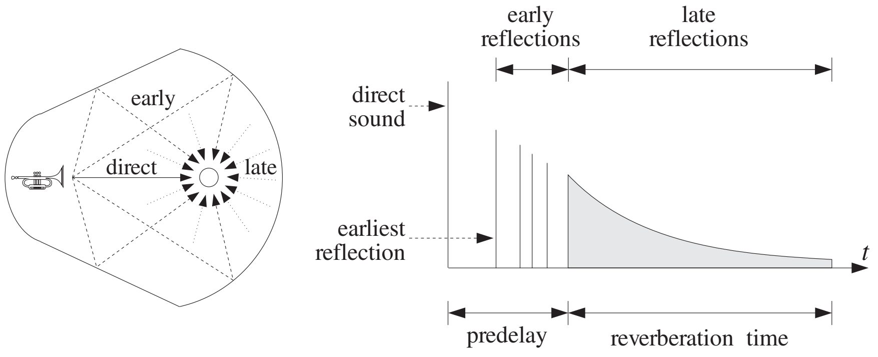
# Lab 6 – Digital Audio Effects

## 6.1. Plain Reverb

The reverberation of a listening space is typically characterized by three distinct time periods: the direct sound, the early reflections, and the late reflections, as illustrated below:



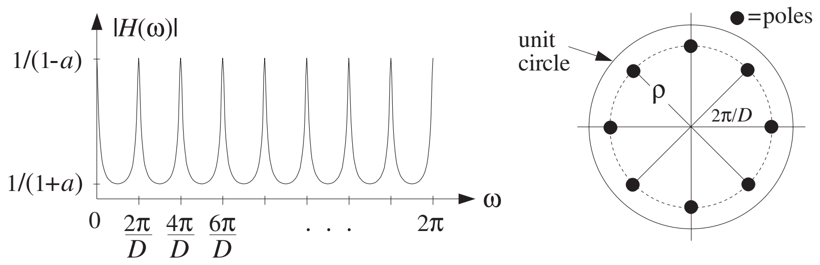
The early reflections correspond to the first few reflections off the walls of the room. As the waves continue to bounce off the walls, their density increases and they disperse, arriving at the listener from all directions. This is the late reflection part.

The reverberation time constant is the time it takes for the room's impulse response to decay by 60 dB. Typical concert halls have time constants of about 1.8–2 seconds.

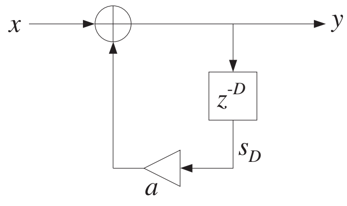
In this and several other labs, we discuss how to emulate such reverberation characteristics using DSP filtering algorithms. A *plain reverberator* can be used as an elementary building block for more complicated reverberation algorithms. It is given by Eq. (8.2.12) of the text [1] and shown in Fig. 8.2.6. Its input/output equation and transfer function are:

$$y(n) = ay(n-D) + x(n), \qquad H(z) = \frac{1}{1 - az^{-D}}$$

The comb-like structure of its frequency response and its pole-pattern on the $z$-plane are depicted in Fig. 8.2.7 of Ref. [1] and shown below.



Its sample processing algorithm using a circular delay-line buffer is given by Eq. (8.2.14) of [1]:



```
for each input sample x do:
    $s_D = *(p + D)$
    $y = x + a\, s_D$
    $*p = y$
    $--p$
```

It can be immediately translated to C code with the help of the function `pwrap()` and embedded in the interrupt service routine `isr()`:

```
interrupt void isr()
{
   float sD, x, y;               // D-th state, input & output

   read_inputs(&xL, &xR);        // read inputs from codec

   x = (float) xL;               // process left channel only

   sD = *pwrap(D,w,p+D);         // extract D-th state relative to p
   y = x + a*sD;                 // compute output sample
   *p = y;                       // delay-line input
   p = pwrap(D,w,--p);           // backshift pointer

   yL = yR = (short) y;

   write_outputs(yL,yR);         // write outputs to codec

   return;
}
```

**Lab Procedure**

a. Modify the template program into a C program. `plain1.c`, that implements the above ISR. Set the sampling rate to 8 kHz and the audio input to MIC. With the values of the parameters $D = 2500$ and $a = 0.5$, compile and run your program on the DSK.

   Listen to the impulse response of the system by lightly tapping the microphone on the table. Speak into the mike.

   Set the audio input to LINE, recompile and run. Play one of the wave files in the directory `c:\dsplab\wav` (e.g., `dsummer`, `noflange` from [3]).

b. Recompile and run the program with the new feedback coefficient $a = 0.25$. Listen to the impulse response. Repeat for $a = 0.75$. Discuss the effect of increasing or decreasing $a$.

c. According to Eq. (8.2.16), the effective reverberation time constant is given by

$$\tau_{\text{eff}} = \frac{\ln \epsilon}{\ln a} T_D, \qquad T_D = DT = D/f_s$$

   For each of the above values of $a$, calculate $\tau_{\text{eff}}$ in seconds, assuming $\epsilon = 0.001$ (which corresponds to the so-called 60-dB time constant.) Is what you hear consistent with this expression?

d. According to this formula, $\tau_{\text{eff}}$ remains invariant under the replacements:

$$D \to 2D, \qquad a \to a^2$$

   Test if this is true by running your program and hearing the output with $D = 5000$ and $a = 0.5^2 = 0.25$ and comparing it with the case $D = 2500$ and $a = 0.5$. Repeat the comparison also with $D = 1250$ and $a = \sqrt{0.5} = 0.7071$.

e. When the filter parameter $a$ is positive and near unity, the comb peak gains $1/(1-a)$ become large, and may cause overflows. In such cases, the input must be appropriately scaled down before it is passed to the filter.

   To hear such overflow effects, choose the feedback coefficients to be very near unity, for example, $a = 0.99$, with a corresponding gain of $(1-a)^{-1} = 100$. You may also need to multiply the input $x$ by an additional gain factor such as 2 or 4.

f. Modify the above ISR so that it processes the input samples in stereo (you will need to define two separate buffers for the left and right channels.) Experiment with choosing slightly different values of the left and right delay parameters $D$, or different values of the feedback parameter $a$. Keep the left/right speakers as far separated as possible.

## 6.2. Allpass Reverb

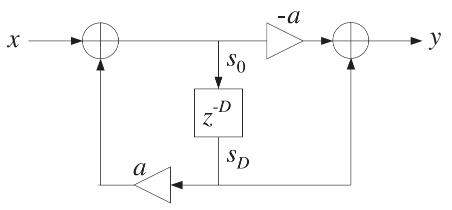
Like the plain reverberator, an allpass reverberator can be used as an elementary building block for building more complicated reverberation algorithms. It is given by Eq. (8.2.25) of the text [1] and shown in Fig. 8.2.17. Its I/O equation and transfer function are:

$$y(n)=ay(n-D)-ax(n)+x(n-D)\,,\qquad H(z)=\frac{-a+z^{-D}}{1-az^{-D}}$$

As discussed in [1], its impulse response is similar to that of the plain reverberator, but its magnitude response remains unity (hence the name “allpass”), that is,

$$|H(e^{j\omega})|=1\,,\quad \text{for all } \omega$$

Its block diagram representation using the so-called canonical realization and the corresponding sample processing algorithm using a circular delay-line buffer is given by Eq. (8.2.14) of [1]:



```
for each input sample x do:
    $s_D = *(p+D)$
    $s_0 = x + as_D$
    $y = -as_0 + s_D$
    $*p = s_0$
    $--p$
```

