Appendix A

Calculation of Linear Error in a Transconductance Amplifier

A fundamental limitation of analog multipliers such as those used in the original implementation of the continuous wavelet transform modulator (Section 4.3.2) is the small linear range of the output with respect to the differential input. This is also one of the primary reasons to consider the use of log-domain filters, with their large-scale linearity, as a replacement for transconductance-C filters (Chapter 3). Transconductance amplifiers operated in subthreshold have inherent limits on linear range. The approximate linear range of a simple transconductance amplifier can be calculated as shown below. Source degeneration and other methods [5] can extend the linear range by a factor of two to perhaps ten. Above-threshold operation extends the linear range because the square-law transistor behavior gives rise to a flatter amplifier transfer function than does the subthreshold exponential behavior, but the range is nevertheless severely limited.

Figure A.1 shows a MOS differential pair as used in a simple transconductance amplifier. The output $I_2 - I_1$ is shown differentially, although normally the output is made single-ended by mirroring I_2 and subtracting I_1 from it.

We use a somewhat simplified MOS subthreshold equation for drain current:

$$I_{ds} = I_0 e^{(\kappa V_g - V_s)/V_t}; \tag{A.1}$$

making other simplifying assumptions, such as perfectly matched input transistors,

$$I_2 - I_1 = \Delta I = I_0 e^{-V_s/V_t} \left(e^{(\kappa V_2)/V_t} - e^{(\kappa V_1)/Vt} \right) \tag{A.2}$$

$$I_2 + I_1 = I_b = I_0 e^{-V_s/V_t} \left(e^{(\kappa V_2)/V_t} + e^{(\kappa V_1)/Vt} \right) \tag{A.3}$$

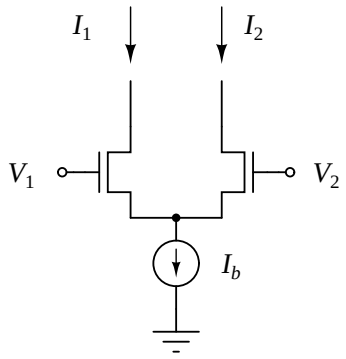

Figure A.1: Simple differential pair transconductance amplifier.

which can be combined to eliminate the common source voltage V_s results in an expression for the output:

$$\Delta I = I_b \left(\frac{e^{(\kappa V_2)/V_t} - e^{(\kappa V_1)/V_t}}{e^{(\kappa V_2)/V_t} + e^{(\kappa V_1)/V_t}} \right). \tag{A.4}$$

As a function of differential input voltage $\Delta V = V_2 - V_1$,

$$\Delta I = I_b \left(\frac{1 - e^{-(\kappa \Delta V)/V_t}}{1 + e^{-(\kappa \Delta V)/V_t}} \right). \tag{A.5}$$

This is a sigmoidal function, with odd symmetry about $\Delta V = 0$, with asymptotes at $\pm I_b$. To find linear error, first find the linear fit to this function at the origin:

$$\left. \frac{\partial \Delta I}{\partial \Delta V} \right|_{\Delta V = 0} = \frac{1}{2} \frac{\kappa}{V_t} I_b \tag{A.6}$$

such that a linear approximation to the output is

$$I_{lin} = \left(\frac{1}{2} \frac{\kappa}{V_t} I_b \right) \Delta V. \tag{A.7}$$

Linear error is

$$\frac{I_{lin} - \Delta I}{\Delta I} = \frac{\kappa \Delta V}{2 V_t} \left(\frac{1 + e^{-(\kappa \Delta V)/V_t}}{1 - e^{-(\kappa \Delta V)/V_t}} \right) - 1. \tag{A.8}$$

This expression is independent of I_b. It is plotted in Figure A.2. From the plot it can be seen that the function is within 10% linear to about ± 50 mV. By approximating the exponents in the above equation, it can be shown that this is a few V_t not by coincidence, but by physical character.

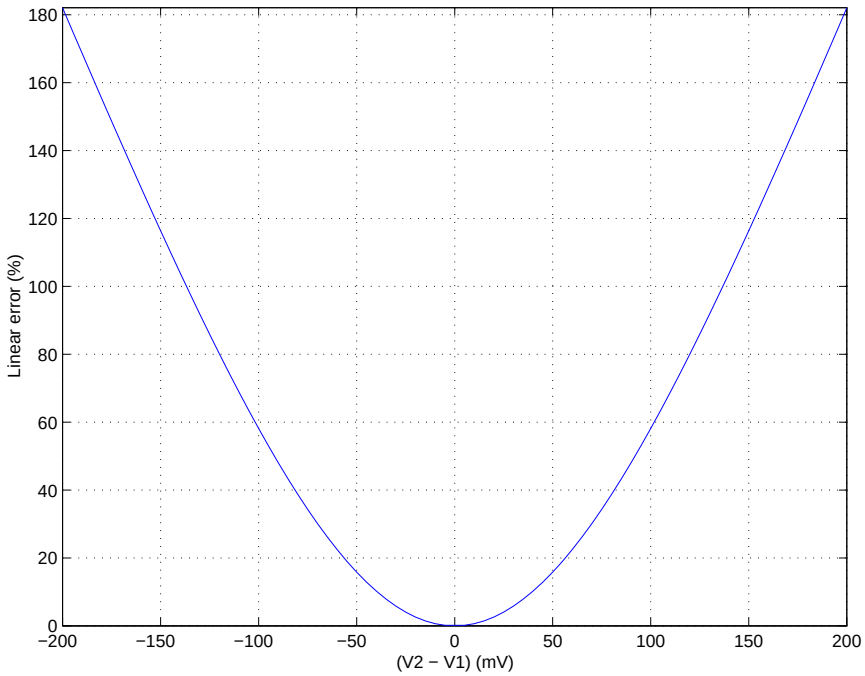

Figure A.2: Linear error in a simple transconductance amplifier as a function of the differential input voltage.

