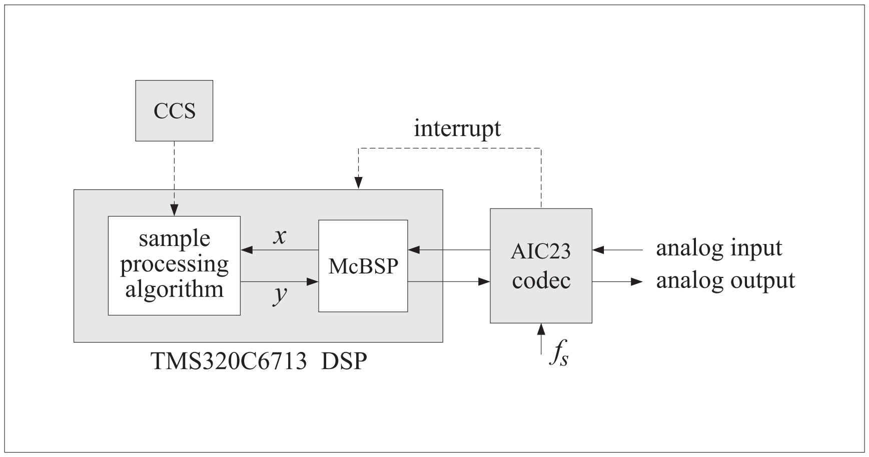
RUTGERS UNIVERSITY
The State University of New Jersey
School of Engineering
Department of Electrical and Computer Engineering

332:348 — Digital Signal Processing Laboratory

DSP Lab Manual

Sophocles J. Orfanidis

Spring 2011



Lab Schedule – Spring 2011

Week	Software Labs	Hardware Labs
1/18		
1/24	Lab0 – Introduction & Lab Requirements	
1/31	Lab1 – Sampling & Quantization – due 2/14	Lab2 – Code Composer Studio – group A
2/07		Lab2 – Code Composer Studio – group B
2/14	Lab3 – Filtering by Convolution – due 2/28	Lab4 – Delays & FIR Filtering – group A
2/21		Lab4 – Delays & FIR Filtering – group B
2/28	Lab5 – DTFT/DFT/FFT – due 3/28	
3/07		
3/14		
3/21		Lab6 – Digital Audio Effects – group A
3/28	Lab7 - IIR Filtering Applications – due 4/18	Lab6 – Digital Audio Effects – group B
4/04		Lab8 – Notch Filters & Equalizers – group A
4/11		Lab8 – Notch Filters & Equalizers – group B
4/18		
4/25		

Notes

1. Labs meet in room ELE-004.
2. Software labs are due on the dates posted on the above schedule. Late labs will not be accepted. In formatting your reports, please adhere to the lab guidelines described in Lab-0. Software labs should be submitted in person to the TAs and should not be left in the Tas' mailboxes.
3. Hardware lab sessions have a duration of two periods. Attendance in all hardware labs is required (it is not possible to get an "A" in the lab if one of these sessions is missed.) Due to the limited number of workstations, missed hardware labs cannot be made up.
4. For the hardware labs, each lab section has been split into two groups, A & B, that meet on alternate weeks as shown on the above schedule. The groups are as follows, divided according to student last names (please note that these may change until registration is closed):

Section	Group A	Group B
Section–1, M 3:20–6:20 PM	Alano – Lin	Litvin – Velez
Section–2, W 3:20–6:20 PM	Angad – Locorriere	Macor – Weidmann
Section–3, F 8:40–11:40 AM	Al Jabowbi – Marcus	Montemarano – Yu
TA	Baozhi Chen	Maja Skataric

Contents

Lab 0 – Introduction

The DSP lab has both a software and a hardware component. In the software component, students carry out a number of computer experiments written in C or MATLAB, illustrating some of the fundamental concepts and applications of digital signal processing, such as quantization and sampling, block processing by convolution, real-time filtering on a sample-by-sample basis, signal enhancement and noise reduction filters, direct, canonical, and cascade realizations of digital filters, spectral analysis by the DFT and FFT, the design of FIR and IIR digital filters, and digital audio effect applications, such as dynamic range control. The choice and number of computer experiments may vary from year to year.

The hardware part of the lab illustrates the programming of real-time processing algorithms on the Texas Instruments TMS320C6713 floating-point DSP. Programming of the DSP chip is done primarily in C (and some assembly) using the Code Composer Studio (CCS) integrated development environment. All of the C filtering functions in the textbook [1] translate with minor changes to the CCS environment.

The hardware experiments include aliasing and quantization effects; the circular buffer implementation of delays, FIR, and IIR filters; voice scramblers; the canceling of periodic interference with notch filters; wavetable generators; and several digital audio effects, such as comb filters, flangers, plain, allpass, and lowpass reverberators, Schroeder's reverberator, and several multi-tap, multi-delay, and stereo-delay type effects, as well as the Karplus-Strong string algorithm; various guitar distortion effects, such as fuzz and overdrive; and, parametric equalizer filters.

The lab assignments contain a short introduction of the required theory. More details, as well as several concrete C and MATLAB implementations, may be found in the book [1], which may be freely downloaded from the web page:

```
http://www.ece.rutgers.edu/~orfanidi/intro2sp/
```

0.1. Lab Guidelines

Attendance is *required* in all hardware-lab sessions (see the lab schedule at the beginning of this manual.) It is not possible to receive a grade of "A" if one of these sessions is missed. Due to the limited number of workstations, missed hardware labs cannot be made up. In addition, a 1–2 page lab report on each hardware lab must be submitted at the next hardware lab.

Students work in pairs on each workstation. Each lab section section has been split into two groups, A & B, that meet on alternate weeks (see lab schedule on the lab web page). Please make sure that you attend the right group (if in doubt please contact your TA).

The software labs are typically due in two weeks (except for lab-8) and require submission of a lab report. To be accepted and receive full credit, the reports must be prepared according to the following guidelines:

1. Include a 1–2 page discussion of the *purposes and objectives* of the experiment.
2. Include a short description of the *programming ideas* you used and a thorough *evaluation* of the results you obtained. All supporting *graphs* must be included in this part.
3. The source code of your programs should be attached at *the end* of your report as an Appendix. Xeroxed copies of the lab reports or figures are not acceptable.
4. Please, note that late labs will *not* be accepted. Labs should be handed to the TAs *in person* and not be left in the TAs mailboxes. Please check with your TA regarding where and when to submit your reports.
5. Students must work alone on their software lab projects. Collaboration with other students is not allowed. It is taken for granted that students accept and adhere to the Rutgers academic integrity policy described in:

```
http://academicintegrity.rutgers.edu/
```

0.2. Running C Programs

The software labs may be carried out on any available computer. The default computer facilities are in room ELE-103. Course computer accounts on `ece.rutgers.edu` will be assigned at the beginning of the semester or they may be obtained by contacting the system administrator of the ECE department, Mr. John Scafidi.

C programs may be compiled using the standard Unix C compiler `cc` or the GNU C compiler `gcc`. Both have the same syntax. It is recommended that C programs be structured in a modular fashion, linking the separate modules together at compilation time. Various versions of GCC, including a Windows version, and an online introduction may be found in the web sites:

```
http://gcc.gnu.org/
http://www.delorie.com/djgpp/
http://www.network-theory.co.uk/docs/gccintro/
```

Some reference books on C are given in [3]. As an example of using `gcc`, consider the following main program `sines.c`, which generates two noisy sinusoids and saves them (in ASCII format) into the data files `y1.dat` and `y2.dat`:

```
/* sines.c - noisy sinusoids */

#include <stdio.h>
#include <math.h>

#define L   100
#define f1  0.05
#define f2  0.03
#define A1  5
#define A2  A1

double gran();                     /* gaussian random number generator */

void main()
{
    int n;
    long iseed=2001;               /* gran requires iseed to be long int */
    double y1, y2, mean = 0.0, sigma = 1.0, pi = 4 * atan(1.0);
    FILE *fp1, *fp2;

    fp1 = fopen("y1.dat", "w");           /* open file y1.dat for write */
    fp2 = fopen("y2.dat", "w");           /* open file y2.dat for write */

    for (n=0; n<L; n++) {                /* iseed is passed by address */
        y1 = A1 * cos(2 * pi * f1 * n) + gran(mean, sigma, &iseed);
        y2 = A2 * cos(2 * pi * f2 * n) + gran(mean, sigma, &iseed);
        fprintf(fp1, "%12.6f\n", y1);
        fprintf(fp2, "%12.6f\n", y2);
        }

    fclose(fp1);
    fclose(fp2);
}
```

