellis; j++) {
           state_history[i][j] = 0;
       }

       /* initial accum_error_metric[x][0] = zero */
       accum_err_metric[i][0] = 0;
       /* by setting accum_error_metric[x][1] to MAXINT, we don't need a flag */
       /* so I don't get any more questions about this:  */
       /* MAXINT is simply the largest possible integer, defined in values.h */
       accum_err_metric[i][1] = MAXINT;

   }

   /* generate the state transition matrix, output matrix, and input matrix
      - input matrix shows how FEC encoder bits lead to next state
      - next_state matrix shows next state given current state and input bit
      - output matrix shows FEC encoder output bits given current presumed
      encoder state and encoder input bit--this will be compared to actual
      received symbols to determine metric for corresponding branch of trellis
   */

   for (j = 0; j < number_of_states; j++) {
       for (l = 0; l < n; l++) {
           next_state = nxt_stat(j, l, memory_contents);
           input[j][next_state] = l;

           /* now compute the convolutional encoder output given the current
              state number and the input value */
           branch_output[0] = 0;
           branch_output[1] = 0;
```

```
        for (i = 0; i < K; i++) {
            branch_output[0] ^= memory_contents[i] & g[0][i];
            branch_output[1] ^= memory_contents[i] & g[1][i];
        }

        /* next state, given current state and input */
        nextstate[j][l] = next_state;
        /* output in decimal, given current state and input */
        output[j][l] = bin2deci(branch_output, 2);

    } /* end of l for loop */

} /* end of j for loop */

#ifdef DEBUG
printf("\nInput:");
for (j = 0; j < number_of_states; j++) {
    printf("\n");
    for (l = 0; l < number_of_states; l++)
        printf("%2d ", input[j][l]);
} /* end j for-loop */

printf("\nOutput:");
for (j = 0; j < number_of_states; j++) {
    printf("\n");
    for (l = 0; l < n; l++)
        printf("%2d ", output[j][l]);
} /* end j for-loop */

printf("\nNext State:");
for (j = 0; j < number_of_states; j++) {
    printf("\n");
    for (l = 0; l < n; l++)
        printf("%2d ", nextstate[j][l]);
} /* end j for-loop */
#endif

channel_output_matrix = malloc( channel_length * sizeof(int) );
if (channel_output_matrix == NULL) {
    printf(
    "\nsdvd.c: Can't allocate memory for channel_output_matrix! Aborting...");
    exit(1);
}

/* now we're going to rearrange the channel output so it has n rows,
   and n/2 columns where each row corresponds to a channel symbol for
   a given bit and each column corresponds to an encoded bit */
channel_length = channel_length / n;

/* interesting to compare performance of fixed vs adaptive quantizer */
/* init_quantizer(); */
init_adaptive_quant(es_ovr_n0);
```

```
    /* quantize the channel output--convert float to short integer */
    /* channel_output_matrix = reshape(channel_output, n, channel_length) */
    for (t = 0; t < (channel_length * n); t += n) {
        for (i = 0; i < n; i++)
            *(channel_output_matrix + (t / n) + (i * channel_length) ) =
                soft_quant( *(channel_output_vector + (t + i) ) );
    } /* end t for-loop */

/* ********************************************************************** */

    /* End of setup. Start decoding of channel outputs with forward
        traversal of trellis!  Stop just before encoder-flushing bits.  */
    for (t = 0; t < channel_length - m; t++) {

       if (t <= m)
           /* assume starting with zeroes, so just compute paths from all-zeroes
state */
           step = pow(2, m - t * 1);
        else
            step = 1;

        /* we're going to use the state history array as a circular buffer so
           we don't have to shift the whole thing left after each bit is
           processed so that means we need an appropriate pointer */
        /* set up the state history array pointer for this time t */
        sh_ptr = (int) ( ( t + 1 ) % (depth_of_trellis + 1) );

        /* repeat for each possible state */
        for (j = 0; j < number_of_states; j+= step) {
            /* repeat for each possible convolutional encoder output n-tuple */
            for (l = 0; l < n; l++) {
                branch_metric = 0;

                /* compute branch metric per channel symbol, and sum for all
                    channel symbols in the convolutional encoder output n-tuple */

                #ifdef SLOWACS
                /* convert the decimal representation of the encoder output to binary
*/
                deci2bin(output[j][l], n, binary_output);