The algorithm can be translated immediately to C with the help of `pwrap()`. In this lab, we are going to put these steps into a separate C function, `allpass()`, which is to be called by `isr()`, and linked to the overall project. The function is defined as follows:

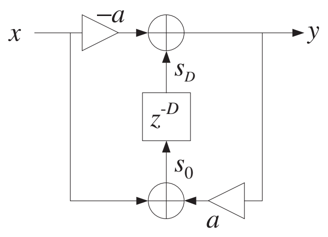
```
// ------------------------------------------------------------------------
// allpass.c - allpass reverb with circular delay line - canonical realization
// ------------------------------------------------------------------------

float *pwrap(int, float *, float *);

float allpass(int D, float *w, float **p, float a, float x)
{
   float y, s0, sD;

   sD = *pwrap(D,w,*p+D);

   s0 = x + a * sD;

   y  = -a * s0 + sD;

   **p = s0;

   *p = pwrap(D,w,--*p);

   return y;
}
// ------------------------------------------------------------------------
```

The `allpass` function is essentially the same as that in the text [1], but slightly modified to use floats and the function `pwrap()`. In the above definition, the parameter $p$ was declared as pointer to pointer to float because in the calling ISR function $p$ must be defined as a pointer to float and must be passed passed by address because it keeps changing from call to call. The calling ISR function `isr()` is defined as follows:

```
interrupt void isr()
{
   float x, y;

   read_inputs(&xL, &xR);          // read inputs from codec

   x = (float) xL;                 // process left channel only

   y = allpass(D,w,&p,a,x);        // to be linked with main()

   yL = yR = (short) y;

   write_outputs(yL,yR);           // write outputs to codec

   return;
}
```

Although the overall frequency response of the allpass reverberator is unity, the intermediate stage of computing the recursive part $s_0$ can overflow because this part is just like the plain reverb and its peak gain is $1/(1-a)$. Such overflow behavior is a potential problem of canonical realizations and we will investigate it further in a future lab.

The allpass reverberator can also be implemented in its transposed realization form, which is less prone to overflows. It is depicted below together with its sample processing algorithm:



```
for each input x do:
    $s_D = *(p+D)$
    $y = s_D - ax$
    $*p = s_0 = x + ay$
    $--p$
```

The following function `allpass_tr()` is the translation into C using `pwrap()`, where again $p$ is defined as a pointer to pointer to float:

```
// ------------------------------------------------------------------------------
// allpass_tr.c - allpass reverb with circular delay line - transposed realization
// ------------------------------------------------------------------------------

float *pwrap(int, float *, float *);                         // defined in dsplab.c

float allpass_tr(int D, float *w, float **p, float a, float x)
{
   float y, sD;

   sD = *pwrap(D,w,*p+D);

   y  = sD - a*x;

   **p = x + a*y;

   *p = pwrap(D,w,--*p);

   return y;
}
// ------------------------------------------------------------------------------
```

## Lab Procedure

a. Incorporate the above ISR into a main program, `allpass1.c`, and create a project. Remember to prototype the `allpass` function at the beginning of your program. Add the file that contains the `allpass` function to the project. Compile and run with the parameter choices: $D = 2500$, $a = 0.5$, with an 8 kHz sampling rate and LINE input.

b. Repeat part (a) using the transposed form implemented by the function `allpass_tr()`, and name your main program `allpass2.c`.

c. Choose a value of $a$ and input gain that causes `allpass1.c` to overflow, then run `allpass2.c` with the same parameter values to see if your are still getting overflows.

## 6.3. Lowpass Reverb

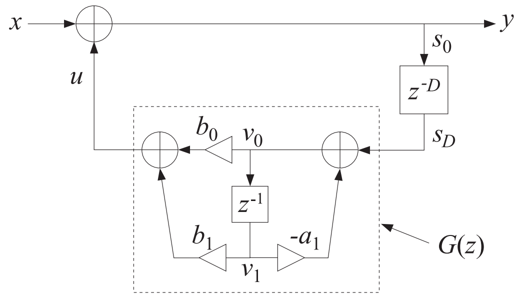
The lowpass reverberator of this experiment is shown in Figs. 8.2.20 and 8.2.21 of Ref. [1]. The feedback gain $a$ of the plain reverb is replaced by a lowpass filter $G(z)$, so that one obtains the new transfer function by the replacement:

$$H(z) = \frac{1}{1 - az^{-D}} \quad \Rightarrow \quad H(z) = \frac{1}{1 - G(z)z^{-D}}$$

The filter $G(z)$ effectively acts as frequency-dependent feedback parameter whose value is smaller at higher frequencies (because it is a lowpass filter), thus attenuating high frequencies faster, and whose value is larger at lower frequencies, and hence attenuating those more slowly—which is a more realistic behavior of reverberating spaces. For this experiment, we will work with the simple choice:

$$G(z) = \frac{b_0 + b_1 z^{-1}}{1 + a_1 z^{-1}}$$

Setting $a = -a_1$, the corresponding sample processing algorithm is:



```
for each input sample x do:
    $s_D = *(p + D)$
    $v_0 = av_1 + s_D$
    $u = b_0 v_0 + b_1 v_1$
    $y = x + u$
    $v_1 = v_0$
    $*p = y$
    $--p$
```

The following is its C translation into the `isr()` function:

```
interrupt void isr()
{
   float x, y, sD, u;

   read_inputs(&xL, &xR);      // read inputs from codec

   x = (float) xL;             // process left channel only

   sD = *pwrap(D,w,p+D);

   v0 = a*v1 + sD;             // feedback filter G(z) = (b0 + b1*z^-1)/(1-a*z^-1)
   u = b0*v0 + b1*v1;          // feedback filter's output
   v1 = v0;                    // update feedback filter's delay

   y = x+u;                    // closed-loop output

   *p = y;

   p = pwrap(D,w,--p);

   yL = yR = (short) y;

   write_outputs(yL,yR);       // write outputs to codec

   return;
}
```

**Lab Procedure**

a. Create a project with this ISR. Choose an 8 kHz sampling rate and MIC input. Set the parameter values $D = 2500$, $a = 0.5$, $b_0 = 0.2$, $b_1 = 0.1$. Compile and run. Listen to its impulse response. Speak into the mike. Notice how successive echoes get more and more mellow as they circulate through the lowpass filter. Note that the DC gain of the loop filter $G(z)$, obtained by setting $z = 1$, and the AC gain at Nyquist, obtained by setting $z = -1$, are:

$$G(z)\big|_{z=1} = \frac{b_0 + b_1}{1 - a} = 0.6\,, \qquad G(z)\big|_{z=-1} = \frac{b_0 - b_1}{1 + a} = \frac{1}{15} = 0.0667$$

These are the effective feedback coefficients at low and high frequencies. Therefore, the lower frequencies persist longer than the higher ones.

Recompile and run with LINE input and play a wave file (e.g., `noflange`) through it.