The noise is generated by calling the gaussian random number generator routine `gauss`, which is defined in the separate module `gran.c`:

```
/* gran.c - gaussian random number generator */
```

```
double ran();                             /* uniform generator */

double gran(mean, sigma, iseed)           /* x = gran(mean,sigma,&iseed) */
double mean, sigma;                       /* mean, variance = sigma^2 */
long *iseed;                              /* iseed passed by reference */
{
    double u = 0;
    int i;

    for (i = 0; i < 12; i++)              /* add 12 uniform random numbers */
        u += ran(iseed);

    return sigma * (u - 6) + mean;        /* adjust mean and variance */
}
```

In turn, `gran` calls a uniform random number generator routine, which is defined in the file `ran.c`:

```
/* ran.c - uniform random number generator in [0, 1) */

#define  a    16807                             /* a = 7^5 */
#define  m    2147483647                        /* m = 2^31 - 1 */
#define  q    127773                            /* q = m / a = quotient */
#define  r    2836                              /* r = m % a = remainder */

double ran(iseed)                               /* usage: u = ran(&iseed); */
long *iseed;                                    /* iseed passed by address */
{
   *iseed = a * (*iseed % q) - r * (*iseed / q);         /* update seed */

   if (*iseed < 0)                              /* wrap to positive values */
         *iseed += m;

   return (double) *iseed / (double) m;
}
```

The three programs can be compiled and linked into an executable file by the following command-line call of `gcc`:

```
gcc sines.c gran.c ran.c -o sines -lm            (unix version of gcc)
gcc sines.c gran.c ran.c -o sines.exe -lm        (MS-DOS version of gcc)
```

The command-line option `-lm` links the math library and must always be last. The option `-o` creates the executable file `sines` (or, `sines.exe` for MS-DOS.) If this option is omitted, the executable filename is `a.out` (or, `a.exe`) by default. Another useful option is the warning message option `-Wall`:

```
gcc -Wall sines.c gran.c ran.c -o sines -lm
```

If the command line is too long and tedious to type repeatedly, one can use a so-called response file, which may contain all or some of the command-line arguments. For example, suppose the file `argfile` contains the lines:

```
-Wall
sines.c
gran.c
ran.c
-o sines
-lm
```

Then, the following command will have the same effect as before, where the name of the response file must be preceded by the at-sign character `@`:

```
gcc @argfile
```

To compile only, without linking and creating an executable, we can use the command-line option `-c`:

```
gcc -c sines.c gran.c ran.c
```

This creates the object-code modules `*.o`, which can be subsequently linked into an executable as follows:

```
gcc -o sines sines.o gran.o ran.o -lm
```

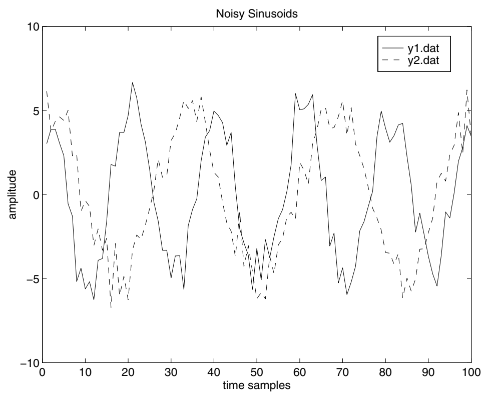
0.3. Using MATLAB

The plotting of data created by C or MATLAB programs can be done using MATLAB's extensive plotting facilities. Here, we present some examples showing how to load and plot data from data files, how to adjust axis ranges and tick marks, how to add labels, titles, legends, and change the default fonts, how to add several curves on the same graph, and how to create subplots.

Suppose, for example, that you wish to plot the noisy sinusoidal data in the files `y1.dat` and `y2.dat` created by running the C program `sines`. The following MATLAB code fragment will load and plot the data files:

```
load y1.dat;                            % load data into vector y1
load y2.dat;                            % load data into vector y2

plot(y1);                               % plot y1
hold on;                                % add next plot
plot(y2, 'r--');                        % plot y2 in red dashed style

axis([0, 100, -10, 10]);                % redefine axes limits
set(gca, 'ytick', -10:5:10);            % redefine yticks
legend('y1.dat', 'y2.dat');             % add legends
xlabel('time samples');                 % add labels and title
ylabel('amplitude');
title('Noisy Sinusoids');
```

The resulting plot is shown below. Note that the command `load y1.dat` strips off the extension part of the filename and assigns the data to a vector named `y1`.



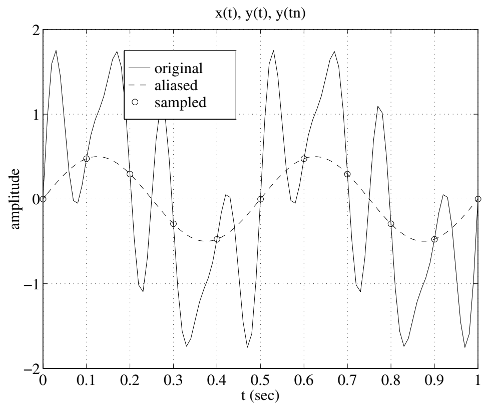
The command `hold on` leaves the first plot on and adds the second plot. The axis command increases the *y*-range in order to make space for the legends. The legends, labels, and title are in the default font and default size (e.g., Helvetica, size 10 for the Windows version.)

A more flexible and formatted way of reading and writing data from/to data files is by means of the commands `fscanf` and `fprintf`, in conjunction with `fopen` and `fclose`. They have similar usage as in C. See Ref. [2] for more details.

The next example is similar to what is needed in Lab-1. The example code below generates two signals $x(t)$ and $y(t)$ and plots them versus t. It also generates the time-samples $y(t_n)$ at the time instants $t_n = nT$. All three signals $x(t)$, $y(t)$, $y(t_n)$ span the same total time interval $[0, t_{\max}]$, but they are represented by arrays of different dimension ($x(t)$ and $y(t)$ have length 101, whereas $y(t_n)$ has length 11). All three can be placed on the same graph as follows:

```
tmax = 1;                                 % max time interval
Nmax = 100;                               % number of time instants
Dt = tmax/Nmax;                           % continuous-time increment
T = 0.1;                                  % sampling time interval

t  = 0:Dt:tmax;                                         % continuous t
x = sin(4*pi*t) + sin(16*pi*t) + 0.5 * sin(24*pi*t);    % signal x(t)
y = 0.5 * sin(4*pi*t);                                  % signal y(t)

tn = 0:T:tmax;                            % sampled version of t
yn = 0.5 * sin(4*pi*tn);                  % sampled version of y(t)

plot(t, x, t, y, '--', tn, yn, 'o');      % plot x(t), y(t), y(tn)

axis([0, 1, -2, 2])                       % redefine axis limits
set(gca, 'xtick', 0:0.1:1);               % redefine x-tick locations
set(gca, 'ytick', -2:1:2);                % redefine y-tick locations
set(gca, 'fontname', 'times');            % Times font
set(gca, 'fontsize', 16);                 % 16-point font size
grid;                                     % default grid

xlabel('t (sec)');
ylabel('amplitude');
title('x(t), y(t), y(tn)');

axes(legend('original', 'aliased', 'sampled'));   % legend over grid
```

The following figure shows the results of the above commands. Note that the x-axis tick marks have been redefined to coincide with the sampled time instants $t_n = nT$.



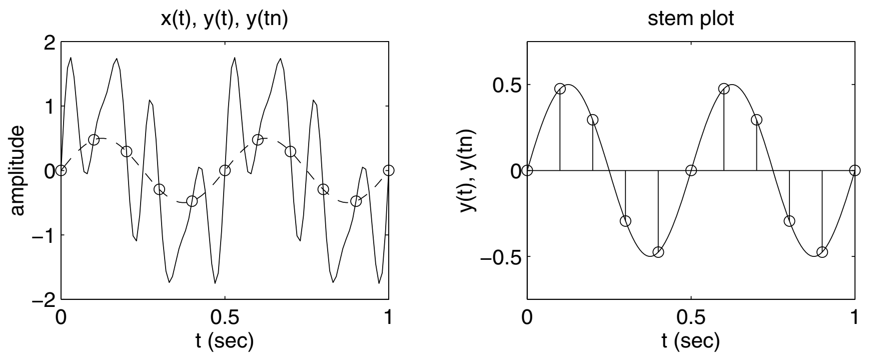
The `'o'` command plots the sampled signal $y(t_n)$ as circles. Without the `'o'`, the plot command would interpolate linearly between the 11 points of $y(t_n)$.

