ETIN80: Algorithms in Signal Processors
Echo Cancellation

Tommy Olofsson
ada09tol@student.lu.se

Gonzalo Blasco Soro
gonzalobsoro@gmail.com

March 4, 2014

1 Problem

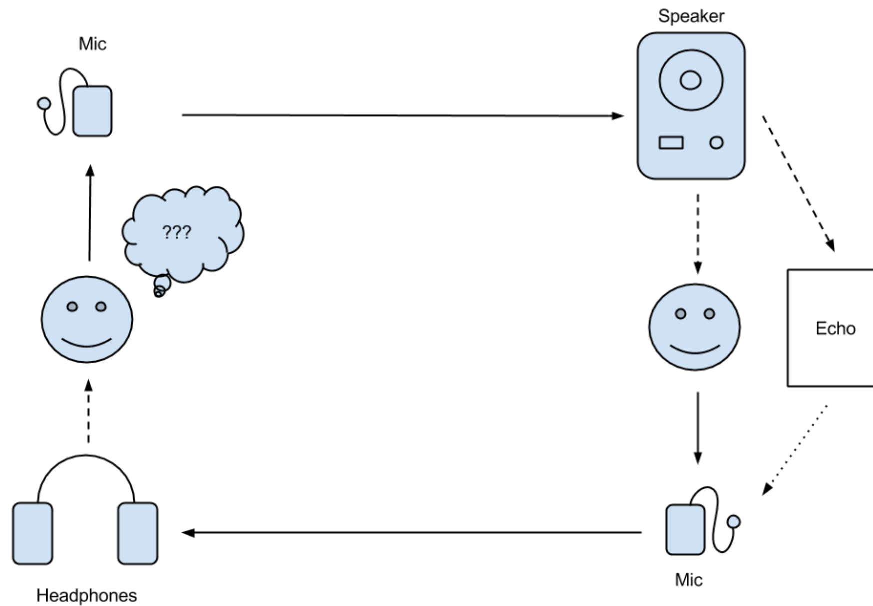

Figure 1: An illustration of the problem of echo in full duplex audio communication systems.

The setup used consists of two sites, each with a microphone. One site has a pair of headphones and one has a large speaker.

The sound produced by the first person will be captured by his microphone and output by the speaker at the location of the second person. As the communicaiton is supposed to go both ways the second person will have another microphone. This microphone will capture not only the words spoken by the second person, but also the echo from the speaker. The signal from this microphone will be output in the headphones of the first person. Due to the distance between the speaker and the microphone the first person will not hear her words as they are spoken, but with a short delay. This will disturb her and make it surpirisingly hard for her to speak properly. The problem is illustrated in figure 1.

2 Solution

In the ideal case the first person would hear only the second person. Therefore, the problem is to cancel the feedback of her voice in the signal from the second site.

This is a common problem encountered in hands-free, mobile phone, and conference phone systems, just to name a few examples. The common solution is to use an adaptive LMS–filter to attenuate the echo. The reason for having an adaptive filter is that even minute changes in the environment in which the system operates will changes its impulse response and thus the echo. For a mobile phone user, this might mean turning ones head or getting into a car.