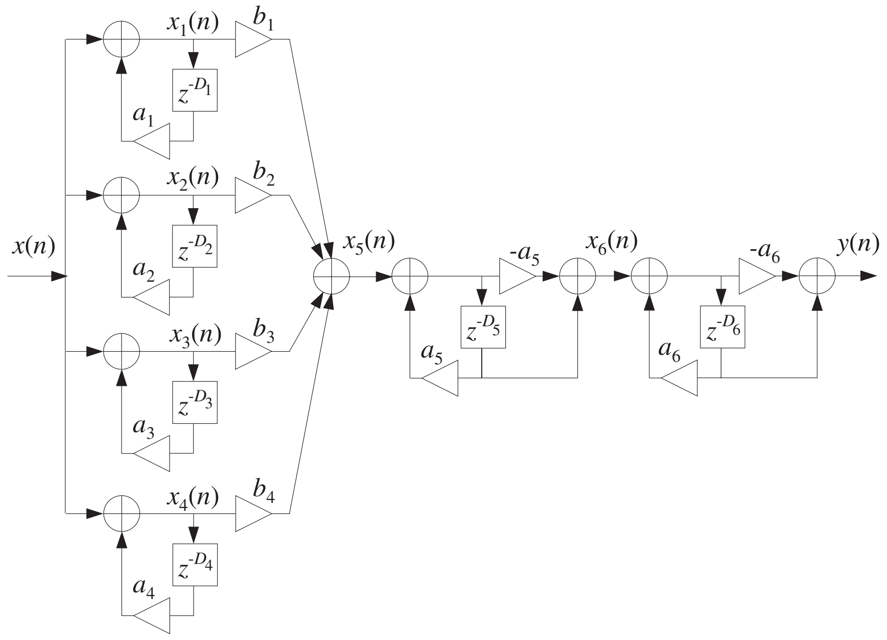
b. Try the case $D = 20$, $a = 0$, $b_0 = b_1 = 0.495$. You will hear a guitar-like sound. Repeat for $D = 100$. What do you hear?

Repeat by setting the sampling rate to 44.1 kHz and $D = 100$.

This type of feedback filter is the basis of the so-called Karplus-Strong string algorithm for synthesizing plucked-string sounds, and we will study it further in another experiment.

## 6.4. Schroeder's Reverb Algorithm

A more realistic reverberation effect can be achieved using Schroeder's model of reverberation, which consists of several plain reverb units in parallel, followed by several allpass units in series. An example is depicted in Fig. 8.2.18 and on the cover of the text [1], and shown below.



The different delays in the six units cause the density of the reverberating echoes to increase, generating an impulse response that exhibits the typical early and late reflection characteristics.

Its sample processing algorithm is given by Eq. (8.2.31) of [1]. It is stated in terms of the functions `plain()` and `allpass()` that implement the individual units:

```
for each input sample x do:
    $x_1 = \text{plain}(D_1, \mathbf{w}_1, \&p_1, a_1, x)$
    $x_2 = \text{plain}(D_2, \mathbf{w}_2, \&p_2, a_2, x)$
    $x_3 = \text{plain}(D_3, \mathbf{w}_3, \&p_3, a_3, x)$
    $x_4 = \text{plain}(D_4, \mathbf{w}_4, \&p_4, a_4, x)$
    $x_5 = b_1x_1 + b_2x_2 + b_3x_3 + b_4x_4$
    $x_6 = \text{allpass}(D_5, \mathbf{w}_5, \&p_5, a_5, x_5)$
    $y = \text{allpass}(D_6, \mathbf{w}_6, \&p_6, a_6, x_6)$
```
(6.1)

There are six multiple delays each requiring its own circular buffer and pointer. The `allpass()` function was already defined in the allpass reverb lab section. The `plain` function is straightforward and implements the steps used in the plain reverb lab section:

```
// -------------------------------------------------------
// plain.c - plain reverb with circular delay line
// -------------------------------------------------------

float *pwrap(int, float *, float *);

float plain(int D, float *w, float **p, float a, float x)
{
   float y, sD;

   sD = *pwrap(D,w,*p+D);

   y = x + a * sD;

   **p = y;

   *p = pwrap(D,w,--*p);

   return y;
}
// -------------------------------------------------------
```

The following (incomplete) C program implements the above sample processing algorithm in its `isr()` function and operates at a sampling rate of 44.1 kHz:

```
// ---------------------------------------------------------------------
// schroeder.c - Schroeder's reverb algorithm using circular buffers
// ---------------------------------------------------------------------

#include "dsplab.h"          // init parameters and function prototypes

short xL, xR, yL, yR;        // input and output samples from/to codec

short fs = 44;               // sampling rate in kHz

#define D1 1759
#define D2 1949
#define D3 2113
#define D4 2293
#define D5  307
#define D6  313

#define a 0.88

float b1=1, b2=0.9, b3=0.8, b4=0.7;
float a1=a, a2=a, a3=a, a4=a, a5=a, a6=a;

float w1[D1+1], *p1;
float w2[D2+1], *p2;
float w3[D3+1], *p3;
float w4[D4+1], *p4;
float w5[D5+1], *p5;
float w6[D6+1], *p6;

float plain(int, float *, float **, float, float);       // must be added to project
float allpass(int, float *, float **, float, float);

// ------------------------------------------------------------------------------------

void main()
{
   int n;
   for (n=0; n<=D1; n++) w1[n] = 0;             // initialize buffers to zero
   for (n=0; n<=D2; n++) w2[n] = 0;
   for (n=0; n<=D3; n++) w3[n] = 0;
   for (n=0; n<=D4; n++) w4[n] = 0;
   for (n=0; n<=D5; n++) w5[n] = 0;
   for (n=0; n<=D6; n++) w6[n] = 0;

   p1 = w1; p2 = w2; p3 = w3; p4 = w4; p5 = w5; p6 = w6;      // initialize pointers

  initialize();             // initialize DSK board and codec, define interrupts

  sampling_rate(fs);        // possible sampling rates: 8, 16, 24, 32, 44, 48, 96 kHz
  audio_source(MIC);        // LINE or MIC for line or microphone input

  while(1);                 // keep waiting for interrupt, then jump to isr()
}

// ------------------------------------------------------------------------------------

interrupt void isr()
{
   read_inputs(&xL, &xR);        // read inputs from codec

   // -----------------------------------------------------------
   // here insert your algorithm implementing Eq.(6.1) given above
   // -----------------------------------------------------------

   write_outputs(yL,yR);         // write outputs to codec

   return;
}

// ------------------------------------------------------------------------------------
```

**Lab Procedure**

a. Create a project for this program, compile and run it with audio input set to MIC. Listen to its impulse response and speak into the mike. To reduce potential overflow effects, you may want to reduce the input level by half, for example, by the statement:

```
x = (float) (xL>>1);
```

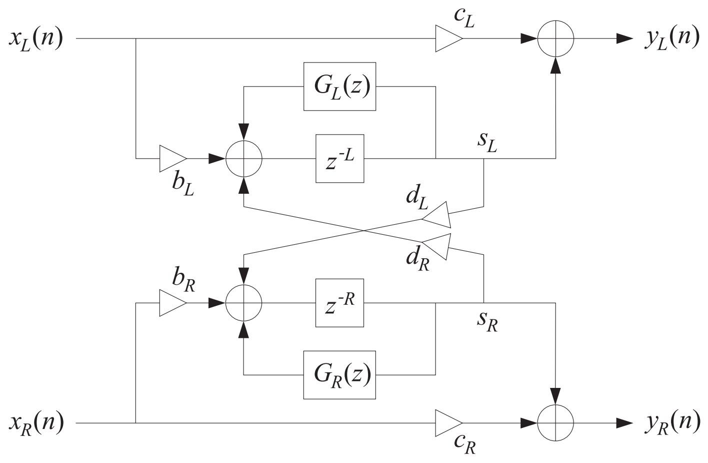
b. What are the feedback delays of each unit in msec? Replace all the delays by double their values, compile, and run again. Compare the output with that of part (a). Repeat when you triple all the delays. (Note that you can just replace the constant definitions by `#define D1 1759*2`, etc.)

c. Repeat part (a) by experimenting with different values of the feedback parameter $a$.