The font has been changed to Times-Roman, size 16, in order to make it more visible when the graph is scaled down for inclusion in this manual. The command `axes` creates a new set of axes containing the legends and superimposes them over the original grid (otherwise, the grid would be visible through the legends box.)

The next program segment shows the use of the command `subplot`, which is useful for arranging several graphs on one page. It also illustrates the `stem` command, which is useful for plotting sampled signals.

```
subplot(2, 2, 1);                           % upper left subplot

plot(t, x, t, y, '--', tn, yn, 'o');       % plot x(t), y(t), y(tn)

xlabel('t (sec)');
ylabel('amplitude');
title('x(t), y(t), y(tn)');

subplot(2, 2, 2);                           % upper right subplot

plot(t, y);                                 % plot y(t)
hold on;                                    % add next plot
stem(tn, yn);                               % stem plot of y(tn)

axis([0, 1, -0.75, 0.75]);                  % redefine axis limits

xlabel('t (sec)');
ylabel('y(t), y(tn)');
title('stem plot');
```

The resulting graph is shown below. Note that a 2×2 subplot pattern was used instead of a 1×2, in order to get a more natural aspect ratio.



Finally, we mention some MATLAB resources. Many of the MATLAB functions needed in the experiments are included in Appendix D of the text [1]. Many MATLAB on-line tutorials can be found at the following web sites:

```
http://www.mathworks.com/academia/student_center/tutorials/index.html
http://www.eece.maine.edu/mm/matweb.html
```

0.4. References

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

[2] MATLAB Documentation: `http://www.mathworks.com/help/techdoc/`

[3] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, 2nd ed., Prentice Hall, Englewood Cliffs, NJ, 1988.

S. P. Harbison and G. L. Steele, *C: A Reference Manual*, Prentice Hall, Englewood Cliffs, NJ, 1984.

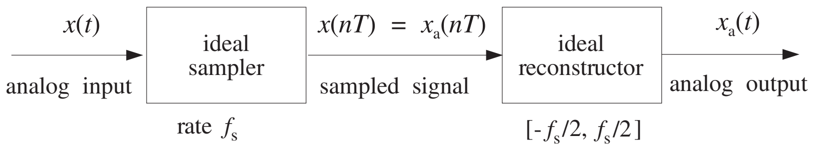
A. Kelly and I. Pohl, *A Book on C*, 2nd ed., Benjamin/Cummings, Redwood City, CA, 1990.

Lab 1 – Sampling and Quantization

1.1. Sampling and Aliasing

The aim of this lab is to demonstrate the effects of aliasing arising from improper sampling. This lab is to be done in MATLAB. The lab is based on Sections 1.3–1.4 of the text [1].

A given analog signal $x(t)$ is sampled at a rate f_s, the resulting samples $x(nT)$ are then reconstructed by an *ideal* reconstructor into the analog signal $x_a(t)$. Improper choice of f_s will result in different signals $x_a(t) \neq x(t)$, even though they agree at their sample values, that is, $x_a(nT) = x(nT)$. The procedure is illustrated in the following figure:



Lab Procedure

a. Consider an analog signal $x(t)$ consisting of three sinusoids of frequencies of 1 kHz, 4 kHz, and 6 kHz:

$$x(t) = \cos(2\pi t) + \cos(8\pi t) + \cos(12\pi t)$$

where t is in milliseconds. Show that if this signal is sampled at a rate of $f_s = 5$ kHz, it will be aliased with the following signal, in the sense that their sample values will be the same:

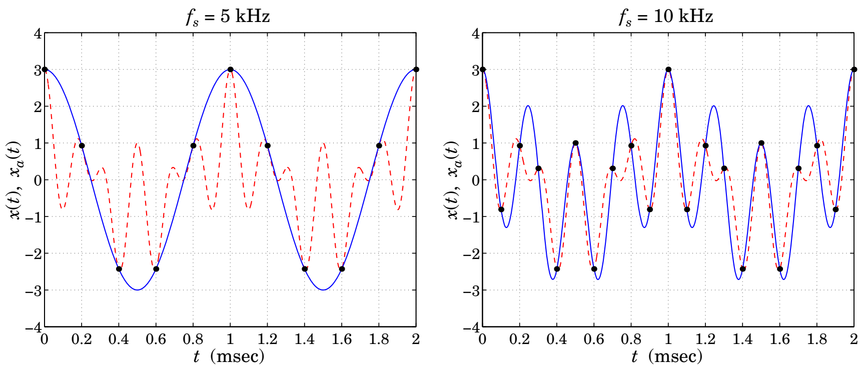
$$x_a(t) = 3\cos(2\pi t)$$

On the same graph, plot the two signals $x(t)$ and $x_a(t)$ versus t in the range $0 \le t \le 2$ msec. To this plot, add the time samples $x(t_n)$ and verify that $x(t)$ and $x_a(t)$ intersect precisely at these samples. These samples can be evaluated and plotted as follows:

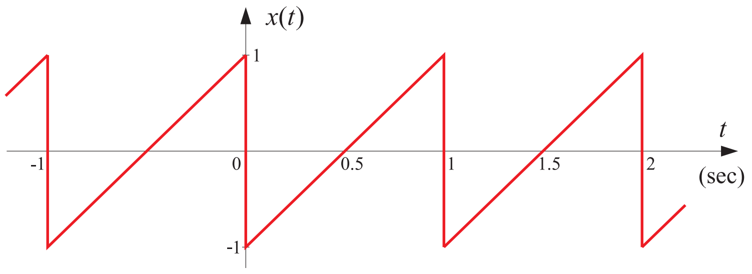
```
fs = 5; T = 1/fs;
tn = 0:T:2; xn = x(tn);
plot(tn, xn, '.');
```

b. Repeat part (a) with $f_s = 10$ kHz. In this case, determine the signal $x_a(t)$ with which $x(t)$ is aliased.

Plot both $x(t)$ and $x_a(t)$ on the same graph over the same range $0 \le t \le 2$ msec. Verify again that the two signals intersect at the sampling instants.



c. Consider a periodic sawtooth wave $x(t)$ with period $T_0 = 1$ sec shown below:



Mathematically, $x(t)$ is defined as follows over one period, that is, over the time interval $0 \le t \le 1$:

$$x(t) = \begin{cases} 2t-1, & \text{if} \quad 0 < t < 1 \\ 0, & \text{if} \quad t = 0 \quad \text{or} \quad t = 1 \end{cases} \tag{1.1}$$

where t is in units of seconds. This periodic signal admits a Fourier series expansion containing only sine terms with harmonics at the frequencies $f_m = m/T_0$, $m = 1, 2, 3, 4, \dots$ Hz:

$$x(t) = \sum_{m=1}^{\infty} b_m \sin(2\pi m t) = b_1 \sin(2\pi t) + b_2 \sin(4\pi t) + b_3 \sin(6\pi t) + \cdots \tag{1.2}$$

Using your prior knowledge of Fourier series, prove that the Fourier coefficients are given as follows:

$$b_m = -\frac{2}{\pi m}, \quad m = 1, 2, 3, 4, \dots$$

The reason why the signal $x(t)$ was defined to have the value zero at the discontinuity points has to do with a theorem that states that any finite sum of Fourier series terms will always pass through the mid-points of discontinuities.

d. The Fourier series of Eq. (1.2) can be thought of as the limit (in the least-squares sense) as $M \to \infty$ of the following signal consisting only of the first M harmonics:

$$x_M(t) = \sum_{m=1}^{M} b_m \sin(2\pi m t) \tag{1.3}$$

On the same graph, plot the sawtooth waveform $x(t)$ over three periods, that is, over $0 \le t \le 3$, together with the plot of $x_M(t)$ for $M = 5$.

To plot the above sawtooth waveform in MATLAB, you may use the functions `upulse` and `ustep`, which can be downloaded from the lab web page (see the link for the *supplementary* M-files.) The following code segment will generate and plot the above sawtooth waveform over one period:

```
x = inline('upulse(t-0.5,0,0.5,0) - upulse(t,0,0,0.5)');
t = linspace(0,1,1001);
plot(t, x(t));
```

In order to plot three periods, use the following example code that adds three shifted copies of the basic period defined in Eq. (1.1):

```
t = linspace(0,3,3001);
plot(t, x(t) + x(t-1) + x(t-2));
```

Repeat the above plot of the sawtooth waveform and $x_M(t)$ for the case $M = 10$, and then for $M = 20$.