Appendix B

MATLAB Code for Sine Sequence Generation

```
%-------------------------------------------------------------
% sequence.m
%-------------------------------------------------------------
%   Searches for optimal single-bit string approximating a
%   sinusoid.  Algorithm is a stochastic partial search, in
%   blocks of consecutive bits.
%
%   Original Pascal program by Gert Cauwenberghs, 1994
%   Converted to Matlab by Tim Edwards 1998
%-------------------------------------------------------------

%-------------------------------------------------------------
% Settings for this run
%-------------------------------------------------------------

period = 256;   % should be power of 2, range from 4 to 2048
nOpt = 2;       % block length optimized per iteration; <= 16
cap = 15;       % capacitance ratio of filter, Cmax / Cmin
stages = 3;     % number of cascaded 1st-order filter stages

%-------------------------------------------------------------
% Create the lowpass filter function
% This generates attenuation factors in the frequency domain
% for each fft bin
%-------------------------------------------------------------

norm = 2 * pi / period;
```

```
factor = 2 * cap * (cap + 1);

ival = [0: (period / 2) - 1];
singleAtten = 1 ./  (1 + factor * (1 - cos(norm * ival)));
Atten = exp(log(singleAtten) * stages);

figure(1);
semilogy(Atten);
title('Lowpass filter function');

%-------------------------------------------------------------
% Derived values

maxPattern = 2^nOpt - 1;
nBits = period / 4;
xbase = [1: nBits];  % for graphing

% Initialize bit sequence and perturbative sequence

bits = ones(nBits, 1);

% Initial values for iteration

bestSNR = -100;         % in dB, conservative starting point
iter = 0;

%-------------------------------------------------------------
% Iterate:
% Non-terminating loop---just print out results as they come
%-------------------------------------------------------------

while 1,
   iter = iter + 1;

   % Compute optimal SNR:
   %  updates bitsF, assigns a random index, and performs
   %  partial exhaustive optimization with leftmost given
   %  index. Yields the minimum SNR under all combinations.

   maxSNR = -100;
   index = random('Discrete Uniform', nBits, 1, 1);

   maxbits = bits;
```

```
      for pattern = 1: maxPattern,

         % Generate the perturbed bit sequence

         bitsP = bits;

         mask = 1;
         for i = 1: nOpt,
            if bitand(pattern, mask) == mask,
               place = mod(i + index, nBits) + 1;
               bitsP(place) = -bitsP(place);
            end;
            mask = bitshift(mask, 1);
         end;

         % Generate the reverse of the bit sequence bitsP()

         bRev = bitsP(length(bitsP):-1:1);

         % Generate the full sequence from its quarter parts and
         % compute the SNR after filtering with Atten().

         CData= abs(fft([bitsP' bRev' -bitsP' -bRev'], period));
         Raw = CData(1:length(CData)/2);
         Amp = Raw .* Atten;

         % "Signal" is the Fundamental harmonic
         % "Noise" is all the other harmonics

         Signal = Amp(2);
         Noise = sum([Amp(1) Amp(3:length(Amp))]);

         if Signal <= 0,
            Signal = 1e-30;
         end;

         SNRtrial = -20 * log10(Noise / Signal);

         if SNRtrial > maxSNR,
            maxSNR = SNRtrial;
            maxbits = bitsP;
         end;
      end;
```

```
        if  maxSNR > bestSNR,
           bestSNR = maxSNR;
           bits = maxbits;
           fprintf('iteration #%d: SNR = %f\n', iter, bestSNR);
120        fprintf('bit pattern: ');
           for i = 1: nBits,
              fprintf('%1d', (bits(i) > 0));
           end;
           fprintf('\n');
125
           % Plot the FFT outputs for this iteration

           figure(2);
           semilogy(xbase, Raw(1:length(xbase)), 'bx',...
130                 xbase, Amp(1:length(xbase)), 'ro');
        end;
     end;

     %-------------------------------------------------------------
```

B.1 Commentary

This code is completely self-contained, and for the most part self-explanatory. It attempts to find a sequence of n bits, where n is a power of 2, which best represents a sine wave. The bulk of the processing is taken care of by the Matlab built-in Fast Fourier Transform (FFT) in line 93. The iteration loop (line 58) does not terminate but prints out the bit sequence (lines 112 to 120) whenever it reaches a new minimum.

The algorithm, which has no proof of convergence and is not in any way guaranteed to find a global minimum, performs an exhaustive search over a subset of bits within the target sequence. The position of the subsequence is randomly determined (line 67), with the subsequence itself being a loop size of `maxPattern`, calculated from the given number of subsequence bits `nOpt` in line 40. The inner loop exhaustively searching the space of bit patterns in the subsequence (0 to `maxPattern` $-$ 1) comprises lines 71 to 110. Only one quarter of the sequence, `bitsP`, needs to be generated directly from the current best sequence and the current subsequence pattern. The remaining three-quarters of the sequence are formed by reversing the sequence (line 88) and negating both the original and reversed sequences (line 93, where the total sequence is constructed

out of its parts as it is passed to the FFT function).

The lowpass filter function which determines the output after lowpass filtering can be applied in the frequency domain. Knowing that the sequence is periodic and therefore its FFT is discrete, consisting of only the harmonics of the sequence period, it is only necessary to know by what factor the lowpass filter function attenuates each of these harmonics. Thus the filter function may be computed once at each of the harmonics out to some sufficiently high harmonic and stored as array of attenutation factors. This filter computation takes place in lines 20 to 31.

The code could be modified to search for optimal bit patterns representing any arbitrary function by applying an appropriate lowpass filter and comparing the resulting FFT to the FFT of the original function. This has not been attempted, though, and its behavior with regard to becoming trapped at local minima is unknown.

Appendix C

Template Correlation Algorithm with Time Differentiation

The algorithm for template correlation presented in Chapter 4 uses a computation of the pairwise difference between neighboring channels to transform the input and template into a zero-mean representation, from which a binary form may be easily extracted for either or both as required by the implementation.

In addition to frequency channel difference calculations, another way to put the input and template in to an easily quantizable form is a time differentiation. In the time-sampled system, this amounts to taking the difference between successive samples of the input or neighboring time bins of the template. Time differentiation did not prove to be especially effective by itself for acoustic transient classification, but in combination with channel difference computations, it can increase overall system robustness, and may also be useful by itself for certain classification tasks. Its form is especially efficient for hardware implementation, and it is the method used for the first prototypes of the mixed-mode VLSI transient classifier system.