                /* compute branch metric per channel symbol, and sum for all
                    channel symbols in the convolutional encoder output n-tuple */
                for (ll = 0; ll < n; ll++) {
                    branch_metric = branch_metric + soft_metric(
*(channel_output_matrix +
                    ( ll * channel_length + t )), binary_output[ll] );

                } /* end of 'll' for loop */
                #endif

                #ifdef FASTACS
                /* this only works for n = 2, but it's fast! */
```

```
                /* convert the decimal representation of the encoder output to binary
*/
                binary_output[0] = ( output[j][l] & 0x00000002 ) >> 1;
                binary_output[1] = output[j][l] & 0x00000001;

                /* compute branch metric per channel symbol, and sum for all
                    channel symbols in the convolutional encoder output n-tuple */
                branch_metric = branch_metric + abs( *( channel_output_matrix +
                    ( 0 * channel_length + t ) ) - 7 * binary_output[0] ) +
                                                abs( *( channel_output_matrix +
                    ( 1 * channel_length + t ) ) - 7 * binary_output[1] );
                #endif

                /* now choose the surviving path--the one with the smaller accumlated
                    error metric... */
                if ( accum_err_metric[ nextstate[j][l] ] [1] > accum_err_metric[j][0]
+
                    branch_metric ) {

                    /* save an accumulated metric value for the survivor state */
                    accum_err_metric[ nextstate[j][l] ] [1] = accum_err_metric[j][0]
+
                        branch_metric;

                    /* update the state_history array with the state number of
                       the survivor */
                    state_history[ nextstate[j][l] ] [sh_ptr] = j;

                } /* end of if-statement */
            } /* end of 'l' for-loop */
        } /* end of 'j' for-loop -- we have now updated the trellis */

        /* for all rows of accum_err_metric, move col 2 to col 1 and flag col 2 */
        for (j = 0; j < number_of_states; j++) {
            accum_err_metric[j][0] = accum_err_metric[j][1];
            accum_err_metric[j][1] = MAXINT;
        } /* end of 'j' for-loop */


        /* now start the traceback, if we've filled the trellis */
        if (t >= depth_of_trellis - 1) {

            /* initialize the state_sequence vector--probably unnecessary */
            for (j = 0; j <= depth_of_trellis; j++)
                state_sequence[j] = 0;

            /* find the element of state_history with the min. accum. error metric */
            /* since the outer states are reached by relatively-improbable runs
               of zeroes or ones, search from the top and bottom of the trellis in */
            x = MAXINT;

            for (j = 0; j < ( number_of_states / 2 ); j++) {
```

```
                if ( accum_err_metric[j][0] < accum_err_metric[number_of_states - 1 -
j][0] ) {
                    xx = accum_err_metric[j][0];
                    hh = j;
                }
                else {
                    xx = accum_err_metric[number_of_states - 1 - j][0];
                    hh = number_of_states - 1 - j;
                }
                if ( xx < x) {
                    x = xx;
                    h = hh;
                }

            } /* end 'j' for-loop */

            #ifdef NORM
            /* interesting to experiment with different numbers of bits in the
               accumulated error metric--does performance decrease with fewer  bits?
*/
            /* if the smallest accum. error metric value is > MAXMETRIC, normalize
the
               accum. errror metrics by subtracting the value of the smallest one
from
               all of them (making the smallest = 0) and saturate all other metrics
               at MAXMETRIC */
            if (x > MAXMETRIC) {
                for (j = 0; j < number_of_states; j++) {
                    accum_err_metric[j][0] = accum_err_metric[j][0] - x;
                    if (accum_err_metric[j][0] > MAXMETRIC)
                        accum_err_metric[j][0] = MAXMETRIC;
                } /* end 'j' for-loop */
            }
            #endif

            /* now pick the starting point for traceback */
            state_sequence[depth_of_trellis] = h;

            /* now work backwards from the end of the trellis to the oldest state
               in the trellis to determine the optimal path. The purpose of this
               is to determine the most likely state sequence at the encoder
                based on what channel symbols we received. */
            for (j = depth_of_trellis; j > 0; j--) {
                sh_col = j + ( sh_ptr - depth_of_trellis );
                if (sh_col < 0)
                   sh_col = sh_col + depth_of_trellis + 1;

                state_sequence[j - 1] = state_history[ state_sequence[j] ] [sh_col];
            } /* end of j for-loop */

            /* now figure out what input sequence corresponds to the state sequence
             in the optimal path */
```

```
            *(decoder_output_matrix + t - depth_of_trellis + 1) =
                input[ state_sequence[0] ] [ state_sequence[1] ];