## 6.5. Stereo Reverb

In some of the previous experiments, we considered processing in stereo, but the left and right channels were processed completely independently of each other. In this experiment, we allow the cross-coupling of the two channels, so that the reverb characteristics of one channel influences those of the other.

An example of such system is given in Problems 8.22 and 8.23 and depicted in Fig. 8.4.1 of the text [1] and shown below.



Here, we assume that the feedback filters are plain multiplier gains, so that

$$G_L(z) = a_L, \qquad G_R(z) = a_R$$

Each channel has its own delay-line buffer and circular pointer. The sample processing algorithm is modified now to take in a pair of stereo inputs and produce a pair of stereo outputs:

```
for each input stereo pair $x_L$, $x_R$ do:
    $s_L = *(p_L + L)$
    $s_R = *(p_R + R)$
    $y_L = c_L x_L + s_L$
    $y_R = c_R x_R + s_R$
    $*p_L = s_{L0} = b_L x_L + a_L s_L + d_R s_R$
    $*p_R = s_{R0} = b_R x_R + a_R s_R + d_L s_L$
    $--p_L$
    $--p_R$
```

where $L$ and $R$ denote the left and right delays. Cross-coupling between the channels arises because of the coefficients $d_L$ and $d_R$. The following is its C translation into an `isr()` function:

```
interrupt void isr()          // sample processing algorithm - interrupt service routine
{
   float sL, sR;

   read_inputs(&xL, &xR);         // read inputs from codec

   sL = *pwrap(L,wL,pL+L);
   sR = *pwrap(R,wR,pR+R);
   yL = cL*xL + sL;
   yR = cR*xR + sR;
  *pL = bL*xL + aL*sL + dR*sR;
  *pR = bR*xR + aR*sR + dL*sL;
   pL = pwrap(L,wL,--pL);
   pR = pwrap(R,wR,--pR);

   write_outputs(yL,yR);          // write outputs to codec

   return;
}
```

**Lab Procedure**

a. Create a project whose main program includes the above ISR. Select an 8 kHz sampling rate and line input. Choose the following parameter values:

$$L = R = 3000, \quad a_L = a_R = 0, \quad b_L = b_R = 0.8, \quad c_L = c_R = 0.5, \quad d_L = d_R = 0.5$$

Compile and run this program. Even though the self-feedback multipliers were set to zero, $a_L = a_R = 0$, you will hear repeated echoes bouncing back and forth between the speakers because of the cross-coupling. Make sure the speakers are as far separated as possible, and play one of the wave files in `c:\dsplab\wav` (e.g., `take5`, `dsummer`).

b. Next try the case $d_L \neq 0$, $d_R = 0$. And then, $d_L = 0$, $d_R \neq 0$. These choices decouple the influence of one channel but not that of the other.

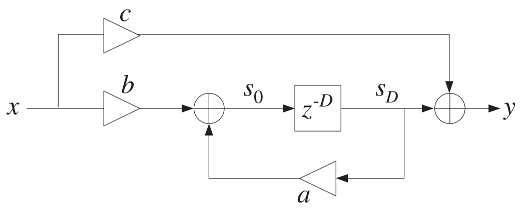
c. Next, introduce some self-feedback, such as $a_L = a_R = 0.2$. Repeat part (a). Vary all the parameters at will to see what you get.

## 6.6. Reverberating Delay

A prototypical delay effect found in most commercial audio effects processors was discussed in Problem 8.17 of the text [1]. Its transfer function is:

$$H(z) = c + b\frac{z^{-D}}{1 - az^{-D}}$$

Its block diagram realization and corresponding sample processing algorithm using a circular delay-line buffer are given below:



```
for each input x do:
    $s_D = *(p + D)$
    $y = cx + s_D$
    $*p = s_0 = bx + as_D$
    $--p$
```

The following is its C translation into an `isr()` function:

```
interrupt void isr()          // sample processing algorithm - interrupt service routine
{
   float sD, x, y;                // D-th state, input & output

   read_inputs(&xL, &xR);         // read inputs from codec

   x = (float) xL;                // process left channel only

   sD = *pwrap(D,w,p+D);          // extract states relative to p
   y = c*x + sD;                  // output sample
   *p = b*x + a*sD;               // delay-line input
   p = pwrap(D,w,--p);            // backshift pointer

   yL = yR = (short) y;

   write_outputs(yL,yR);          // write outputs to codec

   return;
}
```

**Lab Procedure**

a. Create a project, compile and run it with 8 kHz sampling rate and MIC input. Choose the parameters:

$$D = 6000, \quad a = 0.5, \quad b = 1, \quad c = 0$$

Listen to its impulse response and speak into the mike. Here, the direct sound path has been removed, $c = 0$, in order to let the echoes be more clearly heard.

b. What values of $b$ and $c$ would you use (expressed in terms of $a$) in order to implement a plain reverberator of the form:

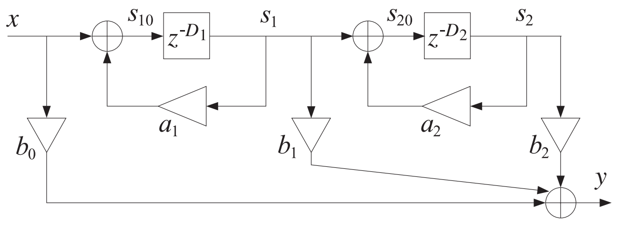
$$H(z) = \frac{1}{1 - az^{-D}}$$

For $a = 0.5$, calculate the proper values of $b, c$, and then compile and run the program. Compare its output with that of `plain1.c`.

c. Compile and run the case: $a = 1$, $b = c = 1$, and then the case: $a = -1$, $b = -1$, $c = 1$. What are the transfer functions in these cases?

## 6.7. Multi-Delay Effects

Here, we consider the multi-delay effects processor shown in Fig. 8.2.27 of the text [1]. We assume that the feedback filters are plain multipliers. Using two separate circular buffers for the two delays, the block diagram realization and sample processing algorithm are in this case:



```
for each input x do:
    $s_1 = *(p_1 + D_1)$
    $s_2 = *(p_2 + D_2)$
    $y = b_0 x + b_1 s_1 + b_2 s_2$
    $*p_2 = s_{20} = s_1 + a_2 s_2$
    $--p_2$
    $*p_1 = s_{10} = x + a_1 s_1$
    $--p_1$
```

Its C translation is straightforward:

```
interrupt void isr()          // sample processing algorithm - interrupt service routine
{
   float x, s1, s2, y;

   read_inputs(&xL, &xR);         // read inputs from codec

   x = (float) xL;                // process left channel only

   s1 = *pwrap(D1, w1, p1+D1);
   s2 = *pwrap(D2, w2, p2+D2);

   y = b0*x + b1*s1 + b2*s2;

  *p2 = s1 + a2*s2;
   p2 = pwrap(D2, w2, --p2);

  *p1 = x + a1*s1;
   p1 = pwrap(D1, w1, --p1);

   yL = yR = (short) y;

   write_outputs(yL,yR);          // write outputs to codec

   return;
}
```