The input is treated in the same manner as it is for the channel differencing system:

$$x[t,m] = \frac{y[t,m]}{\theta + \sum_{k=1}^{M} y[t,k]}, \tag{C.1}$$

where the constant value θ suppresses noise during quiet intervals in the input.

We take the time difference between successive samples of the input on each channel and multiply it by the corresponding template value, which itself is computed from the original template value of the baseline algorithm by taking the difference of values between neighboring time bins. After this step, both input and template have a zero-mean form. We can then quantize the template values by replacing each with its *sign*, indicating whether the energy is expected to be rising or falling at each time sample:

$$c_z[t] = \sum_{m=1}^{M} \sum_{n=1}^{N} (x[t-n,m] - x[t-n-1,m])\, p'_z[n,m] \tag{C.2}$$

where

$$p'_z[n,m] = \text{sign}(p_z[n,m] - p_z[n-1,m]). \tag{C.3}$$

Moving from a positive-valued, magnitude-encoding input and template to a differential form has no effect on classification performance, although it complicates the algorithm by introducing negative values on both sides and requiring four-quadrant multiplications. Rendering each template value binary has a negligible effect on classification performance. Binarization without differentiation severely degrades system performance, and interestingly, system performance suffers dramatically if the input is made binary but the templates are kept continuous valued [56].

When the input is transformed by a time differencing operation, it is possible to rearrange the correlation equation and move the differencing operation so that it becomes the final processing step. By noting that the time difference commutes with the summation, we can write Equation (4.1)

$$\begin{aligned} c_z[t] &= \sum_{m=1}^{M} \sum_{n=1}^{N} x[t-n,m]\, p'_z[n,m] \\ &- \sum_{m=1}^{M} \sum_{n=1}^{N} x[(t-1)-n,m]\, p'_z[n,m]. \end{aligned} \tag{C.4}$$

If we let

$$c'_z[t] = \sum_{m=1}^{M} \sum_{n=1}^{N} x[t-n,m]\, p'_z[n,m], \tag{C.5}$$

then

$$c_z[t] = c'_z[t] - c'_z[t-1]. \tag{C.6}$$

Commuting the time difference operation to the end has several distinct advantages for hardware implementation:

1. We need to compute only one time difference rather than M differences (one for each frequency channel).

2. Architecturally, the algorithm is less affected by device mismatch when computing the output based on the difference of successive outputs rather than the absolute value of the output.

3. Most importantly, since the inputs x are rectified, the product xp' (when p' is binary $[0, 1]$ is always positive and equals either x or zero, which allows us to conveniently implement the entire convolution as an array of simple on/off current switches carrying current in one direction only.

Figure C.1 shows the system as described, where each template value is a single bit controlling a switch (multiplexer) which adds either zero or the unidirectional current input to the sum. The bucket brigade device is responsible for accumulating the summed current over time, and a simple switched capacitor circuit takes the difference at the output.

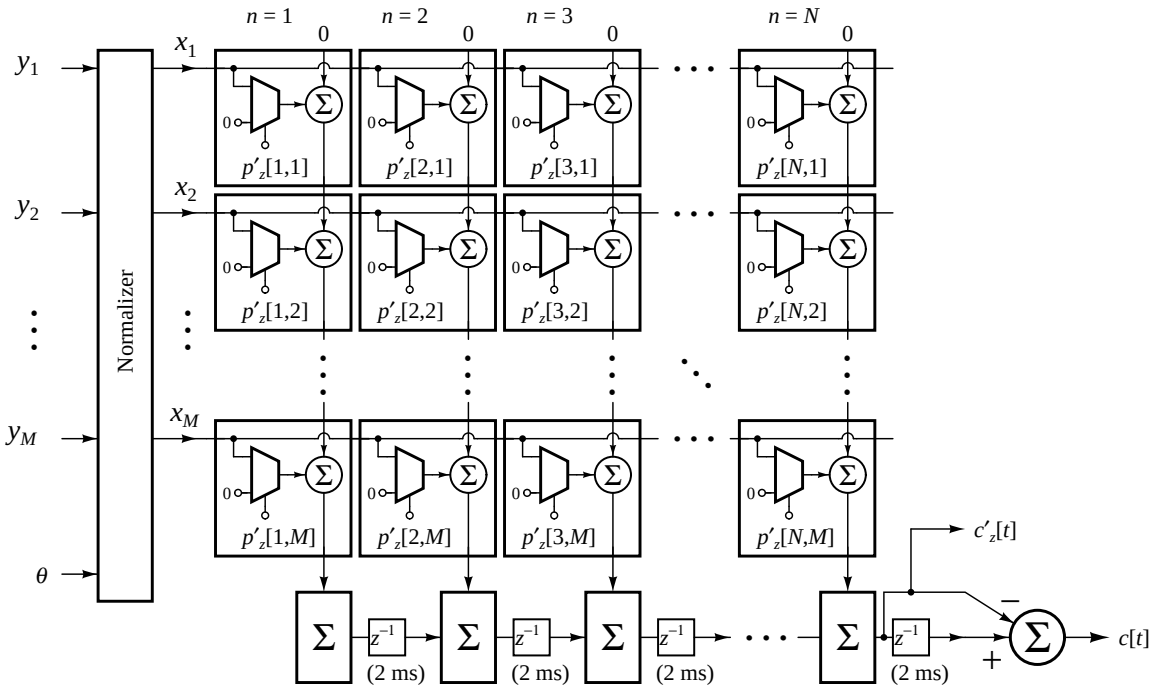

Figure C.1: Block diagram of the temporal current correlator, using time-differencing operations on the input.

C.1 Proof of validity of the pipelined architecture

This is a short proof which verifies that the proposed pipelined architecture produces a correlation result which is exactly equivalent to the result of the non-pipelined correlation.