2.1 Algorithm

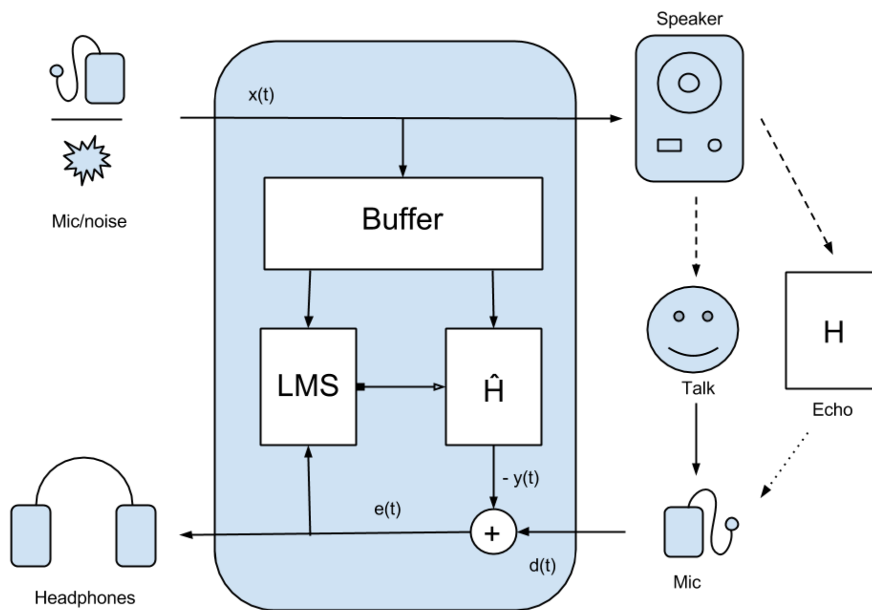

Figure 2: The setup used in the project. For testing purposes white noise can be substituted for the first microphone signal.

An overview of the system is shown in figure 2. The algorithm should strive to remove the echo of signal x, output in the speaker, from signal d, input by the microphone. To accomodate this the signal x is buffered. The length P of the buffer is dictated by maximum delay of the echo one wishes to remove. For each input sample the part which is computed to be a result of echo, y is subtracted from the sample. This is done by approximating the environments impulse response, H, by $\hat{H}$. Practically, this is done by having a coefficient for sample in the buffer. The cleaned up signal is

$$e(n) = d(n) - y(n)$$

where $y(n)$ is computed as

$$y(n) = \hat{H}(n) \cdot X(n)$$

To accomodate a changing environment impulse response, H, the approximation, $\hat{H}$ is refined as often as the hardware will allow. This means adjusting the filter coefficients according to

$$\hat{H}(n+1) = \hat{H}(n) \frac{\mu e(n) X(n)}{X(n)^{\intercal} X(n)}$$

with a certain *learning rate*, denoted μ. To clarify, $\hat{H}$ is the filter coefficients and X is the previous P samples of $x(i)$. The adjustment is normalized.

In this way, a filter starting with filter coefficients set to zero will adapt to the environment automatically. The learning rate was determined experimentally.

For further details regarding the theory, we refer the reader to [1] and page 852– in [2].

2.2 Problems and Discussion

2.2.1 Algorithm

In addition to the constant μ, an additional parameter had to be introduced. When we switched from the plain algorithm to the normalized one we encountered an instability. This was corrected by empirically adding a small constant to the sum of the squares of X in the denominator in the expression for updating $\hat{H}$.

Not quite a problem, but rather a challenge, was the tuning of the filter with regard to length and the learning rate. The filter length was initially set to 200. Using a sample rate of 8kHz this means the buffer covers $8000/200 = 0.025$ seconds, or 25 ms. This should be able to cancel echos with a travel distance of up to $340 * 0.025 = 8.5$ meter. The effectiveness of the filter should theretically be greater at a higher frequency, but no clear difference could be perceived at 16kHz. The ability to perform the adaptation for each sample proved to be of more value and might be part of the explanation, due to the fact that this could not be done at higher sample rates.

2.2.2 Practical

One major source of problems turned out to be the software environment used during development. The VisualDSP++ IDE from Analog Devices turned out to be somewhat unstable.

The problems included:

Seemingly arbitrary halting of execution during debugging
: This could often be temporarily resolved by disconnecting and reconnecting the target in the software. No reason for the behaviour was found.

Random disconnects between DSP and PC
: This could be temporarily resolved by physically disconnecting and reconnecting the ICE to the PC. No reason for the behaviour was found.

Audio processing ISR not being executed
: This behaviour disappeared when the algorithm was rewritten with no obvious errors. This behaviour was most likely due to a silent crash.

2.3 Code

The amount of code developed was surprisingly small. Listing 1 contains the full implementation.

