

TECHNICAL DOCUMENTATION FOR MUTIL- CHANNEL COMPRESSION HEARING AIDS FOR HEARING AID APPLICATIONS

Yiya Hao, Ram Charan, Gautam Shreedhar Bhat, Issa Panahi
STATISTICAL SIGNAL PROCESSING LABORATORY (SSPRL)
UNIVERSITY OF TEXAS AT DALLAS

MARCH 2017

This work was supported by the National Institute of the Deafness and Other Communication Disorders (NIDCD) of the National Institutes of Health (NIH) under the award number 1R01DC015430-01. The content is solely the responsibility of the authors and does not necessarily represent the official views of the NIH.

5/2/2017

Table of Contents

STATISTICAL SIGNAL PROCESSING LABORATORY (SSPRL)	1
INTRODUCTION.....	3
1. Pre-Processing	4
1.1 High Pass Filter	4
1.2 RMS	4
2. Gain Function	7
2.1 Prescription Design	7
2.2 Recalculate	8
3. Filter Banks	8
3.1 Multi channel Filter bank.....	8
3.2 Filter banks and Dual channel AGC	9
4. Gain Limiter	11

INTRODUCTION

The Multi-Channel Audio Compression is designed for multi-channel compression for hearing aid in real time. The document provides detailed description of the implementation of this algorithm. The source codes of this document are all written in C/C++ programming language. Figure 1 shows the pipeline of audio compression algorithm. The input signal is *sig* and the output/processed signal is *opwav*. All the source codes can be found in our website.

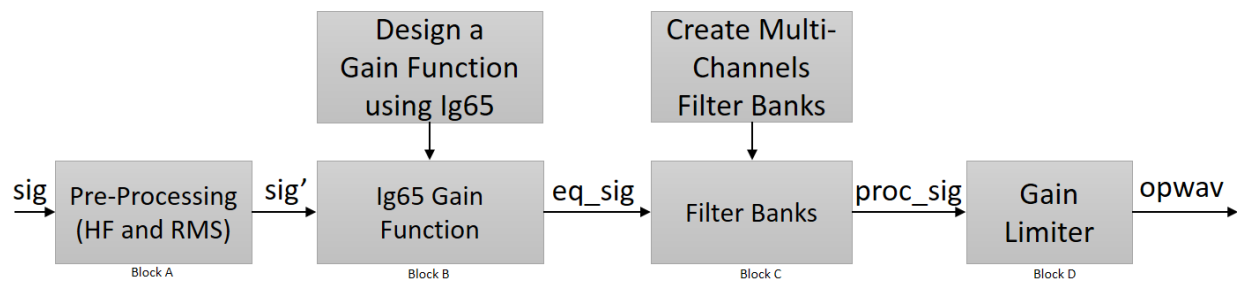


Fig 1. Pipeline of Multi-Channel Audio Compression

- In Block A, there are several pre-processing stages for input signal including an elliptic high pass filter (remove infrasonic) and a levelling based on rms.
- In Block B, there are 10 gains (Insertion gains) corresponding to 10 frequencies (125Hz to 10000Hz) and based on these the input signal is processed by filter.
- In Block C, N-channels AGC (automatic gain control) filter banks which are designed are used to compress. In our source codes we are using 5 channels.
- In Block D, we can adjust the output gain.
- The parameters including *sig*, *eq_sig*, *proc_sig* and *opwav* are same as the parameters in the source code.
- In *Design a gain function using IG65* block, we design the filter to provide the gain coefficients which is used in block B.
- The *Create Multi-Channels filter banks* provide the channel parameters and filter bank coefficients which is used in block C.

1. Pre-Processing

1.1 High Pass Filter

Goal: The High pass filter is an elliptical filter which is mainly used to remove the infrasonic sounds i.e. Frequencies less than 50Hz.