The pipelined architecture follows the ruleset presented in Chapter 4, Section 4.2.1, and repeated here for clarity:

$$q[n,t] = q[n-1,t-1] + \sum_{m=1}^{M} x'[t,m]\, p'_z[n,m] \qquad \forall n \neq 1 \tag{C.7}$$

$$q[1,t] = \sum_{m=1}^{M} x'[t,m]\, p'_z[1,m] \tag{C.8}$$

$$c_z[t] = q[N,t-1] \tag{C.9}$$

Now expand out the final correlation result $c_z[t]$:

$$c_z[t] = q[N,t-1] \tag{C.10}$$

$$= q[N-1,t-2] + \sum_{m=1}^{M} x'[t-1,m]\, p'_z[N,m] \tag{C.11}$$

$$= q[N-2,t-3] + \sum_{m=1}^{M} x'[t-2,m]\, p'_z[N-1,m] + \sum_{m=1}^{M} x'[t-1,m]\, p'_z[N,m] \tag{C.12}$$

$$\vdots$$

$$= q[1,t-N] + \sum_{m=1}^{M} x'[t-N+1,m]\, p'_z[2,m] + \sum_{m=1}^{M} x'[t-N+2,m]\, p'_z[2,m] + \ldots + \sum_{m=1}^{M} x'[t-1,m]\, p'_z[N,m] \tag{C.13}$$

$$= \sum_{m=1}^{M} x'[t-N,m]\, p'_z[1,m] + \sum_{m=1}^{M} x'[t-N+1,m]\, p'_z[2,m] \tag{C.14}$$

$$+ \sum_{m=1}^{M} x'[t-N+2,m]\, p'_z[3,m] + \ldots + \sum_{m=1}^{M} x'[t-1,m]\, p'_z[N,m] \tag{C.15}$$

Now we can collect the terms back under a summation over n:

$$c_z[t] = \sum_{n=1}^{N} \sum_{m=1}^{M} x'[t-N+n-1,m]\, p'_z[n,m] \tag{C.16}$$

This is the original correlation equation, although the indices for x' and p' do not match up one-to-one; as noted in the text, the template values must be reversed from right to left to make the template for the pipelined architecture equivalent to the original template. That is, $p'_z[n,m]$ becomes $p'_z[N-n+1,m]$ for all n, and

$$c_z[t] = \sum_{n=1}^{N} \sum_{m=1}^{M} x'[t-N+n-1,m]\, p'_z[N-n+1,m] \tag{C.17}$$

With a change of variables $n \rightarrow N-n+1$ (n counts N to 1 rather than 1 to N):

$$c_z[t] = \sum_{n=1}^{N} \sum_{m=1}^{M} x'[t-n,m]\, p'_z[n,m] \tag{C.18}$$

which is the original correlation equation.

The time-differential correlator circuit thus requires only one bucket-brigade device and avoids the problem of matching. Matching is, in fact, quite good due to the differential output which compares the difference between the values on the bucket brigade output node at successive time samples. The differential measurement reduces errors due to systematic offsets in the bucket brigade output. A simple switch-cap circuit which computes the time difference is shown in Figure C.2.

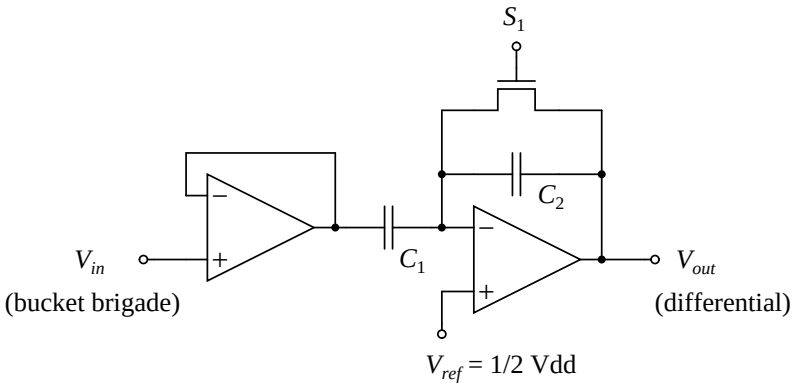

Figure C.2: Switch-cap time-differencing circuit.