**Lab Procedure**

a. Write a main program, `multidel.c`, that incorporates this ISR, compile and run it with an 8 kHz sampling rate and MIC input, and the following parameter choices:

$$D_1 = 5000, \quad D_2 = 2000, \quad a_1 = 0.5, \quad a_2 = 0.4, \quad b_0 = 1, \quad b_1 = 0.8, \quad b_2 = 0.6$$

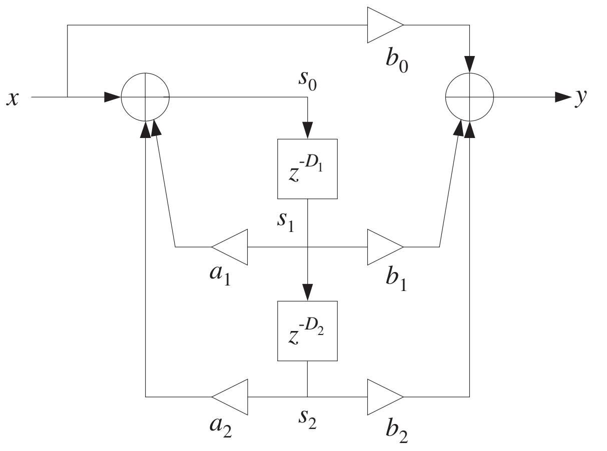
Listen to its impulse response and speak into the mike. Then select LINE input and play a wave file (e.g., `dsummer`) through it.

b. Set $b_1 = 0$ and run again. Then, set $b_2 = 0$ and run. Can you explain what you hear?

## 6.8. Multitap Delay Effects

This experiment is based on the multi-tap delay line effects processor of Fig. 8.2.29 of the text [1]. Both this effect and the multi-delay effect of the previous section are commonly found in commercially available digital audio effects units.

The implementation uses a common circular delay-line buffer of order $D_1+D_2$, which is tapped out at taps $D_1$ and $D_1+D_2$. The sample processing algorithm is:



```
for each input sample x do:
    $s_1 = *(p + D_1)$
    $s_2 = *(p + D_1 + D_2)$
    $y = b_0 x + b_1 s_1 + b_2 s_2$
    $s_0 = x + a_1 s_1 + a_2 s_2$
    $*p = s_0$
    $--p$
```

The following ISR is its C translation:

```
interrupt void isr()          // sample processing algorithm - interrupt service routine
{
   float x, s0, s1, s2, y;

   read_inputs(&xL, &xR);         // read inputs from codec

   x = (float) xL;                // process left channel only

   s1 = *pwrap(D1+D2, w, p+D1);
   s2 = *pwrap(D1+D2, w, p+D1+D2);
   y = b0*x + b1*s1 + b2*s2;
   s0 = x + a1*s1 + a2*s2;
   *p = s0;
   p = pwrap(D1+D2, w, --p);

   yL = yR = (short) y;

   write_outputs(yL,yR);          // write outputs to codec

   return;
}
```

**Lab Procedure**

a. Write a main program, `multidel.c`, that incorporates this ISR, compile and run it with an 8 kHz sampling rate and MIC input, and the following parameter choices:

$$D_1 = 3000, \quad D_2 = 1500, \quad a_1 = 0.2, \quad a_2 = 0.5, \quad b_0 = 1, \quad b_1 = 0.8, \quad b_2 = 0.6$$

Listen to its impulse response and speak into the mike. Then select LINE input and play a wave file (e.g., `dsummer`) through it.

b. Repeat for the following values of the feedback parameters: $a_1 = a_2 = 0.5$, which makes the system marginally stable with a periodic steady output (any random noise would be grow unstable.)

Repeat also for the case $a_1 = a_2 = 0.75$, which corresponds to an unstable filter. Please reset the processor before the output grows too loud. However, do let it grow loud enough to hear the overflow effects arising from the growing feedback output $s_0$.

As discussed in Ref. [1], the condition of stability for this filter is $|a_1| + |a_2| < 1$. Interestingly, most commercially available digital audio effects units allow the setting of the parameters $D_1, D_2, a_1, a_2, b_0, b_1, b_2$ from their front panel, but do not check this stability condition.

## 6.9. Karplus-Strong String Algorithm

A model of a plucked string is obtained by running the lowpass reverb filter with zero input, but with initially filling the delay line with random numbers. These random numbers model the initial harshness of plucking the string. But, as the random numbers recirculate through the lowpass filter, their high frequencies are gradually removed, resulting in a sound that models the string vibration.

The model can be approximately "tuned" to a frequency $f_1$ by picking $D$ such that $D = f_s/f_1$. (There are ways to "fine-tune", but we do not consider them in this simple experiment.) The Karplus-Strong model [9] assumes a simple averaging FIR filter for the lowpass feedback filter as given by Eq. (8.2.40) of the text [1]. Here, we take the transfer function to be:

$$G(z) = b_0(1 + z^{-1})$$

with some $b_0 \lesssim 0.5$ to improve the stability of the closed-loop system. See Refs. [4–15] for more discussion on such models and computer music in general. The following program implements the algorithm. The code is identical to that of the lowpass reverb case.

The sampling rate is set to 44.1 kHz and the generated sound is the note A440, that is, having frequency 440 Hz. The correct amount of delay is then

$$D = \frac{f_s}{f_1} = \frac{44100}{440} \approx 100$$

The delay line must be filled with $D+1$ random numbers. They were generated as follows by MATLAB and exported to the file `rand.dat` using the function `C_header()`, e.g., by the code:

```
iseed = 1000; randn('state', iseed);
r = 10000 * randn(101,1);
C_header('rand.dat', 'r', 'D', r);
```

The full program is as follows:

```
// ----------------------------------------------------------------------------------
// ks.c - Karplus-Strong string algorithm
// ----------------------------------------------------------------------------------

#include "dsplab.h"            // init parameters and function prototypes

short xL, xR, yL, yR;          // input and output samples from/to codec
```

```
#define D 100

float w[D+1], *p;             // circular delay-line buffer, circular pointer

#include "rand.dat"           // D+1 random numbers

float a = 0;
float b0 = 0.499, b1 = 0.499;

float v0, v1;                 // states of feedback filter

short fs = 44;                // sampling rate is 44.1 kHz

// ------------------------------------------------------------------------------------

void main()                  // main program executed first
{
   int n;
   for (n=0; n<=D; n++)      // initialize circular buffer to zero
      w[n] = r[n];
   p = w;                    // initialize pointer
   v1 = 0;                   // initialize feedback filter

  initialize();              // initialize DSK board and codec, define interrupts

  sampling_rate(fs);         // possible sampling rates: 8, 16, 24, 32, 44, 48, 96 kHz
  audio_source(LINE);        // LINE or MIC for line or microphone input

  while(1);                  // keep waiting for interrupt, then jump to isr()
}

// ------------------------------------------------------------------------------------

interrupt void isr()          // sample processing algorithm - interrupt service routine
{
   float y, sD, u;

   // read_inputs(&xL, &xR);         // inputs not used

   sD = *pwrap(D,w,p+D);

   v0 = a*v1 + sD;             // feedback filter G(z) = (b0 + b1*z^-1)/(1-a*z^-1)
   u = b0*v0 + b1*v1;          // feedback filter's output
   v1 = v0;                    // update feedback filter's delay

   y = u;                      // closed-loop output - with x=0

   *p = y;

   p = pwrap(D,w,--p);

   yL = yR = (short) y;

   write_outputs(yL,yR);          // write outputs to codec

   return;
}
// ------------------------------------------------------------------------------------
```

**Lab Procedure**

a. Create a project, compile and run. The program disables the inputs and simply outputs the recirculating and gradually decaying random numbers.

b. Repeat for $D = 200$ by generating a new file `rand.dat` using the above MATLAB code. The note you hear should be an octave lower.