The ripples that you see accumulating near the sawtooth discontinuities as M increases are an example of the so-called Gibbs phenomenon in Fourier series.

d. The sawtooth waveform $x(t)$ is now sampled at the rate of $f_s = 5$ Hz and the resulting samples are reconstructed by an *ideal* reconstructor. Using the methods of Example 1.4.6 of the text [1], show that the aliased signal $x_a(t)$ at the output of the reconstructor will have the form:

$$x_a(t) = a_1 \sin(2\pi t) + a_2 \sin(4\pi t)$$

Determine the coefficients a_1, a_2. On the same graph, plot one period of the sawtooth wave $x(t)$ together with $x_a(t)$. Verify that they agree at the five sampling time instants that lie within this period.

e. Assume, next, that the sawtooth waveform $x(t)$ is sampled at the rate of $f_s = 10$ Hz. Show now that the aliased signal $x_a(t)$ will have the form:

$$x_a(t) = a_1 \sin(2\pi t) + a_2 \sin(4\pi t) + a_3 \sin(6\pi t) + a_4 \sin(8\pi t)$$

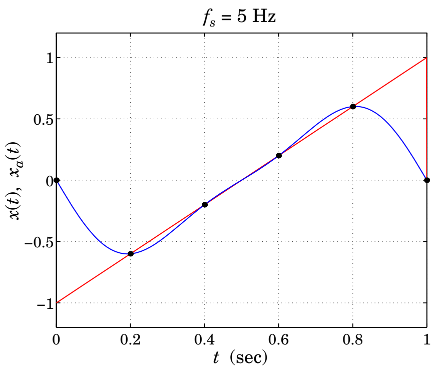
where the coefficients a_i are obtained by the condition that the signals $x(t)$ and $x_a(t)$ agree at the first four sampling instants $t_n = nT = n/10$ Hz, for $n = 1, 2, 3, 4$. These four conditions can be arranged into a 4×4 matrix equation of the form:

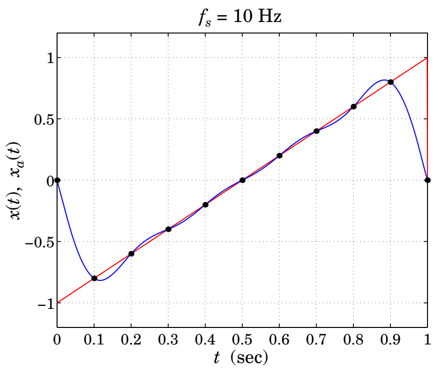
$$\begin{bmatrix} * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{bmatrix} = \begin{bmatrix} * \\ * \\ * \\ * \end{bmatrix}$$

Determine the numerical values of the starred entries and explain your approach. Then, using MATLAB, solve this matrix equation for the coefficients a_i. Once a_i are known, the signal $x_a(t)$ is completely defined.

On the same graph, plot one period of the sawtooth waveform $x(t)$ together with $x_a(t)$. Verify that they agree at the 10 sampling time instants that lie within this period.

Also, make another plot that displays $x(t)$ and $x_a(t)$ over three periods.





Please note the following caveat. In reducing the harmonics of the sawtooth waveform to the Nyquist interval $[-5, 5]$ Hz, you may conclude that the Nyquist frequency $f = 5$ Hz should also be included as a fifth term in the aliased signal, that is,

$$x_a(t) = a_1 \sin(2\pi t) + a_2 \sin(4\pi t) + a_3 \sin(6\pi t) + a_4 \sin(8\pi t) + a_5 \sin(10\pi t)$$

However, the sampled values of the last term vanish. Indeed, $a_5 \sin(10\pi t_n) = a_5 \sin(\pi n) = 0$. Therefore the aliased signals with both four and five terms would agree with the sawtooth wave at the time samples. We chose to work with the one with the four terms because it has lower frequency content than the one with five terms.

1.2. Quantization

The purpose of this lab is to study the quantization process by simulating the operation of A/D and D/A converters using the routines adc and dac that were discussed in the quantization chapter. A sampled sinusoidal signal is generated and then quantized using a 2's complement successive approximation ADC with rounding. In addition, the theoretical mean and mean square values of the quantization error:

$$\overline{e} = 0, \qquad \overline{e^2} = \frac{Q^2}{12} \tag{1.4}$$

will be tested by generating a *statistical* sample of quantization errors and computing the corresponding sample means. This will also give us the opportunity to work with random number generators.

This lab is best done in MATLAB. The needed functions dac.m and adc.m are also part of the *supplementary* M-files attached to this lab. The required theory is presented in Chapter 2 of the text [1].

Lab Procedure

a. Generate $L = 50$ samples of a sinusoidal signal $x(n) = A\cos(2\pi f n)$, $n = 0, 1, \dots, L-1$ of frequency $f = 0.02$ [cycles/sample] and amplitude $A = 8$. Using a 3-bit ($B = 3$) bipolar two's complement successive approximation A/D converter, as implemented by the routine adc, with full-scale range $R = 32$, quantize $x(n)$ and denote the quantized signal by $x_Q(n)$. This can be done as follows: for each $x(n)$, first call adc to generate the corresponding two's complement bit vector $\mathbf{b} = [b_1, b_2, \dots, b_B]$ and feed it into dac to generate the corresponding quantized value $x_Q(n)$.

   For $n = 0, 1, \dots, L-1$, print in three parallel columns the true analog value $x(n)$, the quantized value $x_Q(n)$, and the corresponding two's complement bit vector $\mathbf{b}$.

   On the same graph, plot the two signals $x(n)$ and $x_Q(n)$ versus n. Scale the vertical scales from $[-16, 16]$ and use 8 y-grid lines to indicate the 8 quantization levels.

b. Repeat part (a) using a $B = 4$ bit A/D converter. In plotting $x(n)$ and $x_Q(n)$, use the *same* vertical scales as above, namely, from $[-16, 16]$, but use 16 y-grid lines to show the 16 quantization levels.

c. What happens if the analog signal to be quantized has amplitude that *exceeds* the full-scale range of the quantizer? Most D/A converters will *saturate* to their largest (positive or negative) levels. To see this, repeat part (b) by taking the amplitude of the sinusoid to be $A = 20$.

d. Finally, we would like to test the theoretical values given in Eq. (1.4). It is not a good idea to use the above sinusoidal signal because it is periodic and our results will not improve statistically as we increase the size of the statistical sample. Therefore, here we use a signal $x(n)$ generated by MATLAB's random number generator rand, which generates random numbers in the range $[0, 1)$. By shifting by $1/2$ and multiplying by the desired full-scale range R, we can get a number in $[-R/2, R/2)$.

   Generate $L = 50$ random samples $x(n)$ with full-scale range $R = 32$ using the following for-loop, starting with any initial value of iseed:

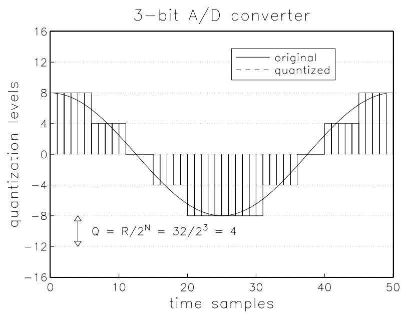
```
iseed = 2006;                     % initial seed is arbitrary
rand('state', iseed);             % initialize generator
x = R * (rand(L,1) - 0.5);        % length-L column vector
```

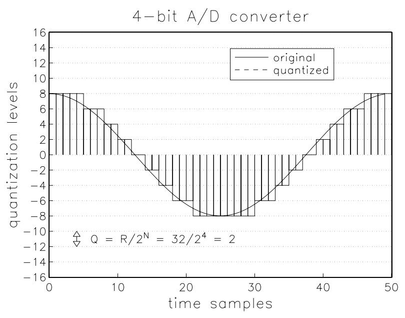
   Using the converter of part (a) with $B = 10$ bits, compute the L quantized values $x_Q(n)$ and quantization errors $e(n) = x_Q(n) - x(n)$, $n = 0, 1, \dots, L-1$. Then, compute the *experimental* values of the mean and mean-square errors:

$$\overline{e} = \frac{1}{L}\sum_{n=0}^{L-1} e(n), \qquad \overline{e^2} = \frac{1}{L}\sum_{n=0}^{L-1} e^2(n) \tag{1.5}$$

   and compare them with the theoretical values given by Eq. (1.4)

e. Repeat part (d) for $L = 200$ and $L = 500$. Discuss the improvement, if any, obtained by increasing the size of the statistical sample.