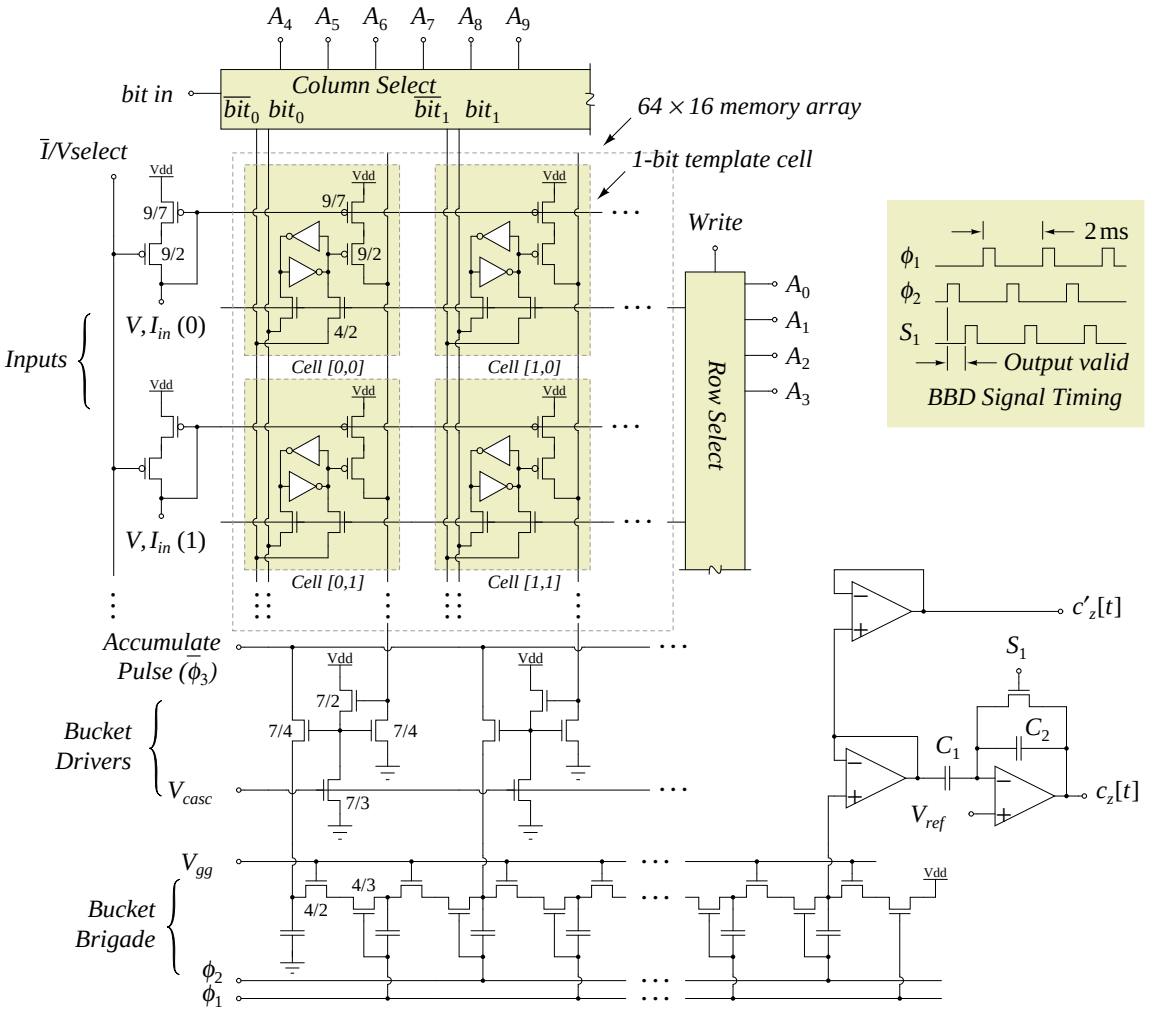

Figure C.3: Time-differencing correlation architecture.