        } /* end of if-statement */

    } /* end of 't' for-loop */

/* ********************************************************************** */

    /* now decode the encoder flushing channel-output bits */
    for (t = channel_length - m; t < channel_length; t++) {

        /* set up the state history array pointer for this time t */
        sh_ptr = (int) ( ( t + 1 ) % (depth_of_trellis + 1) );

        /* don't need to consider states where input was a 1, so determine
         what is the highest possible state number where input was 0 */
        last_stop = number_of_states / pow(2, t - channel_length + m);

        /* repeat for each possible state */
        for (j = 0; j < last_stop; j++) {

            branch_metric = 0;
            deci2bin(output[j][0], n, binary_output);

            /* compute metric per channel bit, and sum for all channel bits
                in the convolutional encoder output n-tuple */
            for (ll = 0; ll < n; ll++) {
                branch_metric = branch_metric + soft_metric( *(channel_output_matrix
+
                (ll * channel_length + t)), binary_output[ll] );
            } /* end of 'll' for loop */

            /* now choose the surviving path--the one with the smaller total
                metric... */
            if ( (accum_err_metric[ nextstate[j][0] ][1] > accum_err_metric[j][0] +
                branch_metric) /*|| flag[ nextstate[j][0] ] == 0*/) {

                /* save a state metric value for the survivor state */
                accum_err_metric[ nextstate[j][0] ][1] = accum_err_metric[j][0] +
                    branch_metric;

                /* update the state_history array with the state number of
                    the survivor */
                state_history[ nextstate[j][0] ][sh_ptr] = j;

            } /* end of if-statement */

        } /* end of 'j' for-loop */

        /* for all rows of accum_err_metric, swap columns 1 and 2 */
        for (j = 0; j < number_of_states; j++) {
            accum_err_metric[j][0] = accum_err_metric[j][1];
```

```
            accum_err_metric[j][1] = MAXINT;
        } /* end of 'j' for-loop */

        /* now start the traceback, if i >= depth_of_trellis - 1*/
        if (t >= depth_of_trellis - 1) {

            /* initialize the state_sequence vector */
            for (j = 0; j <= depth_of_trellis; j++) state_sequence[j] = 0;

            /* find the state_history element with the minimum accum. error metric */
            x = accum_err_metric[0][0];
            h = 0;
            for (j = 1; j < last_stop; j++) {
                if (accum_err_metric[j][0] < x) {
                    x = accum_err_metric[j][0];
                    h = j;
                } /* end if */
            } /* end 'j' for-loop */

            #ifdef NORM
            /* if the smallest accum. error metric value is > MAXMETRIC, normalize
the
               accum. errror metrics by subtracting the value of the smallest one
from
               all of them (making the smallest = 0) and saturate all other metrics
               at MAXMETRIC */
            if (x > MAXMETRIC) {
                for (j = 0; j < number_of_states; j++) {
                    accum_err_metric[j][0] = accum_err_metric[j][0] - x;
                    if (accum_err_metric[j][0] > MAXMETRIC) {
                        accum_err_metric[j][0] = MAXMETRIC;
                    } /* end if */
                } /* end 'j' for-loop */
            }
            #endif

            state_sequence[depth_of_trellis] = h;

            /* now work backwards from the end of the trellis to the oldest state
                in the trellis to determine the optimal path. The purpose of this
                is to determine the most likely state sequence at the encoder
                based on what channel symbols we received. */
            for (j = depth_of_trellis; j > 0; j--) {

                sh_col = j + ( sh_ptr - depth_of_trellis );
                if (sh_col < 0)
                   sh_col = sh_col + depth_of_trellis + 1;

                state_sequence[j - 1] = state_history[ state_sequence[j] ][sh_col];
            } /* end of j for-loop */

            /* now figure out what input sequence corresponds to the
                optimal path */
```

```
            *(decoder_output_matrix + t - depth_of_trellis + 1) =
               input[ state_sequence[0] ][ state_sequence[1] ];

       } /* end of if-statement */
    } /* end of 't' for-loop */

    for (i = 1; i < depth_of_trellis - m; i++)
        *(decoder_output_matrix + channel_length - depth_of_trellis + i) =
            input[ state_sequence[i] ] [ state_sequence[i + 1] ];