## *6.10. Wavetable Generators*

Wavetable generators are discussed in detail in Sect. 8.1.3 of the text [1]. A wavetable is defined by a circular buffer **w** whose dimension $D$ is chosen such that the smallest frequency to be generated is:

$$f_{\min} = \frac{f_s}{D} \quad \Rightarrow \quad D = \frac{f_s}{f_{\min}}$$

For example, if $f_s = 8$ kHz and the smallest desired frequency is $f_{\min} = 10$ Hz, then one must choose $D = 8000/10 = 800$. The $D$-dimensional buffer holds one period at the frequency $f_{\min}$ of the desired waveform to be generated. The shape of the stored waveform is arbitrary, and can be a sinusoid, a square wave, sawtooth, etc. For example, if it is sinusoidal, then the buffer contents will be:

$$w[n] = \sin\left(\frac{2\pi f_{\min}}{f_s}\, n\right) = \sin\left(\frac{2\pi n}{D}\right), \quad n = 0, 1, \dots, D-1$$

Similarly, a square wave whose first half is $+1$ and its second half, $-1$, will be defined as:

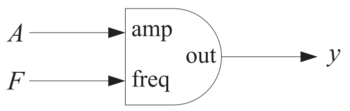
$$w[n] = \begin{cases} +1, & \text{if } \; 0 \le n < D/2 \\ -1, & \text{if } \; D/2 \le n < D \end{cases}$$

To generate higher frequencies (with the Nyquist frequency $f_s/2$ being the highest), the wavetable is cycled in steps of $c$ samples, where $c$ is related to the desired frequency by:

$$f = c f_{\min} = c\,\frac{f_s}{D} \quad \Rightarrow \quad c = D\,\frac{f}{f_s} \equiv DF, \quad F = \frac{f}{f_s}$$

where $F = f/f_s$ is the frequency in units of [cycles/sample]. The generated signal of frequency $f$ and amplitude $A$ is obtained by the loop:

```
repeat forever:
    $y = A\, w[q]$
    $q = (q + c) \bmod (D)$
```



The shift $c$ need not be an integer. In such case, the quantity $q + c$ must be truncated to the integer just below it. The text [1] discusses alternative methods, for example, rounding to the nearest integer, or, linearly interpolating. For the purposes of this lab, the truncation method will suffice.

The following function, `wavgen()`, based on Ref. [1], implements this algorithm. The mod-operation is carried out with the help of the function `qwrap()`:

```
// ------------------------------------------------
// wavgen.c - wavetable generator
// Usage: y = wavgen(D,w,A,F,&q);
// -----------------------------------------------

int qwrap(int, int);

float wavgen(int D, float *w, float A, float F, int *q)
{
   float y, c=D*F;

   y = A * w[*q];

   *q = qwrap(D-1, (int) (*q+c));

   return y;
}

// ------------------------------------------------
```

We note that the circular index $q$ is declared as a pointer to int, and therefore, must be passed by address in the calling program. Before using the function, the buffer **w** must be loaded with one period of length $D$ of the desired waveform. This function differs from the one in Ref. [1] in that it loads the buffer in forward order and cycles the index $q$ forward.

Here, we present some examples of wavetable generators using the function `wavgen()`. Two wavetables can be used in combination to illustrate AM and FM modulation.

## Sinusoidal Wavetable

The following program generates a 1 kHz sinusoid from a wavetable of length $D = 4000$. At a sampling rate of 8 kHz, the smallest frequency that can be generated is $f_{\min} = f_s/D = 8000/4000 = 2$ Hz.

```
// sinex.c - sine wavetable example
// -------------------------------------------------------------------------------------

#include "dsplab.h"          // DSK initialization declarations and function prototypes
#include <math.h>
#define PI 3.14159265

short xL, xR, yL, yR;        // left and right input and output samples from/to codec

#define D 4000               // fmin = fs/D = 8000/4000 = 2 Hz
float w[D];                  // wavetable buffer

short fs=8;                  // fs = 8 kHz
float A=10000, f=1;          // f = 1 kHz
int q;

float wavgen(int, float *, float, float, int *);

// -------------------------------------------------------------------------------------

void main()
{
  int i;

  q=0;                                           // initialize circular index

  for (i=0; i<D; i++) w[i] = sin(2*PI*i/D);      // load wavetable in forward order

  initialize();
  sampling_rate(fs);
  audio_source(LINE);

  while(1);
}

// -------------------------------------------------------------------------------------

interrupt void isr()
{
  float y;                 // filter input & output

   //read_inputs(&xL, &xR);           // codec inputs are not used

   y = wavgen(D, w, A, f/fs, &q);

   yL = yR = (short) y;

   write_outputs(yL,yL);

  return;
}

// -------------------------------------------------------------------------------------
```

The wavetable is loaded with the sinusoid in `main()`. At each sampling instant, the program does nothing with the codec inputs, rather, it generates a sample of the sinusoid by a call to `wavgen` and sends it to the codec.

**Lab Procedure**

a. Create a project for this program and run it. The amplitude was chosen to be $A = 10000$ in order to make the wavetable output audible. Reset the frequency to 200 Hz, recompile and run.

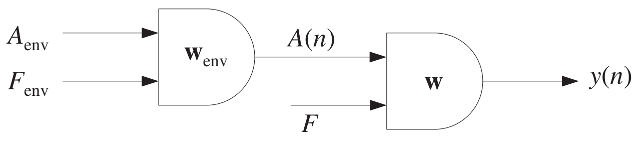
b. Create a GEL file with a slider for the value of the frequency over the interval $0 \le f \le 1$ kHz in steps of 100 Hz. Open the slider and run the program while changing the frequency with the slider.

c. Set the frequency to 30 Hz and run the program. Keep decreasing the frequency by 5 Hz at a time and determine the lowest frequency that you can hear (but, to be fair don't increase the speaker volume; that would compensate the attenuation introduced by your ears.)

d. Replace the sinusoidal table with the square wavetable, which has period 4000 and is equal to $+1$ for the first half of the period and $-1$ for the second half (see the FM example on how to do that). Run the program with frequency $f = 1$ kHz and $f = 200$ Hz.

**AM Modulation**

Here, we use two wavetables to illustrate AM modulation. The picture below shows how one wavetable is used to generate a modulating amplitude signal, which is fed into the amplitude input of a second wavetable.