How to apply: For initial processing, *Loadfile* function is called in the main file as shown in the Figure 2. Other Parameters like *framesize* and *Frequency* are also provided along with the input data frame – *sig_frame*.

```
sig_filt_out = m_loadfile(main_parameters->main_variables, sig_frame, framesize, Fs);
```

Figure 2: LoadFile Function

The Elliptic High Pass Filter is of order 3, the *initial_variables* are used for initialization of the frame processing. The *ellip()* function is used to retrieve the predesigned IIR High Pass Elliptic Filter coefficients as shown in the Figure 3.

```
m_variables = initial_variables(FrameSize); // FrameSize Used here

ellip(m_variables,3,0.1,30, 40 * 1.22 / (Fs / 2),3);

iirhp(m_variables->hpfB, m_variables->hpfA, sig, zst, sig_filt, zo,FrameSize);
```

Figure 3: Initialization and IIR High Pass Filtering

1.2 RMS

Goal: In this stage, RMS value of the input audio signal is corrected to *ipfiledigrms*, taking into account the gaps between sounds i.e.by ignoring them. This is presumed to be equivalent to a 65 dB input level.

How to apply: The Input Frame passes through the RMS (Root Mean Square) Stage as shown in the figure 4. The RMS of the Signal is calculated for every RMSFrameLength. Currently this parameter is fixed to evaluate the RMS for every 5 seconds of Data.

```
//Update RMS for every RMSFrameLength
if (RMSFrameCounter == RMSFrameLength - 1)
{
    for (int k = 0; k < framesize; k++)
        sig_filt_rms[RMSFrameCounter * framesize + k] = sig_frame[k];

    RMSFrameCounter = 0;
    RMSUpdate = 1;
}
else
{
    for (int k = 0; k < framesize; k++)
        sig_filt_rms[RMSFrameCounter*framesize + k] = sig_frame[k];
    RMSUpdate = 0;
    RMSFrameCounter++;
}

if (RMSUpdate)
{
    RMSUpdate = 0;

    rms = channel_rms(sig_filt_rms, framesize * RMSFrameLength, Fs, dB_rel_rms);
```

Figure 4: RMS Stage

Channel RMS of the 5 seconds of Input Signal is calculated in 2 stages. Figure 5 shows the 1st stage.

```
for (int i = 0; i < len_s; i++)
    sum_sig_square += pow(sig[i], 2);

first_stage_rms = sqrt(sum_sig_square / len_s);
```

Figure 5: First Stage Channel RMS

To generate 2nd Stage RMS, a parameter *every_dB* is calculated as shown in Figure 6 which is used to construct histogram.

```

for (int ix = 0; ix < a; ix = ix + (int) winlen)
{
    this_sum = 0;
    for (int i = ix; i <= ix + winlen - 1; i++)
    {
        this_sum = this_sum + sig[i] * sig[i]; // calculating sum of squares of signal at instants in a frame.
    }

    every_dB[nframes] = 10 * log10(non_zero + this_sum / winlen);
    nframes = nframes + 1;
}

```

Figure 6: *every_dB* Calculation

A histogram is constructed by sorting the frames in descending order with respect to value of the parameter *every_dB* and only those percentage of frames which have larger *every_dB* is considered for calculation of the Second Stage RMS. This is shown in Figure 7.

```

if (track_percent==1)
{
    inactive_bins = (100 - percent2track)*nframes*.01;

    nlvls = 140;

    double inactive_ix = 0;
    int ixcnt = 0;
    //int k;
    int p;
    int q;

    for (int k = 0; k < 140; k++)
    {
        inactive_ix = inactive_ix + nbins[k];

        if (inactive_ix > inactive_bins)
            break;
        else
        {
            ixcnt = ixcnt + 1;
        }

        p = k;
        q = ixcnt;
    }
}

```

Figure 7: Second Stage Channel RMS

The RMS is used to determine the ratio, which is used to scale the Incoming input signal as shown in Figure 8. All these operations are carried out in block A.

```

file_rms_dB = 20 * log10(rms);

adjustdB = ipfiledigrms - file_rms_dB;

ratio = pow(10, 0.05*adjustdB);

for (int i = 0; i < framesize; i++)
    sig_filt[i] = sig_filt_out[i] * ratio;

```

Figure 8: RMS Scaling of Incoming Signal Frames.

2. Gain Function

2.1 Prescription Design

Goal: In *Design a gain function using IG65* block, the insert gains and the frequencies are processed in the Prescription design function to calculate the gain coefficients which are used in Block B.