Listing 1: main.c (Line–wrapped to fit document. Originally standard <80 columns.)

```
#include <processor_include.h>
#include <stdlib.h>
#include <signal.h>
#include <stdfix.h>
#include <string.h>
#include <filter.h>
#include <stdio.h>
#include <limits.h>
#include <math.h>

#include "framework.h"


#define DEBUG_STATS 0                        /* Print stats to
    console. */


#define                  P (200)             /* Filter length.
    */
static unsigned int     opt_adapt  = 1;  /* Enable/disable
    adaptation. */
static unsigned int     opt_filter = 1;  /* Enable/disable
    filter. */
static unsigned int     opt_mic    = 0;  /* Mic. or noice.
    */
static float            h_hat[P];        /* Approx. impulse
    response of room. */
static float            xbuff[P];        /* Buffer
    containing past samples. */
static ptrdiff_t        bi;              /* Index of newest
    sample in ibuff. */
```

```
static const pm float mu = 0.250;       /* LMS filter step
    size. */
#if DEBUG_STATS
static float           avg_damp = 0.0; /* Average damping
    factor. */
#endif

static void keyboard(int sig)
{
        unsigned int keys = dsp_get_keys();

        if (keys & 1) opt_filter = 1;
        if (keys & 2) opt_filter = 0;

        if (keys & 4) opt_mic  = 1;
        if (keys & 8) opt_mic  = 0;
}

#if DEBUG_STATS
static void timer(int sig)
{
        static unsigned int cnt;
        unsigned int mini = 0;
        unsigned int maxi = 0;
        float min          = INT_MAX;
        float max          = INT_MIN;
        unsigned int i;

        for (i = 0; i < P; i++) {
                if (h_hat[i] < min) {
                        min  = h_hat[i];
                        mini = i;
                }
                if (h_hat[i] > max) {
                        max  = h_hat[i];
                        maxi = i;
                }
        }
        printf("f: %d\nm: %d\nad:%.4f\n", opt_filter,
            opt_mic, avg_damp);
        printf("%u: %f(%u) %f(%u)\n", cnt, min, mini, max
            , maxi);
        cnt++;
}
#endif

static void process_lms(int sig)
{
        sample_t *mics        = dsp_get_audio();        /*
            Left is x and right is d. */
```

```
sample_t *headphones = dsp_get_audio();        /*
    Two channels. */
sample_t *speaker    = dsp_get_audio_23();     /*
    Only the right one used. */
unsigned int i;

for (i = 0; i < DSP_BLOCK_SIZE; i++, bi =
    circindex(bi, 1, P)) {
        float x;
        float d;
        float y = 0.0;
        float e;
        ptrdiff_t ti;
        unsigned int k;
        float ftmp;
        int itmp;
        float sq = 0.0;

        /* Input */
        x = mics[i].left  / (float) INT_MAX;
        d = mics[i].right / (float) INT_MAX;

        if (!opt_mic)
                x = .5 * rand() / (float) INT_MAX
                    ;

        /* Compute */
        xbuff[bi] = x;
        for(k = 0, ti = bi; k < P; k++, ti =
            circindex(ti, -1, P))
                y += h_hat[k] * xbuff[ti];
        e = d - y;

        /* Adapt */
        for (k = 0; k < P; k++)
                sq += xbuff[k] * xbuff[k];
        for (k = 0, ti = bi; k < P; k++, ti =
            circindex(ti, -1, P))
                h_hat[k] += (mu * e * xbuff[ti])
                    / (sq +.000025);

        /* Output */
        speaker[i].right = (int) (x * INT_MAX);
        speaker[i].left  = 0.0;

        itmp = (int) (((opt_filter) ? e : d) *
            INT_MAX);
        headphones[i].left = itmp;
        headphones[i].right = itmp;
}
```

```
}


void main()
{
        dsp_init();

        interrupt(SIG_SP1, process_lms);
        interrupt(SIG_USR0, keyboard);
#if DEBUG_STATS
        interrupt(SIG_TMZ, timer);
        timer_set(98304000, 98304000);
        timer_on();
#endif
        dsp_start();

        for(;;) idle();
}
```

References

[1] https://en.wikipedia.org/w/index.php?title=Least_mean_squares_filter. [Online; last accessed 2014-03-01].

[2] Dimitris G. Manolakis John G. Proakis. Pearson Education, Inc., fourth edition, 2007.