The AM-modulated signal is of the form:

$$x(t) = A(t)\sin(2\pi f t), \qquad \text{where} \quad A(t) = A_{\text{env}}\sin(2\pi f_{\text{env}} t)$$

The following program, `amex.c`, shows how to implement this with the function `wavgen()`. The envelope frequency is chosen to be 2 Hz and the signal frequency 200 Hz. A common sinusoidal wavetable sinusoidal buffer is used to generate both the signal and its sinusoidal envelope.

```
// amex.c - AM example
// ------------------------------------------------------------------------------------

#include "dsplab.h"          // DSK initialization declarations and function prototypes
#include <math.h>
#define PI 3.14159265

short xL, xR, yL, yR;        // left and right input and output samples from/to codec

#define D 8000               // fmin = fs/D = 8000/8000 = 1 Hz
float w[D];                  // wavetable buffer

short fs=8;
float A, f=0.2;
float Ae=10000, fe=0.002;
int q, qe;

float wavgen(int, float *, float, float, int *);

// ------------------------------------------------------------------------------------

void main()
{
  int i;

  q=qe=0;

  for (i=0; i<D; i++) w[i] = sin(2*PI*i/D);          // fill sinusoidal wavetable

  initialize();
  sampling_rate(fs);
  audio_source(LINE);

  while(1);
}

// -------------------------------------------------------------------------------------

interrupt void isr()
{
  float y;

   // read_inputs(&xL, &xR);              // inputs not used

   A = wavgen(D, w, Ae, fe/fs, &qe);
   y = wavgen(D, w, A, f/fs, &q);

   yL = yR = (short) y;

   write_outputs(yL,yL);

  return;
}

// ------------------------------------------------------------------------------------
```

Although the buffer is the same for the two wavetables, two different circular indices, $q, q_e$ are used for the generation of the envelope amplitude signal and the carrier signal.

**Lab Procedure**

a. Run and listen to this program with the initial signal frequency of $f = 200$ Hz and envelope frequency of $f_{\text{env}} = 2$ Hz. Repeat for $f = 2000$ Hz. Repeat the previous two cases with $f_{\text{env}} = 20$ Hz.

b. Repeat and explain what you hear for the cases:

$$\begin{aligned} f &= 200\text{ Hz}, \quad f_{\text{env}} = 100\text{ Hz} \\ f &= 200\text{ Hz}, \quad f_{\text{env}} = 190\text{ Hz} \\ f &= 200\text{ Hz}, \quad f_{\text{env}} = 200\text{ Hz} \end{aligned}$$

**FM Modulation**

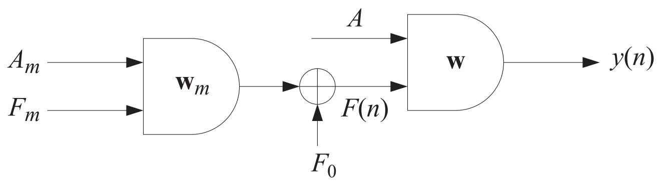
The third program, `fmex.c`, illustrates FM modulation in which the frequency of a sinusoid is time-varying. The generated signal is of the form:

$$x(t) = \sin\big[2\pi f(t)t\big]$$

The frequency $f(t)$ is itself varying sinusoidally with frequency $f_m$:

$$f(t) = f_0 + A_m \sin(2\pi f_m t)$$

Its variation is over the interval $f_0 - A_m \le f(t) \le f_0 + A_m$. In this experiment, we choose the modulation depth $A_m = 0.3f_0$, so that $0.7f_0 \le f(t) \le 1.3f_0$. The center frequency is chosen as $f_0 = 500$ Hz and the modulation frequency as $f_m = 1$ Hz. Again two wavetables are used as shown below, with the first one generating $f(t)$, which then drives the frequency input of the second generator.



```
// fmex.c - FM example
// ------------------------------------------------------------------------------------

#include "dsplab.h"          // DSK initialization declarations and function prototypes
#include <math.h>
#define PI 3.14159265

short xL, xR, yL, yR;        // left and right input and output samples from/to codec

#define D 8000               // fmin = fs/D = 8000/8000 = 1 Hz
float w[D];                  // wavetable buffer

short fs=8;
float A=5000, f=0.5;
float Am=0.3, fm=0.001;
int q, qm;

float wavgen(int, float *, float, float, int *);

// ------------------------------------------------------------------------------------

void main()
{
  int i;

  q = qm = 0;

  for (i=0; i<D; i++) w[i] = sin(2*PI*i/D);            // load sinusoidal wavetable
  //for (i=0; i<D; i++) w[i] = (i<D/2)? 1 : -1;        // square wavetable

  initialize();
  sampling_rate(fs);
  audio_source(LINE);

  while(1);
}

// -------------------------------------------------------------------------------------

interrupt void isr()
{
  float y, F;

   // read_inputs(&xL, &xR);                            // inputs not used

   F = (1 + wavgen(D, w, Am, fm/fs, &qm)) * f/fs;       // modulated frequency

   y = wavgen(D, w, A, F, &q);                          // FM signal

   yL = yR = (short) y;

   write_outputs(yL,yL);

  return;
}

// ------------------------------------------------------------------------------------
```

**Lab Procedure**

a. Compile, run, and hear the program with the following three choices of the modulation depth: $A_m = 0.3f_0$, $A_m = 0.8f_0$, $A_m = f_0$, $A_m = 0.1f_0$. Repeat these cases when the center frequency is changed to $f_0 = 1000$ Hz.

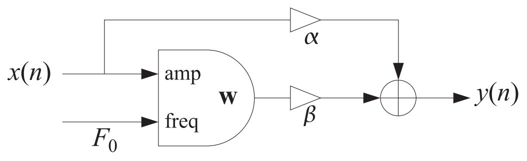
b. Replace the sinusoidal wavetable with a square one and repeat the case $f_0 = 500$ Hz, $A_m = 0.3f_0$. You will hear a square wave whose frequency switches between a high and a low value in each second.

c. Keep the square wavetable that generates the alternating frequency, but generate the signal by a sinusoidal wavetable. To do this, generate a second sinusoidal wavetable and define a circular buffer for it in `main()`. Then generate your FM-modulated sinusoid using this table. The generated signal will be of the form:

$$x(t) = \sin\big[2\pi f(t)t\big], \qquad f(t) = 1 \text{ Hz square wave}$$

## 6.11. Ring Modulators and Tremolo

Interesting audio effects can be obtained by feeding the audio input to the amplitude of a wavetable generator and combining the resulting output with the input, as shown below:



For example, for a sinusoidal generator of frequency $F_0 = f_0/f_s$, we have:

$$y(n) = \alpha x(n) + \beta x(n)\cos(2\pi F_0 n) = x(n)\big[\alpha + \beta\cos(2\pi F_0 n)\big] \tag{6.2}$$

The *ring modulator* effect is obtained by setting $\alpha = 0$ and $\beta = 1$, so that

$$y(n) = x(n)\cos(2\pi F_0 n) \tag{6.3}$$

whereas, the *tremolo* effect corresponds to $\alpha = 1$ and $\beta \neq 0$

$$y(n) = x(n) + \beta x(n)\cos(2\pi F_0 n) = x(n)\big[1 + \beta\cos(2\pi F_0 n)\big] \tag{6.4}$$

The following ISR function implements either effect:

```
// ------------------------------------------------------------------------------------

interrupt void isr()
{
  float x, y;

   read_inputs(&xL, &xR);

   x = (float) xL;

   y = alpha * x + beta * wavgen(D, w, x, f/fs, &q);

   yL = yR = (short) y;

   write_outputs(yL,yL);

  return;
}

// -----------------------------------------------------------------------------------
```

**Lab Procedure**

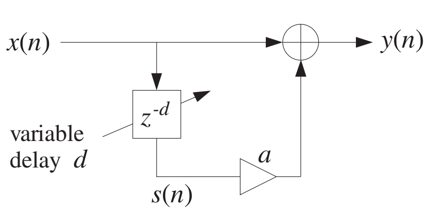
a. Modify the `amex.c` project to implement the ring modulator/tremolo effect. Set the carrier frequency to $f_0 = 400$ Hz and $\alpha = \beta = 1$. Compile, run, and play a wavefile with voice in it (e.g., `dsummer`.)