1.3. References

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

Lab 2 – TMS320C6713 DSK and Code Composer Studio

2.1. Introduction

The hardware experiments in the DSP lab are carried out on the Texas Instruments TMS320C6713 DSP Starter Kit (DSK), based on the TMS320C6713 floating point DSP running at 225 MHz. The basic clock cycle instruction time is $1/(225\text{ MHz}) = 4.44$ nanoseconds. During each clock cycle, up to eight instructions can be carried out in parallel, achieving up to $8\times 225 = 1800$ million instructions per second (MIPS).

The DSK board includes a 16MB SDRAM memory and a 512KB Flash ROM. It has an on-board 16-bit audio stereo codec (the Texas Instruments AIC23B) that serves both as an A/D and a D/A converter. There are four 3.5 mm audio jacks for microphone and stereo line input, and speaker and head-phone outputs. The AIC23 codec can be programmed to sample audio inputs at the following sampling rates:

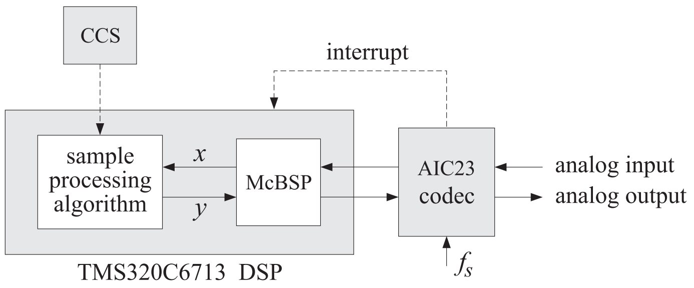
$$f_s = 8,\ 16,\ 24,\ 32,\ 44.1,\ 48,\ 96\ \text{kHz}$$

The ADC part of the codec is implemented as a multi-bit third-order noise-shaping delta-sigma converter (see Ch. 2 & 12 of [1] for the theory of such converters) that allows a variety of oversampling ratios that can realize the above choices of f_s. The corresponding oversampling decimation filters act as anti-aliasing prefilters that limit the spectrum of the input analog signals effectively to the Nyquist interval $[-f_s/2, f_s/2]$. The DAC part is similarly implemented as a multi-bit second-order noise-shaping delta-sigma converter whose oversampling interpolation filters act as almost ideal reconstruction filters with the Nyquist interval as their passband.

The DSK also has four user-programmable DIP switches and four LEDs that can be used to control and monitor programs running on the DSP.

All features of the DSK are managed by the Code Composer Studio (CCS). The CCS is a complete integrated development environment (IDE) that includes an optimizing C/C++ compiler, assembler, linker, debugger, and program loader. The CCS communicates with the DSK via a USB connection to a PC. In addition to facilitating all programming aspects of the C6713 DSP, the CCS can also read signals stored on the DSP's memory, or the SDRAM, and plot them in the time or frequency domains.

The following block diagram depicts the overall operations involved in all of the hardware experiments in the DSP lab. Processing is interrupt-driven at the sampling rate f_s, as explained below.



The AIC23 codec is configured (through CCS) to operate at one of the above sampling rates f_s. Each collected sample is converted to a 16-bit two's complement integer (a **short** data type in C). The codec actually samples the audio input in stereo, that is, it collects two samples for the left and right channels.

At each sampling instant, the codec combines the two 16-bit left/right samples into a single 32-bit unsigned integer word (an **unsigned int**, or **Uint32** data type in C), and ships it over to a 32-bit receive-register of the multichannel buffered serial port (McBSP) of the C6713 processor, and then issues an interrupt to the processor.

Upon receiving the interrupt, the processor executes an interrupt service routine (ISR) that implements a desired sample processing algorithm programmed with the CCS (e.g., filtering, audio effects, etc.). During the ISR, the following actions take place: the 32-bit input sample (denoted by x in the diagram) is read from the McBSP, and sent into the sample processing algorithm that computes the corresponding 32-bit output word (denoted by y), which is then written back into a 32-bit transmit-register of the

McBSP, from where it is transferred to the codec and reconstructed into analog format, and finally the ISR returns from interrupt, and the processor begins waiting for the next interrupt, which will come at the next sampling instant.

Clearly, all processing operations during the execution of the ISR must be completed in the time interval between samples, that is, $T = 1/f_s$. For example, if $f_s = 44.1$ kHz, then, $T = 1/f_s = 22.68$ μsec. With an instruction cycle time of $T_c = 4.44$ nsec, this allows $T/T_c = 5108$ cycles to be executed during each sampling instant, or, up to $8\times5108 = 40864$ instructions, or half of that per channel.

Resources

Most of the hardware experiments in the DSP lab are based on C code from the text [1] adapted to the CCS development environment. Additional experiments are based on the Chassaing-Reay text [2].

The web page of the lab, `http://www.ece.rutgers.edu/~orfanidi/ece348/`, contains additional resources such as tutorials and user guides. Some books on C and links to the GNU GCC C compiler are given in Ref. [5].

As a prelab, before you attend Lab-2, please go through the powerpoint presentations of Brown's workshop tutorial in Ref. [3], Part-1, and Dahnoun's chapters 1 & 3 listed in Ref. [4]. These will give you a pretty good idea of the TMS320C6000 architecture and features.

2.2. Template Program

You will begin with a basic talkthrough program, listed below, that simply reads input samples from the codec and immediately writes them back out. This will serve as a template on which to build more complicated sample processing algorithms by modifying the interrupt service routine `isr()`.

```
// template.c - to be used as starting point for interrupt-based programs
// ---------------------------------------------------------------------------------

#include "dsplab.h"          // DSK initialization declarations and function prototypes

short xL, xR, yL, yR;        // left and right input and output samples from/to codec
float g=1;                   // gain to demonstrate watch windows and GEL files

// here, add more global variable declarations, #define's, #include's, etc.
// ---------------------------------------------------------------------------------

void main()                  // main program executed first
{
  initialize();              // initialize DSK board and codec, define interrupts

  sampling_rate(8);          // 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
{
   read_inputs(&xL, &xR);    // read left and right input samples from codec

   yL = g * xL;              // replace these with your sample processing algorithm
   yR = g * xR;

   write_outputs(yL,yR);     // write left and right output samples to codec

   return;
}

// ---------------------------------------------------------------------------------
// here, add more functions to be called within isr() or main()
```

The template has three sections. In the top section, global variables are declared and defined, such as the left/right input/output audio samples x_L, x_R, y_L, y_R, whose scope is the entire file and are known to all functions in the file. Additional `#define` and `#include` statements, such as `#include <math.h>`, and additional global variable declarations may be added in this section.

The second section consists of the function `main()`, which is executed first, and performs the initialization of the DSK board, sets the sampling rate, selects the audio input, and then goes into an infinite loop waiting for an interrupt. Upon receiving the interrupt, it jumps to the function `isr()`. Additional local variables and other preliminary operations, such as the zeroing of delay-line buffers, may be added in this section before the `wait(1)` statement.

The third section consists of the interrupt service routine `isr()`, which implements the desired sample processing algorithm. Note that the keyword **interrupt** has been added to the C language implementation of the CCS. In the template file, the ISR function reads the left/right input samples, process them by multiplying them by a gain, sends them to the output, and returns back to `main()`.

The reading and writing of the input and output samples are done with the help of the functions `read_inputs()` and `write_outputs()`, which are declared in the header file `dsplab.h` and defined in `dsplab.c`. These two files must always be included in your programs and reside in the common directory `C:\dsplab\common\`.

Besides the above three basic sections, other sections may be added that define additional functions to be called within `isr()` or `main()`.

Working with CCS

For each application to be run on the C6713 processor, one must create a “project” in the Code Composer Studio, which puts together all the information about the required C source files, header files, and C libraries, including all the compiler and linker build options.