    /* free the dynamically allocated array storage area */
    free(channel_output_matrix);

    return;
} /* end of function sdvd */


/* ******************** END OF SDVD FUNCTION ****************************** */
/* this initializes a 3-bit soft-decision quantizer optimized for about 4 dB Eb/No.
*/
void init_quantizer(void) {

    int i;
    for (i = -128; i < -31; i++)
        quantizer_table[i + 128] = 7;
    for (i = -31; i < -21; i++)
        quantizer_table[i + 128] = 6;
    for (i = -21; i < -11; i++)
        quantizer_table[i + 128] = 5;
    for (i = -11; i < 0; i++)
        quantizer_table[i + 128] = 4;
    for (i = 0; i < 11; i++)
        quantizer_table[i + 128] = 3;
    for (i = 11; i < 21; i++)
        quantizer_table[i + 128] = 2;
    for (i = 21; i < 31; i++)
        quantizer_table[i + 128] = 1;
    for (i = 31; i < 128; i++)
        quantizer_table[i + 128] = 0;
}


/* this initializes a quantizer that adapts to Es/No */
void init_adaptive_quant(float es_ovr_n0) {

    int i, d;
    float es, sn_ratio, sigma;

    es = 1;
    sn_ratio = (float) pow(10.0, ( es_ovr_n0 / 10.0 ) );
    sigma =  (float) sqrt( es / ( 2.0 * sn_ratio ) );

    d = (int) ( 32 * 0.5 * sigma );
```

```
    for (i = -128; i < ( -3 * d ); i++)
        quantizer_table[i + 128] = 7;

    for (i = ( -3 * d ); i < ( -2 * d ); i++)
        quantizer_table[i + 128] = 6;

    for (i = ( -2 * d ); i < ( -1 * d ); i++)
        quantizer_table[i + 128] = 5;

    for (i = ( -1 * d ); i < 0; i++)
        quantizer_table[i + 128] = 4;

    for (i = 0; i < ( 1 * d ); i++)
        quantizer_table[i + 128] = 3;

    for (i = ( 1 * d ); i < ( 2 * d ); i++)
        quantizer_table[i + 128] = 2;

    for (i = ( 2 * d ); i < ( 3 * d ); i++)
        quantizer_table[i + 128] = 1;

    for (i = ( 3 * d ); i < 128; i++)
        quantizer_table[i + 128] = 0;
}


/* this quantizer assumes that the mean channel_symbol value is +/- 1,
   and translates it to an integer whose mean value is +/- 32 to address
   the lookup table "quantizer_table". Overflow protection is included. */
 int soft_quant(float channel_symbol) {
    int x;

    x = (int) ( 32.0 * channel_symbol );
    if (x < -128) x = -128;
    if (x > 127) x = 127;

    return(quantizer_table[x + 128]);
}


/* this metric is based on the algorithm given in Michelson and Levesque,
   page 323. */
int soft_metric(int data, int guess) {

    return(abs(data - (guess * 7)));
}


/* this function calculates the next state of the convolutional encoder, given
   the current state and the input data.  It also calculates the memory
   contents of the convolutional encoder. */
int nxt_stat(int current_state, int input, int *memory_contents) {
```

```
    int binary_state[K - 1];            /* binary value of current state */
    int next_state_binary[K - 1];       /* binary value of next state */
    int next_state;                     /* decimal value of next state */
    int i;                              /* loop variable */

    /* convert the decimal value of the current state number to binary */
    deci2bin(current_state, K - 1, binary_state);

    /* given the input and current state number, compute the next state number */
    next_state_binary[0] = input;
    for (i = 1; i < K - 1; i++)
        next_state_binary[i] = binary_state[i - 1];

    /* convert the binary value of the next state number to decimal */
    next_state = bin2deci(next_state_binary, K - 1);

    /* memory_contents are the inputs to the modulo-two adders in the encoder */
    memory_contents[0] = input;
    for (i = 1; i < K; i++)
        memory_contents[i] = binary_state[i - 1];

    return(next_state);
}


/* this function converts a decimal number to a binary number, stored
   as a vector MSB first, having a specified number of bits with leading
   zeroes as necessary */
void deci2bin(int d, int size, int *b) {
    int i;

    for(i = 0; i < size; i++)
        b[i] = 0;

    b[size - 1] = d & 0x01;

    for (i = size - 2; i >= 0; i--) {
        d = d >> 1;
        b[i] = d & 0x01;
    }
}


/* this function converts a binary number having a specified
   number of bits to the corresponding decimal number
   with improvement contributed by Bryan Ewbank 2001.11.28 */
int bin2deci(int *b, int size) {
    int i, d;

    d = 0;

    for (i = 0; i < size; i++)
        d += b[i] << (size - i - 1);
```

```
    return(d);
}
```