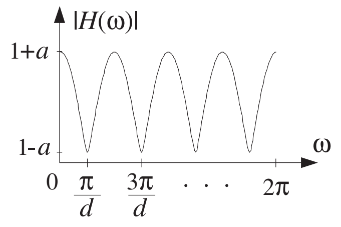
b. Experiment with higher and lower values of $f_0$.

c. Repeat part (a) when $\alpha = 0$ and $\beta = 1$ to hear the ring-modulator effect.

## 6.12. Flangers and Vibrato

As discussed in Ref. [1], a flanging effect is implemented as an FIR comb filter with a time-variable delay.





If the delay $d$ varies sinusoidally between $0 \le d(n) \le D$, with some low frequency $f_d$, then

$$d(n) = \frac{D}{2}\big[1 - \cos(\omega_d n)\big], \quad \omega_d = \frac{2\pi f_d}{f_s} \quad \text{[rads/sample]}$$

and the flanger output is obtained by

$$y(n) = x(n) + ax\big(n - d(n)\big)$$

If the delay $d$ were fixed, the transfer function would be:

$$H(z) = 1 + az^{-d}$$

The peaks of the frequency response of the resulting time-varying comb filter, occurring at multiples of $f_s/d$, and its notches at odd multiples of $f_s/2d$, will sweep up and down the frequency axis resulting in the characteristic whooshing type sound called flanging. The parameter $a$ controls the depth of the notches. In units of [radians/sample], the notches occur at odd multiples of $\pi/d$.

In the early days, the flanging effect was created by playing the music piece simultaneously through two tape players and alternately slowing down each tape by manually pressing the flange of the tape reel.

Because the variable delay $d$ can take non-integer values within its range $0 \le d \le D$, the implementation requires the calculation of the output $x(n-d)$ of a delay line at such non-integer values. This can be accomplished easily by truncating to the nearest integer, or as discussed in [1], by rounding, or by linear interpolation. To sharpen the comb peaks one may use a plain-reverb filter with variable delay, that is,

$$y(n) = x(n) + ay(n-d), \qquad H(z) = \frac{1}{1 - az^{-d}}$$

Its sample processing algorithm using a circular buffer of maximum order $D$ is:

```
for each input $x$ do:
    $d = \text{floor}\big[(1-\cos(\omega_d n))D/2\big]$
    $s_d = *(p+d)$
    $y = x + as_d$
    $*p = y$
    $--p$
```

Its translation to C is straightforward and can be incorporated into the ISR function:

```
interrupt void isr()          // sample processing algorithm - interrupt service routine
{
   float sd;

   read_inputs(&xL, &xR);           // read inputs from codec

   x = (float) xL;                  // work with left input only

   d = (1 - cos(wd*n))*D/2;         // automatically cast to int, wd = 2*PI*fd/fs
   if (++n>=L) n=0;                 // L = 16000 to allow fd = 0.5 Hz

   sd = *pwrap(D,w,p+d);            // extract d-th state relative to p
   y = x + a*sd;                    // output
   *p = y;                          // delay-line input
   p = pwrap(D,w,--p);              // backshift pointer

   yL = yR = (short) y;

   write_outputs(yL,yR);            // write outputs to codec

   return;
}
```

**Lab Procedure**

a. Create a project for this ISR. You will need to include `<math.h>` and define `PI`. Choose $D$ to correspond to a 2 msec maximum delay and let $f_d = 1$ Hz and $a = 0.7$. Run the program and play a wave file through it (e.g., `noflange`, `dsummer`, `take5`). Repeat when $f_d = 0.5$ Hz.

b. Experiment with other values of $D$, $f_d$, and $a$.

c. Rewrite part (a) so that an FIR comb filter is used as shown at the beginning of this section. Play the same material through the IIR and FIR versions and discuss differences in their output sounds.

d. A *vibrato* effect can be obtained by using the filter $H(z) = z^{-d}$ with a variable delay. You can easily modify your FIR comb filter of part (c) so that the output is taken directly from the output of the delay. For this effect the typical delay variations are about 5 msec and their frequency about 5 Hz. Create a vibrato project with $D = 16$ (correspondoing to 2 msec at an 8 kHz rate) and $f_d = 5$ Hz, and play a wave file through it. Repeat by doubling $D$ and/or $f_d$.

## 6.13. References

[1] S. J. Orfanidis, *Introduction to Signal Processing*, online book, 2010, available from: `http://www.ece.rutgers.edu/~orfanidi/intro2sp/`

[2] R. Chassaing and D. Reay, *Digital Signal Processing and Applications with the TMS320C6713 and TMS320C6416 DSK*, 2nd ed., Wiley, Hoboken, NJ, 2008.

[3] M. J. Caputi, "Developing Real-Time Digital Audio Effects for Electric Guitar in an Introductory Digital Signal Processing Class," *IEEE Trans. Education*, **41**, no.4, (1998), available online from: `http://www.ewh.ieee.org/soc/es/Nov1998/01/BEGIN.HTM`

[4] F. R. Moore, *Elements of Computer Music*, Prentice Hall, Englewood Cliffs, NJ, 1990.

[5] C. Roads and J. Strawn, eds., *Foundations of Computer Music*, MIT Press, Cambridge, MA, 1988.

[6] C. Roads, ed., *The Music Machine*, MIT Press, Cambridge, MA, 1989.

[7] C. Dodge and T. A. Jerse, *Computer Music*, Schirmer/Macmillan, New York, 1985.

[8] J. M. Chowning, "The Synthesis of Complex Audio Spectra by Means of Frequency Modulation," *J. Audio Eng. Soc.*, **21**, 526 (1973). Reprinted in Ref. [5].

[9] R. Karplus and A. Strong, “Digital Synthesis of Plucked String and Drum Timbres,” *Computer Music J.*, **7**, 43 (1983). Reprinted in Ref. [6].

[10] D. A. Jaffe and J. O. Smith, “Extensions of the Karplus-Strong Plucked-String Algorithm,” *Computer Music J.*, **7**, 56 (1983). Reprinted in Ref. [6].

[11] C. R. Sullivan, “Extending the Karplus-Strong Algorithm to Synthesize Electric Guitar Timbres with Distortion and Feedback,” *Computer Music J.*, **14**, 26 (1990).

[12] J. O. Smith, “Physical Modeling Using Digital Waveguides,” *Computer Music J.*, **16**, 74 (1992).

[13] J. A. Moorer, “Signal Processing Aspects of Computer Music: A Survey,” *Proc. IEEE*, **65**, 1108 (1977). Reprinted in Ref. [5].

[13] M. Kahrs and K. Brandenburg, eds., *Applications of Digital Signal Processing to Audio and Acoustics*, Kluwer, Boston, 1998.

[15] Udo Zölzer, ed., *DAFX – Digital Audio Effects*, Wiley, Chichester, England, 2003. See also the DAFX Conference web page: `http://www.dafx.de/`.