Appendix D

C-code fragment from simulation of the Acoustic Transient Processor frontend filterbank

```
/*-------------------------------------------------------*/
/*  frontend.c                                           */
/*  A modification of the original frontend.f program.   */
/*-------------------------------------------------------*/
/*  This program computes the bandpass filter and        */
/*  detector outputs for PCM input data files.           */
/*  The file determined by the  command line filename    */
/*  is sequentially read for the names of speech files   */
/*  to be converted.                                     */
/*-------------------------------------------------------*/

#include "atp.h"
#include "defines.h"
#include <stdio.h>
#include <fcntl.h>
#include <string.h>
#include <math.h>
#include <sys/types.h>
#include <sys/wait.h>

#define abs(a) ((a) < 0 ? -(a) : (a))
#define max(a,b) ((a) < (b) ? (b) : (a))
#define min(a,b) ((a) < (b) ? (a) : (b))
```

```
#define sign(a) ((a) < 0 ? -1 : 1)

/*-------------------------------------------------------------*/
/*      External variable declarations                         */
/*-------------------------------------------------------------*/
/* itime            = current sample                           */
/* itimen1, itimen2 = last sample                              */
/* filtdata         = bandpass-filtered data                   */
/* lpfiltdata       = time-averaged filter data                */
/* normdata         = peak-detector + time-averaged output     */
/* samprate         = sample rate of input                     */
/* filterfreq       = filter center frequency values           */
/* abcs             = filter coefficients                      */
/* bfiltdata        = intermediate bandpass filter results     */
/* lfiltdata        = intermediate lowpass filter results      */
/* minout, maxout   = peak-detector minimum and maximum        */
/*-------------------------------------------------------------*/

extern long itime, itimen1, itimen2;
extern float *filtdata[NFILTERS];
extern float *lpfiltdata[NFILTERS];
extern float *normdata[DETS];
extern short *shortpcm;
extern int samprate;

float filterfreq[NFILTERS];
float abcs[6][NFILTERS];

float bfiltdata[3][NFILTERS];
float lfiltdata[4][NFILTERS];

float minout[NFILTERS], maxout[NFILTERS];

/*-------------------------------------------------------------*/
/* Function for generating filter center frequencies      */
/*-------------------------------------------------------------*/
/*-------------------------*/
/*  Filter coefficients:   */
/*  Bandpass b0 = abcs[0]  */
/*  Bandpass b2 = abcs[1]  */
/*  Bandpass a1 = abcs[2]  */
/*  Bandpass a2 = abcs[3]  */
/*  Lowpass  b  = abcs[4]  */
/*  Lowpass  a  = abcs[5]  */
```

```
/*-------------------------*/

void makefilters()
{
   short nfilt;
   float X, Y, x1;
   double nn = 1.0 / (double) (NFILTERS - 1);
   double K = (TOPFREQ - BOTFREQ) / STEPFREQ;

   /* The equation for X has no symbolic solution, so     */
   /* compute the coefficient X by iteration.             */
   /* Note that X is generally close to 1 but 1 is also a */
   /* solution of this equation, so we'll start above 1   */
   /* and iterate downward to the solution.               */

   X = 2.0;
   do {
      x1 = X;
      X = (float) exp(nn * log(1.0 + K * (double)(X - 1)));
   } while ((x1 - X) > 0.000001);

   Y = STEPFREQ - BOTFREQ * (X - 1);

   filterfreq[0] = BOTFREQ;

   for (nfilt = 1; nfilt < (NFILTERS + PREPROC); nfilt++)
      filterfreq[nfilt] = filterfreq[nfilt - 1] * X + Y;
}

/*-------------------------------------------------------*/
/* Computer filter coefficients for each frequency band */
/*-------------------------------------------------------*/

void filtinit()
{
   short nfilt;
   float sfreq = SFREQ;
   float Q, denominv, tau, tausq, tauQ;

   makefilters();

   /* for now, fixed-Q filters */
   Q = 5.0;
```

```
      for (nfilt = 0; nfilt < NFILTERS; nfilt++) {

         /* compute filter cutoff and Q value */
115
         tau = samprate / (3.14159 * filterfreq[nfilt]);

         /* compute bandpass filter coefficients */

120      tausq = tau * tau;
         tauQ = tau / Q;
         denominv = 1 / (1 + tauQ + tausq);
         abcs[0][nfilt] = tau * denominv;
         abcs[1][nfilt] = -abcs[0][nfilt];
125      abcs[2][nfilt] = 2 * (tausq - 1) * denominv;
         abcs[3][nfilt] = -(1 - tauQ + tausq) * denominv;

         /* Compute lowpass filter coefficients. "sfreq" */
         /* makes a graded cutoff on a geometric scale   */
130
         tau = samprate / (3.14159 * sfreq);
         abcs[4][nfilt] = 1 / (1 + tau);
         abcs[5][nfilt] = (tau - 1) / (tau + 1);
         sfreq *= FGRAD;
135
         /* initialize filter endpoints */

         bfiltdata[1][nfilt] = bfiltdata[0][nfilt] = 0.0;
         lfiltdata[0][nfilt] = 0.0;
140      lfiltdata[2][nfilt] = 0.0;

         /* Initialize envelope-detection parameters */

         minout[nfilt] = 0;
145      maxout[nfilt] = 0;
      }
   }

   /*-------------------------------------------------------*/
150 /*      Bandpass Filter-bank execution                   */
   /*-------------------------------------------------------*/

   void filterbank()
   {
155   short nfilt;
```

```
      for (nfilt = 0; nfilt < NFILTERS; nfilt++) {

         /* 1st bandpass filter */

         bfiltdata[2][nfilt] = bfiltdata[1][nfilt];
         bfiltdata[1][nfilt] = bfiltdata[0][nfilt];

         bfiltdata[0][nfilt] =
                 abcs[0][nfilt] * (float)shortpcm[itime]
               + abcs[1][nfilt] * (float)shortpcm[itimen2]
               + abcs[2][nfilt] * bfiltdata[1][nfilt]
               + abcs[3][nfilt] * bfiltdata[2][nfilt];

         /* 2nd (cascaded) bandpass filter */

         filtdata[nfilt][itime] =
                 abcs[0][nfilt] * bfiltdata[0][nfilt]
               + abcs[1][nfilt] * bfiltdata[2][nfilt]
               + abcs[2][nfilt] * filtdata[nfilt][itimen1]
               + abcs[3][nfilt] * filtdata[nfilt][itimen2];
      }
   }

   /*-------------------------------------------------------*/
   /* Smooth with a 3rd-order (cascaded) lowpass filter     */
   /*-------------------------------------------------------*/

   makesmooth()
   {
      short nfilt;
      float ntau;

      /* also for now, we don't decimate data down to a new */
      /* timescale based on the Nyquist rate of the data    */
      /* smoothing (i.e., samples at 2ms each)              */

      for (nfilt = 0; nfilt < NFILTERS; nfilt++) {

         lfiltdata[1][nfilt] = lfiltdata[0][nfilt];
         lfiltdata[0][nfilt] = abcs[4][nfilt]
                   * (abs(filtdata[nfilt][itime])
                   + abs(filtdata[nfilt][itimen1]))
                   + abcs[5][nfilt] * lfiltdata[1][nfilt];
```

```
200
         /* filter the output 2x more, for 3rd-order result */

         lfiltdata[3][nfilt] = lfiltdata[2][nfilt];
         lfiltdata[2][nfilt] =
205                 abcs[4][nfilt] * (lfiltdata[0][nfilt]
                  + lfiltdata[1][nfilt]) + abcs[5][nfilt]
                  * lfiltdata[3][nfilt];

         lpfiltdata[nfilt][itime] = abcs[4][nfilt]
210               * (lfiltdata[2][nfilt] +
                  lfiltdata[3][nfilt]) + abcs[5][nfilt] *
                  lpfiltdata[nfilt][itimen1];
      }
   }
215
   /*-------------------------------------------------------*/
   /* Simulate peak-peak detector in hardware version of ATP */
   /*-------------------------------------------------------*/

220 #define TAUSCALE  0.1   /* frequency-to-drop ratio */

   peakdet()
   {

225    for (nfilt = 0; nfilt < NFILTERS; nfilt++) {
         ntau = TAUSCALE * filterfreq[nfilt];

         /* maximum peak find */

230      maxout[nfilt] -= ntau;
         if (filtdata[nfilt][itime] > maxout[nfilt])
            maxout[nfilt] = filtdata[nfilt][itime];

         /* minimum peak finder */
235
         minout[nfilt] += ntau;
         if (filtdata[nfilt][itime] < minout[nfilt])
            minout[nfilt] = filtdata[nfilt][itime];

240      lpfiltdata[nfilt][itime] =
                  maxout[nfilt] - minout[nfilt];
      }
   }
```

```
245  /*-------------------------------------------------------*/
     /* compute L-1 Norm and normalization channel            */
     /*-------------------------------------------------------*/

     computenorm()
250  {
        short nfilt;
        float nsum = NCHAN;

        for (nfilt = 0; nfilt < NFILTERS; nfilt++)
255        nsum += lpfiltdata[nfilt][itime];

        for (nfilt = 0; nfilt < NFILTERS; nfilt++)
           normdata[nfilt][itime] = NORMTO
                     * (lpfiltdata[nfilt][itime] / nsum);
260
        normdata[NFILTERS][itime] = NORMTO * (NCHAN / nsum);
     }

     /*-------------------------------------------------------*/
```

D.1 Commentary

The software simulation of the ATP frontend filterbank processor sufficed for tests of the ATP correlator in the absence of a hardware frontend processor. It is a direct implementation of the intended function of the hardware. Filter transfer functions have been converted from the continuous $j\omega$ frequency domain into the discrete-time z-domain using bilinear transforms.

The channels of the bandpass filterbank can have either a mel-scale or exponential center frequency distribution (90 to 92). Routine `makefilters()` facilitates the computation by determining the frequency spacing from the parameters `TOPFREQ`, `BOTFREQ`, and `STEPFREQ`. The coefficients of the recursion have no symbolic solution and so are determined iteratively (lines 80 to 88).

Routine `filtinit()` (line 98) determines the coefficients of the z-domain IIR filters from the center frequency distribution and the designated constant Q value.

Routine `filterbank()` (line 152) performs a single time sample computation of the filters, retaining the internal values for subsequent computations. In keeping with the hardware spec-

ifications of Chapter 3, Section 3.4, there are two cascaded bandpass filters. They are followed by rectification and smoothing, which can either take the form of the signal full-wave rectifier followed by lowpass filtering (routine `makesmooth()`) or the peak-peak detector (routine `peakdet()`). All outputs are subject to L-1 normalization across all channels in routine `computenorm()`, with the θ value from Equation (4.2) provided by `NCHAN`.