How to get the coefficients:

The Gain coefficients are calculated offline by changing the insertion gains in the frequency band which is part of the Prescription Design function. Figure 9 shows the snapshot of the code where the user can change the insertion gains corresponding to the frequencies to get the 132 order filter coefficients.

```

//prescription design function
double *ig_eq;
ig_eq = (double *)malloc(134 * sizeof(double));
float freq[10] = { 125,250,500,1000,2000,3000,4000,6000,8000,10000 };
float gain[10] = { 0,3,5,10,15,20,25,30,30,30 };
int span_msec = 6;
ig_eq = prescription_design(freq, fs, gain, span_msec);

```

Figure 9: Prescription Design Function

These gain coefficients are then used in the recalculate stage in the real-time pipeline.

2.2 Recalculate

Goal: Based on the gain coefficients calculated in the prescription design stage, the input frame processed by filter.

How to apply: The scaled input from block A is then fed to the block B (*recalculate* stage), which performs FIR Filtering using *IG_EQ_Filter* FIR Coefficients which are predetermined in Prescription Design stage as shown in figure 10.

```
double *m_recalculate(double *sig, int len_s)
{
    double *eq_sig = firFrameFilt(sig, IG_EQ_Filter, RecalFilterBuffer, len_s, RecalFilterSize);
    return eq_sig;
}
```

Figure 10: Recalculate Stage

3. Filter Banks

3.1 Multi-channel Filter bank

Goal: To design 5 Channel FilterBank and Obtain Channel Specific DualChannelAGC Parameters like *dig_chan_lvl_0dBgain* and *dig_chan_dBthrs and recombdB*

How to apply: Using the coefficients of prescription design and the Band Pass filter coefficients, additional Dual Channel AGC parameters which are specific to each channel are calculated. These parameters are calibrated using the *calibratefilterbankoverlap* function which is shown in Figure 11.


```

CalibrateFilterBankOverlap *CFBO;
CFBO = newCalibrate( NChans);

CFBO->Calibrate(CFBO, bpfs, Fs, NChans, edges, ig65_eqfir, 133); //bpfs

for (int i = 0; i < 5; i++)
{
    printf(" %1.9g\n", CFBO->chan_reldB_lvls[i]);
    printf(" %1.9g\n", CFBO->BPF_chan_reldB_lvls[i]);
    printf(" %1.9g\n", CFBO->chan_reldB_lvls_postig65[i]);
    printf(" %1.9g\n", CFBO->BPF_chan_reldB_lvls_postig65[i]);
    printf(" %1.9g\n", CFBO->chan_reldB_lvls[i]);
}

for (int i = 0; i < 5; i++)
{
    chan_dBSPL_lvls[i] = 65 + CFBO->chan_reldB_lvls[i];
    spl_chan_thrs[i] = chan_dBSPL_lvls[i] + chan_thrs[i];
    dig_chan_lvl_0dBgain[i] = ipfiledigrms + CFBO->chan_reldB_lvls_postig65[i];
    dig_chan_dBthrs[i] = dig_chan_lvl_0dBgain[i] + chan_thrs[i];
    FBanalysis_dBpost[i] = CFBO->chan_reldB_lvls_postig65[i] - CFBO->BPF_chan_reldB_lvls_postig65[i];
    FBanal_recomb_dBpost[i] = CFBO->chan_reldB_lvls_postig65[i] - CFBO->RecombinedBPF_dBpost[i];
    FBanal_recomb_dBpre[i] = CFBO->chan_reldB_lvls[i] - CFBO->RecombinedBPF_dBpre[i];
    Calib_recomb_dBpost[i] = FBanal_recomb_dBpost[i] - FBanalysis_dBpost[i];
}

```

Figure 11: Calibratefilterbankoverlap Function

The parameters *chan0dBgn_lvl* and *chan_dBthr* are used as *dig_chan_lvl_0dBgain* and *dig_chan_dBthrs* in the Dual Channel AGC Function and *Calib_recomb_dBpost* is used as *recombdB* in the *NChanFBankAGCAid* function.

3.2 Filter banks and Dual channel AGC

Goal: Using preset attack/release times and other specific channel parameters, process the signal in five filter banks.