To save you time, the project file, `template.pjt`, for the above template has already been created, and may be simply edited for all other projects. To proceed, copy the following three files from the template directory `C:\dsplab\template\`

```
template.c
template.pjt
template.gel
```

into your temporary working directory, e.g., `C:\labuser\tempwork\`, and double-click the project file, `template.pjt`, which will open in a ordinary text editor. The first few lines of that file are shown below:

```
[Project Settings]
ProjectDir="C:\dsplab\template\"
ProjectType=Executable
CPUFamily=TMS320C67XX
Tool="Compiler"
Tool="CustomBuilder"
Tool="DspBiosBuilder"
Tool="Linker"
Config="Debug"
Config="Release"

[Source Files]
Source="C:\CCStudio_v3.1\C6000\cgtools\lib\rts6700.lib"
Source="C:\CCStudio_v3.1\C6000\csl\lib\csl6713.lib"
Source="C:\CCStudio_v3.1\C6000\dsk6713\lib\dsk6713bsl.lib"
Source="C:\dsplab\common\dsplab.c"
Source="C:\dsplab\common\vectors.asm"
Source="template.c"
```

Only the second and bottom lines in the above listing need to be edited. First, edit the project directory entry to your working directory, e.g.,

```
ProjectDir="C:\labuser\tempwork\"
```

Alternatively, you may delete that line—it will be recreated by CCS when you load the project. Then, edit the source-file line `Source="template.c"` to your new project's name, e.g.,

```
Source="new_project.c"
```

Finally, rename the three files with your new names, e.g.,

```
new_project.c
new_project.pjt
new_project.gel
```

Next, turn on the DSK kit and after the initialization beep, open the CCS application by double-clicking on the CCS desktop icon. Immediately after it opens, use the keyboard combination "ALT+C" (or the menu item *Debug -> Connect*) to connect it to the processor. Then, with the menu item *Project -> Open* or the key combination "ALT+P O", open the newly created project file by navigating to the project's directory, e.g., `C:\labuser\tempwork\`. Once the project loads, you may edit the C source file to implement your algorithm. Additional C source files can be added to your project by the keyboard combination "ALT+P A" or the menu choices *Project -> Add Files to Project*.

Set up CCS to automatically load the program after building it, with the menu commands: *Option -> Customize -> Program/Project Load -> Load Program After Build*. The following key combinations or menu items allow you to compile and load your program, run or halt the program:

```
compile & load:   F7,         Project -> Build
run program:      F5,         Debug -> Run
halt program:     Shift+F5,   Debug -> Halt
```

It is possible that the first time you try to build your program you will get a warning message:

```
warning: creating .stack section with default size of 400 (hex) words
```

In such case, simply rebuild the project, or, in the menu item *Project -> Build Options -> Linker*, enter a value such as `0x500` in the stack entry.

When you are done, please remember to save and close your project with the keyboard combinations "ALT+P S" and "ALT+P C", and save your programs in your account on ECE.

Lab Procedure

a. Copy the template files into your temporary working directory, edit the project's directory as described above, and build the project in CCS. Connect your MP3 player to the line input of the DSK board and play your favorite song, or, you may play one of the wave files in the directory: `c:\dsplab\wav`.

b. Review the template project's build options using the menu commands: *Project -> Build Options*. In particular, review the Basic, Advanced, and Preprocessor options for the Compiler, and note that the optimization level was set to `none`. In future experiments, this may be changed to `-o2` or `-o3`.

   For the Linker options, review the Basic and Advanced settings. In particular, note that the default output name `a.out` can be changed to anything else. Note also the library include paths and that the standard included libraries are:

   ```
   rts6700.lib       (run-time library)
   dsk6713bsl.lib    (board support library)
   csl6713.lib       (chip support library)
   ```

   The run-time library must always be included. The board support library (BSL) contains functions for managing the DSK board peripherals, such as the codec. The chip support library (CSL) has functions for managing the DSP chip's features, such as reading and writing data to the chip's McBSP. The user manuals for these may be found on the TI web site listed on the lab's web page.

c. The gain parameter g can be controlled in real-time in two ways: using a watch window, or using a GEL file. Open a watch window using the menu item: *View -> Watch Window*, then choose *View -> Quick Watch* and enter the variable g and add it to the opened watch window using the item *Add to Watch*. Run the program and click on the g variable in the watch window and enter a new value, such as $g = 0.5$ or $g = 2$, and you will hear the difference in the volume of the output.

d. Close the watch window and open the GEL file, `template.gel`, with the menu *File -> Load GEL*. In the *GEL* menu of CCS a new item called "gain" has appeared. Choose it to open the gain slider. Run the program and move the slider to different positions. Actually, the slider does not represent the gain g itself, but rather the integer increment steps. The gain g changes by $1/10$ at each step. Open the GEL file to see how it is structured. You may use that as a template for other cases.

e. Modify the template program so that the output pans between the left and right speakers every 2 seconds, i.e., the left speaker plays for 2 sec, and then switches to the right speaker for another 2 sec, and so on. There are many ways of doing this, for example, you may replace your ISR function by

```
#define D 16000     // represents 2 sec at fs = 8 kHz
short d=0;          // move these before main()

interrupt void isr()
{
   read_inputs(&xL, &xR);

   yL = (d < D)  * xL;
   yR = (d >= D) * xR;

   if (++d >= 2*D) d=0;

   write_outputs(yL,yR);

   return;
}
```

Rebuild your program with these changes and play a song. In your lab write-up explain why and how this code works.

2.3. Aliasing

This part demonstrates aliasing effects. The smallest sampling rate that can be defined is 8 kHz with a Nyquist interval of $[-4, 4]$ kHz. Thus, if a sinusoidal signal is generated (e.g. with MATLAB) with frequency outside this interval, e.g., $f = 5$ kHz, and played into the line-input of the DSK, one might expect that it would be aliased with $f_a = f - f_s = 5 - 8 = -3$ kHz. However, this will not work because the antialiasing oversampling decimation filters of the codec filter out any such out-of-band components before they are sent to the processor.

An alternative is to decimate the signal by a factor of 2, i.e., dropping every other sample. If the codec sampling rate is set to 8 kHz and every other sample is dropped, the effective sampling rate will be 4 kHz, with a Nyquist interval of $[-2, 2]$ kHz. A sinusoid whose frequency is outside the decimated Nyquist interval $[-2, 2]$ kHz, but inside the true Nyquist interval $[-4, 4]$ kHz, will not be cut off by the antialiasing filter and will be aliased. For example, if $f = 3$ kHz, the decimated sinusoid will be aliased with $f_a = 3 - 4 = -1$ kHz.

Lab Procedure

Copy the template programs to your working directory. Set the sampling rate to 8 kHz and select line-input. Modify the template program to output every other sample, with zero values in-between. This can be accomplished in different ways, but a simple one is to define a "sampling pulse" periodic signal whose values alternate between 1 and 0, i.e., the sequence $[1, 0, 1, 0, 1, 0, \dots]$ and multiply the input samples by that sequence. The following simple code segment implements this idea:

```
yL = pulse * xL;
yR = pulse * xR;

pulse = (pulse==0);
```

where `pulse` must be globally initialized to 1 before `main()` and `isr()`. Why does this work? Next, rebuild the new program with CCS.

Open MATLAB and generate three sinusoids of frequencies $f_1 = 1$ kHz, $f_2 = 3$ kHz, and $f_3 = 1$ kHz, each of duration of 1 second, and concatenate them to form a 3-second signal. Then play this out of the PCs sound card using the `sound()` function. For example, the following MATLAB code will do this:

```
fs = 8000; f1 = 1000; f2 = 3000; f3 = 1000;
L = 8000; n = (0:L-1);
A = 1/5;                         % adjust playback volume

x1 = A * cos(2*pi*n*f1/fs);
x2 = A * cos(2*pi*n*f2/fs);
x3 = A * cos(2*pi*n*f3/fs);

sound([x1,x2,x3], fs);
```

a. Connect the sound card's audio output to the line-input of the DSK and rebuild/run the CCS downsampling program after commenting out the line:

```
pulse = (pulse==0);
```

This disables the downsampling operation. Send the above concatenated sinusoids to the DSK input and you should hear three distinct 1-sec segments, with the middle one having a higher frequency.

b. Next, uncomment the above line so that downsampling takes place and rebuild/run the program. Send the concatenated sinusoids to the DSK and you should hear all three segments as though they have the same frequency (because the middle 3 kHz one is aliased with other ones at 1 kHz). You may also play your favorite song to hear the aliasing distortions, e.g., out of tune vocals.

c. Set the codec sampling rate to 44 kHz and repeat the previous two steps. What do you expect to hear in this case?

d. To confirm the antialiasing prefiltering action of the codec, replace the first two lines of the above MATLAB code by the following two:

```
fs = 16000; f1 = 1000; f2 = 5000; f3 = 1000;
L = 16000; n = (0:L-1);
```

Now, the middle sinusoid has frequency of 5 kHz and it should be cutoff by the antialiasing prefilter. Set the sampling rate to 8 kHz, turn off the downsampling operation, rebuild and run your program, and send this signal through the DSK, and describe what you hear.