Appendix E

C-code fragment from simulation of the ATP correlator

```
/*-------------------------------------------------------*/
/*                    template classifier                */
/*-------------------------------------------------------*/

#define MIN(a,b) ((a) < (b)) ? (a) : (b)

/*-------------------------------------------------------*/
/* correlate() -- correlates template with sample        */
/*-------------------------------------------------------*/

float correlate(float *template, float *data, int numchans,
        long samples)
{
    float dot = 0.0;
    int i, j;

    for(j = 0; j < samples; j++) {
        for(i = 0; i < numchans - 1; i++) {
            dot += (*template++) * (*data++);
        }
        template++;
        data++;
    }
    return dot/samples;
}
```

```
/*-------------------------------------------------------*/
/* correlate_dt() -- correlates template with data       */
/*-------------------------------------------------------*/

float correlate_dt (float *template, float *data,
        int numchans, long samples)
{
    float dot = 0.0;
    float t0, t1, d0, d1;
    int i, j, bit;

    for (j = 0; j < samples - 1; j++) {
        for (i = 0; i < numchans; i++) {
            t1 = *(template + numchans);
            t0 = *template++;
            d1 = *(data + numchans);
            d0 = *data++;
            dot += ((t1 - t0) > 0) ? (d1 - d0) : 0;
        }
    }
    return dot;
}

/*-------------------------------------------------------*/
/* correlate_old() -- correlates template with data      */
/*-------------------------------------------------------*/
/* This is according to Fernando's original code, but    */
/*  repaired to skip the normalization channel and not to */
/* wrap around from top to bottom channel.               */
/*-------------------------------------------------------*/

float correlate_old(float *template, float *data,
        int numchans, long samples)
{
    float dot = 0.0;
    float t0, t1, d0, d1;
    int i, j;

    for (j = 0; j < samples; j++) {
        for (i = 0; i < numchans - 2; i++) {

            t1 = *(template + 1);
            d1 = *(data + 1);

```

```
                t0 = *template++;
                d0 = *data++;

                dot += (t1 > t0) ? d1 : d0;

/*              dot += (t1 > t0) ? (d1 - d0) : 0;           */
/*              dot += (t1 - t0) * (d1 - d0);               */
            }
            data++;
            template++;
        }
        return dot;
    }

/*---------------------------------------------------------*/
/* correlate_new() --- correlates template w/data          */
/*---------------------------------------------------------*/

float correlate_new(float *template, float *data,
        int numchans, long samples)
{
    float dot = 0.0;
    float t0, t1, t2, d0, d1, d2;
    int i, j;

    for(j = 0; j < samples - 1; j++) {
        for(i = 0; i < numchans - 2; i++) {

            t0 = *(template + numchans);
            t1 = *template++;
            t2 = *(template + numchans);
            d0 = *(data + numchans);
            d1 = *data++;
            d2 = *(data + numchans);
            dot += ((t1 + t2 - 2 * t0) > 0) ?
                        (d2 + d1 - 2 * d0) : 0;
        }
        data += 2;
        template += 2;
    }
    return dot;
}

/*---------------------------------------------------------*/
```

```
115 /* Classification algorithm                                    */
    /*-----------------------------------------------------------*/

    int classify(FILEHEADER *fh, NEWSEGMENT *sh,
            GENERAL_RECORD *record)
120 {
        int class, bestclass;
        float *data, *template, dot;
        long int j, samples;
        int i, numchans;
125     float maxdot;
        class = 0;
        numchans = fh->numchans;
        maxdot = 0;

130     while (templates[class]) {
           data = record->rec.flt;
           template = templates[class];
    /*     samples = MIN(samples, 64);              */
           samples = MIN(sh->samples, template_samples[class]);
135        dotprod[class] = correlate(template, data, numchans,
                             samples);

           if(dotprod[class]>maxdot) {
               maxdot = dotprod[class];
140            bestclass = class;
           }
           class++;
        }
        return bestclass;
145 }

    /*-----------------------------------------------------------*/
```

E.1 Commentary

This code fragment is the central part of the ATP correlator simulations. The entire program, `classify.c`, originally written by Fernando Pineda, takes input from the database generated from the HEEAR chip, and performs a “leave-one-out” cross-validation loop over all of the recorded transient examples.

The correlation is called from routine `classify()`, line 113, which loops over all

classes and determines which class is the “winner” based on the maximum returned dot product. Several versions of the correlation routine have been included here: Tests of variations of the algorithm were performed by uncommenting the necessary parts of the code and calling the correct correlation subroutine from `classify()`.

Line 128 is the original software simulation which allowed templates to be of variable length, depending on the maximum length of the recorded examples of each class. Because the hardware does not allow such flexibility, line 127 is a replacement which more accurately reflects the hardware by limiting the number of time bins in each template to a specific value (shown here set to 64).

The correlation routine at line 11 is the baseline algorithm, based on the direct multiplication of template and input. The routine `correlate_dt()` at line 30 returns a correlation between the binary, time-differenced template and the continuous-valued, time-differenced input. Routine `correlate_old()` at line 56 returns a correlation using channel differences, although several variations are explored in the commented lines (lines 73 and 74). Finally `correlate_new()` (line 86) includes information from both the time difference and channel difference and calculates a binary template value based on a zero-mean representation derived from both.