How to apply:

Based on Desired Simulation Level of Hearing Aid, The Input Signal is rescaled with *re_level*. Figure 12 is the snapshot of the above process.

```

for (int i = 0; i < framesize; i++)
    eg_sig_re[i] = eg_sig[i] * re_level;

```

Figure 12: Rescaling Input Frame with *re_level*

The signal is then processed through a Multi-Channel FilterBanks - *NChanFBankAGCAid* comprised of 5 filters followed by a *DualChannelAGC* in each channel. All the filtered output is time aligned by compensating for the shifts in each frame, this function is shown in figure 13.

```

for (ix = 1; ix <= nchans; ix++)
{
    bpf_len = (int)bpf[ix - 1][0];

    for (j = 2; j <= bpf_len + 1; j++)
    {
        bpf_chan[j - 2] = bpf[ix - 1][j - 1];
    }

#ifdef BRIANMOORE
    int alignFrameLen = (bpf_len_max - bpf_len) / 2;
#endif

    double *sigframeout = FrameBasedFilterBank(signal, bpf_chan, FrameSize, bpf_len, ix - 1);

#ifdef BRIANMOORE
    for (int ii = 0; ii < alignFrameLen; ii++)
        alignFrame[ii] = alignBuffer[ix - 1][ii];

    for (int ii = 0; ii < (FrameSize - alignFrameLen); ii++)
        alignFrame[ii + alignFrameLen] = sigframeout[ii];

    for (int ii = 0; ii < (alignFrameLen); ii++) {
        alignBuffer[ix - 1][ii] = sigframeout[ii + (FrameSize - alignFrameLen)];
    }
#endif
#endif

```

Figure 13: Multi-Channel FilterBanks -*NChanFBankAGCAid* Stage

Dual Channel AGC system uses two different gain control systems to protect the user from uncomfortable loudness produced by brief intense sounds. Based on the ANSI standards, the attack and the release time and the channel parameters, and the limiter parameters, the envelope is weighted by the gain.

```

for (i = 0; i <= framesize; i++)
{
    gain_envp[i] = pow(env[i], expon) * g0dB; //Multiplying with Normalized gain to get 0dB gain from compressor for channel
    //printf("%f ", gain_envp[i]);
}

int *limit = (int*)calloc((framesize + 1), (sizeof(int)));

for (i = 0; i <= framesize; i++)
{
    if (envlim[i] > env[i])
    {
        limit[j] = i;
        //printf(" %.16lf ", limit[j]);
        j++;
    }
}

```

Figure 14: Dual Channel AGC Compressor Envelope

Here is the relevant part of the envelope which is multiplied by the normalized gain. This is shown in figure 14

```
for (i = 0; i < envlim_size; i++)
{
    gain_envp[(int)(limit[i])] = gain_envp[(int)(limit[i])] * (gain_lim[i]); //Limiter Gain Applied
    //printf("%lf ", gain_envp1[1000]);
}

//spc_act = round(10000 * length(limit) / length(gain_envlp)) / 100; %% activity measure, to 0.01% NOT ENCODED
int nshift;
nshift = (int)round((time_shiftmsec*fs) / 1000);
//printf("%d ", nshift);
int siglen;
siglen = framesize;
//printf("%d ", siglen);
double *ip = (double*)calloc((framesize + 1), (sizeof(double)));
```

Figure 15: Dual Channel AGC Limiter Envelope

The channel limiter gain is applied. This operation is done on all 5 channels for an input frame by appropriately applying specific channel parameters. This is shown in in figure 15.

The Output from *DualChannelAGC* is weighted with the parameter *recombdB*, which is based on the channel number and added with rest of the channel outputs. This is shown in figure 16.

```
for (i = 0; i < FrameSize; i++) {
    proc[i] += wx1[i] * pow(10, 0.05 * recombdB[ix - 1]);
    wx1[i] = 0;
}
```

Figure 16: *NChanFBankAGCAid RecombdB* Outputs

4. Gain Limiter

Goal:

Change the volume of output signal to the desired volume based the SPL (sound pressure level).

How to apply:

The output frame is finally multiplied with final gain. Figure 17 shows the gain scaling.

```
final_gain = pow(10, 0.05*(opfiledigrms - ipfiledigrms));

for (int i = 0; i < framesize; i++)
    opwav_frame[i] = proc[i] * final_gain;
```

Figure 17: Final Gain Scaling for the Signal Frame