2.4. Quantization

The DSK's codec is a 16-bit ADC/DAC with each sample represented by a two's complement integer. Given the 16-bit representation of a sample, $[b_1 b_2 \cdots b_{16}]$, the corresponding 16-bit integer is given by

$$x = (-b_1 2^{-1} + b_2 2^{-2} + b_3 2^{-3} + \cdots + b_{16} 2^{-16}) 2^{16} \tag{2.1}$$

The MSB bit b_1 is the sign bit. The range of representable integers is: $-32768 \le x \le 32767$. As discussed in Ch. 2 of Ref. [1], for high-fidelity audio at least 16 bits are required to match the dynamic range of human hearing; for speech, 8 bits are sufficient. If the audio or speech samples are quantized to less than 8 bits, quantization noise will become audible.

The 16-bit samples can be requantized to fewer bits by a right/left bit-shifting operation. For example, right shifting by 3 bits will knock out the last 3 bits, then left shifting by 3 bits will result in a 16-bit number whose last three bits are zero, that is, a 13-bit integer. These operations are illustrated below:

$$[b_1, b_2, \ldots, b_{13}, b_{14}, b_{15}, b_{16}] \quad \Rightarrow \quad [0, 0, 0, b_1, b_2, \ldots, b_{13}] \quad \Rightarrow \quad [b_1, b_2, \ldots, b_{13}, 0, 0, 0]$$

Lab Procedure

a. Modify the basic template program so that the output samples are requantized to B bits, where $1 \le B \le 16$. This requires right/left shifting by $L = 16 - B$ bits, and can be implemented very simply in C as follows:

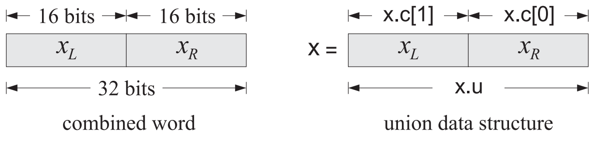
```
yL = (xL >> L) << L;
yR = (xR >> L) << L;
```

Start with $B = 16$, set the sampling rate to 8 kHz, and rebuild/run the program. Send a wave file as input and listen to the output.

b. Repeat with the following values: $B = 8, 6, 4, 2, 1$, and listen to the gradual increase in the quantization noise.

2.5. Data Transfers from/to Codec

We mentioned in the introduction that the codec samples the input in stereo, combines the two 16-bit left/right samples x_L, x_R into a single 32-bit unsigned integer word, and ships it over to a 32-bit receive-register of the McBSP of the C6713 processor. This is illustrated below.



The packing and unpacking of the two 16-bit words into a 32-bit word is accomplished with the help of a union data structure (see Refs. [2,3]) defined as follows:

```
union {               // union structure to facilitate 32-bit data transfers
   Uint32 u;          // both channels packed as codec.u = 32-bits
   short c[2];        // left-channel = codec.c[1], right-channel = codec.c[0]
} codec;
```

The two members of the data structure share a common 32-bit memory storage. The member `codec.u` contains the 32-bit word whose upper 16 bits represent the left sample, and its lower 16 bits, the right sample. The two-dimensional short array member `codec.c` holds the 16-bit right-channel sample in its first component, and the left-channel sample in its second, that is, we have:

```
xL = codec.c[1];
xR = codec.c[0];
```

The functions `read_inputs()` and `write_outputs()`, which are defined in the common file `dsplab.c`, use this structure in making calls to low-level McBSP read/write functions of the chip support library. They are defined as follows:

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

void read_inputs(short *xL, short *xR)                // read left/right channels
{
   codec.u = MCBSP_read(DSK6713_AIC23_DATAHANDLE);    // read 32-bit word

   *xL = codec.c[1];                                  // unpack the two 16-bit parts
   *xR = codec.c[0];
}

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

void write_outputs(short yL, short yR)              // write left/right channels
{
   codec.c[1] = yL;                                 // pack the two 16-bit parts
   codec.c[0] = yR;                                 // into 32-bit word

   MCBSP_write(DSK6713_AIC23_DATAHANDLE,codec.u);     // output left/right samples
}

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

Lab Procedure

The purpose of this lab is to clarify the nature of the union data structure. Copy the template files into your working directory, rename them `unions.*`, and edit the project file by keeping in the source-files section only the run-time library and the main function below.

```
// unions.c - test union structure

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

void main(void)
{
   unsigned int v;
   short xL,xR;

   union {
      unsigned int u;
      short c[2];
   } x;

   xL = 0x1234;
   xR = 0x5678;
   v  = 0x12345678;

   printf("\n%x  %x  %d  %d\n", xL,xR, xL,xR);

   x.c[1] = xL;
   x.c[0] = xR;
   printf("\n%x  %x  %x  %d  %d\n", x.u, x.c[1], x.c[0], x.c[1], x.c[0]);

   x.u = v;
   printf("%x  %x  %x  %d  %d\n", x.u, x.c[1], x.c[0], x.c[1], x.c[0]);

   x.u = (((int) xL)<<16 | ((int) xR) & 0x0000ffff);
   printf("%x  %x  %x  %d  %d\n", x.u, x.c[1], x.c[0], x.c[1], x.c[0]);
```

The program defines first a union structure variable `x` of the codec type. Given two 16-bit left/right numbers `xL,xR` (specified as 4-digit hex numbers), it defines a 32-bit unsigned integer `v` which is the concatenation of the two. The first `printf` statement prints the two numbers `xL,xR` in hex and decimal format. Note that the hex printing conversion operator `%x` treats the numbers as unsigned (some caution is required when printing negative numbers), whereas the decimal operator `%d` treats them as signed integers.

Next, the numbers `xL,xR` are assigned to the array members of the union `x`, such that `x.c[1] = xL` and `x.c[0] = xR`, and the second `printf` statement prints the contents of the union `x`, verifying that the 32-bit member `x.u` contains the concatenation of the two numbers with `xL` occupying the upper 16 bits, and `xR`, the lower 16 bits. Explain what the other two `printf` statements do.

Build and run the project (you may have to remove the file `vectors.asm` from the project's list of files). The output will appear in the `stdout` window at the bottom of the CCS. Alternatively, you may run this outside CCS using GCC. To do so, open a DOS window in your working directory and type the DOS command `djgpp`. This establishes the necessary environment variables to run GCC, then, run the following GCC command to generate the executable file `unions.exe`:

```
gcc unions.c -o unions.exe -lm
```

Repeat the run with the following choice of input samples:

```
xL = 0x1234;
xR = 0xabcd;
v = 0x1234abcd;
```

Explain the outputs of the print statements in this case by noting the following properties, which you should prove in your report:

$$(\texttt{0xffff0000})_{\text{unsigned}} = 2^{32} - 2^{16}$$

$$(\texttt{0xffffabcd})_{\text{unsigned}} = 2^{32} + (\texttt{0xabcd})_{\text{signed}}$$

$$(\texttt{0xffffabcd})_{\text{signed}} = (\texttt{0xabcd})_{\text{signed}}$$

2.6. Guitar Distortion Effects

In all of the experiments of Lab-2, the input/output maps are memoryless. We will study implementation of delays in a later lab. A memoryless mapping can be linear but time-varying, as was for example the case of panning between the speakers or the AM/FM wavetable experiments discussed discussed in another lab. The mapping can also be nonlinear.

Many guitar distortion effects combine delay effects with such nonlinear maps. In this part of Lab-1, we will study only some nonlinear maps in which each input sample x is mapped to an output sample y by a nonlinear function $y = f(x)$. Typical examples are hard clipping (called fuzz) and soft clipping that tries to emulated the nonlinearities of tube amplifiers. A typical nonlinear function is $y = \tanh(x)$. It has a sigmoidal shape that you can see by the quick MATLAB plot:

```
fplot('tanh(x)', [-4,4]); grid;
```

As suggested in Ref. [6], by keeping only the first two terms in its Taylor series expansion, that is, $\tanh(x) \approx x - x^3/3$, we may define a more easily realizable nonlinear function with built-in soft clipping:

$$y = f(x) = \begin{cases} +2/3, & x \geq 1 \\ x - x^3/3, & -1 \leq x \leq 1 \\ -2/3, & x \leq -1 \end{cases} \tag{2.2}$$

This can be plotted easily with

```
fplot('(abs(x)<1).*(x-1/3*x.^3) + sign(x).*(abs(x)>=1)*2/3', [-4,4]); grid;
```

The threshold value of $2/3$ is chosen so that the function $f(x)$ is continuous at $x = \pm 1$. To add some flexibility and to allow a variable threshold, we consider the following modification:

$$y = f(x) = \begin{cases} +\alpha c, & x \geq c \\ x - \beta c(x/c)^3, & -c \leq x \leq c \\ -\alpha c, & x \leq -c \end{cases}, \qquad \beta = 1 - \alpha \tag{2.3}$$

where we assume that $c > 0$ and $0 < \alpha < 1$. The choice $\beta = 1 - \alpha$ is again dictated by the continuity requirement at $x = \pm c$. Note that setting $\alpha = 1$ gives the hard-thresholding, fuzz, effect:

$$y = f(x) = \begin{cases} +c, & x \geq c \\ x, & -c \leq x \leq c \\ -c, & x \leq -c \end{cases} \tag{2.4}$$