Appendix F

C-code fragment from the optimizer for per-class gains in the ATP

```
/*-------------------------------------------------------------*/

#ifdef CHANDIFF
#define NCHANS  (channels - 1)
#else
#define NCHANS  channels
#endif

/*-------------------------------------------------------------*/

int channels, timebins, filters, classes;

double ***weights = NULL;  /* weights for reconstruction */
double     *gains = NULL;  /* weights on output vector   */

int quantize;

/*-------------------------------------------------------------*/
/* Normalize the gain coefficients                             */
/*-------------------------------------------------------------*/

void gainnorm()
{
   int i;
   double gsum = 0.0, gscale;
```

```
      for (i = 0; i < filters; i++) {
         gsum += gains[i];
      }
      gscale = (double)filters / gsum;

      /* adapt toward the solution */

      for (i = 0; i < filters; i++) {
         gains[i] *= (0.5 + gscale * 0.5);
/*       gains[i] *= gscale;                        */
         fprintf(stdout, "%8.3f", gains[i]);
      }
      fprintf(stdout, "\n");
}

/*-------------------------------------------------------------*/
/* Print the value of the minimization function.               */
/* return column number with lowest value                      */
/*-------------------------------------------------------------*/

double printerr(double **cgains)
{
   int i, j, k;
   double cval, sum = 0.0, lval;

   for (i = 0; i < classes; i++) {
      lval = gains[i] * cgains[i][i];
      for (j = 0; j < classes; j++) {
         if (j == i) continue;
         sum += exp(gains[j] * cgains[j][i] - lval);
      }
   }
   fprintf(stderr, "%8.3e  ", sum);

   return sum;
}

/*-------------------------------------------------------------*/
/* print out cross-correlations of weight matrices             */
/*-------------------------------------------------------------*/

void printgains(double **cgains)
{
   int i, j, k, t;
```

```
      fprintf(stdout, "Weight matrix cross-correlations\n");

      for (i = 0; i < filters; i++) {
         for (j = 0; j < filters; j++) {
            cgains[i][j] = 0.0;
            for (k = 0; k < channels; k++)
               for (t = 0; t < timebins; t++)
                  switch (quantize) {

                     /* Quantized template and input */
                     case 2:
                       cgains[i][j] += gains[i]
                        * ((weights[j][k][t] > 0) ? .01 : -.01)
                        * ((weights[i][k][t] > 0) ? .01 : -.01);
                       break;

                     /* Quantized template only */
                     case 1:
                       cgains[i][j] += gains[i]
                           * weights[j][k][t]
                           * ((weights[i][k][t] > 0) ? 1 : -1);
                       break;

                     /* Continuous-valued input & template */
                     default:
                       cgains[i][j] += gains[i]
                           * weights[j][k][t]
                           * weights[i][k][t];
                       break;
                  }
            fprintf(stdout, "%5.3f  ", cgains[i][j]);
         }
         fprintf(stdout, "\n");
      }
   }

   /*-------------------------------------------------------------*/
   /* compute relative gain on each template output               */
   /*-------------------------------------------------------------*/

   #define alpha 0.1
   #define ahpla 0.9
```

```
findgains()
{
   int i, j, k, t;
   int bcount;
   double **cgains;
   double tout, newgain, nsum, dsum, ndiff;
   double nngain, nndiff;
   double newerr, lasterr;

   cgains = (double **)malloc(filters * sizeof(double *));
   for (i = 0; i < filters; i++)
      cgains[i] = (double *)malloc(filters * sizeof(double));

   fprintf(stdout, "\nPer-template gain calculations:\n\n");

   for (i = 0; i < filters; i++) {
      gains[i] = 1.0;
   }
   printgains(cgains);
   lasterr = printerr(cgains);

   fprintf(stdout,
      "Computing gains by minimizing cross-correlations.\n");

   for (i = 0; i < filters; i++) {
      gains[i] = 1.0;
   }
   do {
      for (i = 0; i < filters; i++) {
         nngain = 1.0;
         bcount = 0;
         do {
            nsum = 0.0;
            dsum = 0.0;
            for (j = 0; j < filters; j++) {
               if (j == i) continue;
               dsum += exp(gains[j] * cgains[j][i]);
               if (cgains[i][j] > 0)
                  nsum += cgains[i][j]
                        * exp((nngain * cgains[i][j])
                        - (gains[j] * cgains[j][j]));
            }
            dsum *= cgains[i][i];
```

```
            /* move gain in the direction of the optimal */
            /* value by a factor alpha                   */

            newgain = alpha
                 * (-log(nsum / dsum) / cgains[i][i])
                 + ahpla * nngain;
            ndiff = fabs(newgain - nngain);
            nngain = newgain;
            bcount++;
         } while(ndiff > 0.0001 && bcount < 10000);
         if (bcount == 10000) break;

         gains[i] = nngain;
      }
      fprintf(stdout, "\n");

      /* Renormalize the gain coefficients */

      gainnorm();

      newerr = printerr(cgains);
      nndiff = fabs(newerr - lasterr);
      lasterr = newerr;

      if (bcount == 10000) {  /* failed to converge */
         fprintf(stdout,
                "Gain calculation failed to converge.\n");
         for (i = 0; i < filters; i++) {
            gains[i] = 1.0;
         }
         break;
      }

   } while (nndiff > 1e-8);

   printgains(cgains);

   /* free up allocated memory */

   for (i = 0; i < filters; i++)
      free(cgains[i]);
   free(cgains);
}
```

```
/*-------------------------------------------------------*/
```

F.1 Commentary

This is a portion of the code used to independently confirm Fernando Pineda's results using a simulation of the frontend filterbank rather than the HEEAR chip output and operating in continuous time rather than depending on a segmenter, the main point of which was to look for false positives which might occur between presented examples.

This code fragment is the part of the program which determines gains for each class in an attempt to optimize the classification accuracy by maximizing the difference among cross-correlations of the class templates.

The main optimization loop has no proof of convergence and so cannot be guaranteed to find a solution; however, it seems to work well in practice.

Routine `findgains()` is the main iterative loop which seeks to minimize an error function as described in Chapter 4, Section 4.3.6. The error function is computed in lines 141 through 155. The cross-correlations `cgains[i][j]` between templates are computed using the current estimate of the per-channel gains `gains[i]` within the function `printgains()`, lines 66 to 101.

The choice of error function does not compensate for one problem: If all gains are multiplied by a constant, the error measure does not change, which is reasonable since the system accuracy does not change either. This extra degree of freedom allows the adapting gains to slowly drift up or down. The function `gainnorm()`, lines 22 to 38, keeps the gains centered around unity. The routine moves the gains toward the desired mean rather than jumping there in a singe step to avoid throwing the adaptation loop into a limit cycle.

Routine `printerr()` finds a global estimate of the error function which the main loop of the program uses both to report the status of the iterative loop and to automatically determine when the iterative process has converged.