Lab Procedure

First, run the above two `fplot` commands in MATLAB to see what these functions look like. The following program is a modification of the basic `template.c` program that implements Eq. (2.3):

```
// soft.c - guitar distortion by soft thresholding
// ----------------------------------------------------------------------------------

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

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

#define a 0.67               // approximates the value 2/3
#define b (1-a)

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

int x, y, on=1, c=2048;      // on/off variable and initial threshold c

int f(int);                  // function declaration

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

void main()                  // main program executed first
{
  initialize();              // initialize DSK board and codec, define interrupts

  sampling_rate(16);         // 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
{
   read_inputs(&xL, &xR);    // read left and right input samples from codec

   if (on) {
      yL = (short) f((int) xL);  yL = yL << 1;     // amplify by factor of 2
      yR = (short) f((int) xR);  yR = yR << 1;
      }
   else
      {yL = xL; yR = xR;}

   write_outputs(yL,yR);     // write left and right output samples to codec

   return;
}

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

int f(int x)
{
   float y, xc = x/c;             // this y is local to f()

   y = x * (1 - b * xc * xc);

   if (x>c)  y =  a*c;            // force the threshold values
   if (x<-c) y = -a*c;

   return ((int) y);
}

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

a. Create a project for this program. In addition, create a GEL file that has two sliders, one for the `on` variable that turns the effect on or off in real time, and another slider for the threshold parameter `c`. Let `c` vary over the range $[0, 2^{14}]$ in increments of 512.

   Build and run the program, load the gel file, and display the two sliders. Then, play your favorite guitar piece and vary the slider parameters to hear the changes in the effect. (The wave file `turn-turn3.wav` in the directory `c:\dsplab\wav` is a good choice.)

b. Repeat the previous part by turning off the nonlinearity (i.e., setting $\alpha = 1$), which reduces to a fuzz effect with hard thresholding.

2.7. 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] D. R. Brown III, 2009 Workshop on Digital Signal Processing and Applications with the TMS320C6713 DSK, Parts 1 & 2, available online from:
`http://spinlab.wpi.edu/courses/dspworkshop/dspworkshop_part1_2009.pdf`
`http://spinlab.wpi.edu/courses/dspworkshop/dspworkshop_part2_2009.pdf`

[4] N. Dahnoun, "DSP Implementation Using the TMS320C6711 Processors," contained in the Texas Instruments "C6000 Teaching Materials" CD ROM, 2002-04, and available online from TI:
`http://www.ti.com/ww/cn/uprogram/share/ppt/c6000/Chapter1.ppt`
`http://www.ti.com/ww/cn/uprogram/share/ppt/c6000/Chapter2.ppt`
`http://www.ti.com/ww/cn/uprogram/share/ppt/c6000/Chapter3.ppt`

[5] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, 2nd ed., Prentice Hall, Englewood Cliffs, NJ, 1988.

S. P. Harbison and G. L. Steele, *C: A Reference Manual*, Prentice Hall, Englewood Cliffs, NJ, 1984.

A. Kelly and I. Pohl, *A Book on C*, 2nd ed., Benjamin/Cummings, Redwood City, CA, 1990.

GNU gcc, `http://gcc.gnu.org/`
DJGPP - Windows version of GCC, `http://www.delorie.com/djgpp/`
GCC Introduction, `http://www.network-theory.co.uk/docs/gccintro/`

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

Lab 3 – Filtering by Convolution

3.1. Introduction

Depending on the application and hardware, a digital filtering operation can be organized to operate either on a *block* basis or a *sample-by-sample* basis. In the block processing case, the input signal is considered to be one big block of signal samples. The block is filtered by *convolving* it with the filter, generating the output signal as another big block of samples.

If the input signal is very long or infinite in duration, this method requires modification, for example, breaking the input into multiple blocks.

An alternative approach is to organize the processing on a sample-by-sample basis, such that whenever an input sample arrives, it gets filtered producing the corresponding output sample. This approach is very useful in real-time applications involving very long input signals. It is also useful in adaptive filtering applications where the filter itself changes after each filtering operation. Moreover, it is efficiently implementable with present day special purpose DSP chips, such as the Texas Instruments, Analog Devices, or Motorola DSP chips.

In this lab, you will study filtering by convolution methods. In the next, hardware, lab you will implement filtering on sample by sample basis.

This lab may be carried out in MATLAB or C. The theoretical background on convolution and FIR filtering and the necessary routines and their usage are contained in Chapter 4 of the text [1].

3.2. Convolution

The convolution of a filter, h_n, $n = 0,1,\dots,M$, of order M, and a finite-duration causal signal, x_n, $n = 0,1,\dots,L-1$, of length L, was determined in Eq. (4.1.16) of the text [1] to be:

$$y_n = \sum_{m=\max(0,n-L+1)}^{\min(n,M)} h_m x_{n-m}\,, \quad n = 0,1,\dots,L+M-1 \tag{3.1}$$

Write a MATLAB function named `myconv` (to avoid confusion with the built-in function `conv`), that implements Eq. (3.1) and is patterned after the C version discussed in Sect. 4.1.9 of [1]. It must have usage:

```
y = myconv(h,x);
```

where `h,x,y` are the filter, input, and output vectors. The function must be able to accept the input vectors `h,x` whether they are entered as rows or columns. Moreover, the output vector `y` must match the type of the input vector `x`, that is, row or column.

Lab Procedure

a. Consider an integrator-like filter defined by the I/O equation:

$$y(n) = 0.1\big[x(n) + x(n-1) + x(n-2) + \cdots + x(n-14)\big]$$

Such a filter accumulates (integrates) the present and past 14 samples of the input signal. The factor 0.1 represents only a convenient scale factor for this experiment. It follows that the impulse response of this filter will be

$$h_n = \begin{cases} 0.1, & \text{for } 0 \le n \le 14 \\ 0, & \text{otherwise} \end{cases}$$

To observe the steady-state response of this filter, as well as the input-on and input-off transients, consider a *square wave* input signal x_n of length $L = 200$ and period of $K = 50$ samples. Such a signal may be defined by the simple for-loop (given in C):

```
for (n=0; n<L; n++)
       if (n%K < K/2)        /* n%K is the MOD operation */
              x[n] = 1;
       else
              x[n] = 0;
```

Using your function `myconv`, compute the output signal y_n versus n and plot it on the same graph with x_n. As the square wave periodically goes on and off, you can observe the on-transient, steady-state, and off-transient behavior of the filter. Verify the your function produces the same results as the built-in function `conv`.

b. Repeat part (a) for the filter:

$$h_n = \begin{cases} 0.25(0.75)^n, & \text{for } 0 \le n \le 14 \\ 0, & \text{otherwise} \end{cases}$$

This filter acts more like an RC-type integrator than an accumulator.

c. Repeat part (a) for the filter with transfer function:

$$H(z) = \frac{1}{5}(1 - z^{-1})^5$$

This filter acts as 5-fold differentiator. Thus, in its steady-state it differentiates a constant to zero. As the square wave goes on and off you may observe this differentiating action as well as the transients.

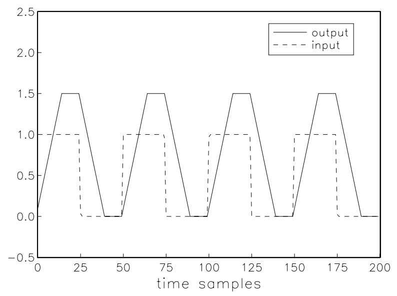
d. To demonstrate the concepts of impulse response, linearity, and time-invariance, consider a filter with finite impulse response $h_n = (0.95)^n$, for $0 \le n \le 24$. The input signal

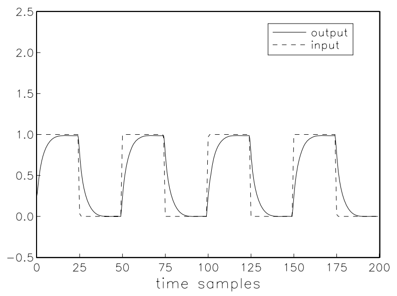
$$x(n) = \delta(n) + 2\delta(n-40) + 2\delta(n-70) + \delta(n-80), \qquad n = 0, 1, \ldots, 120$$

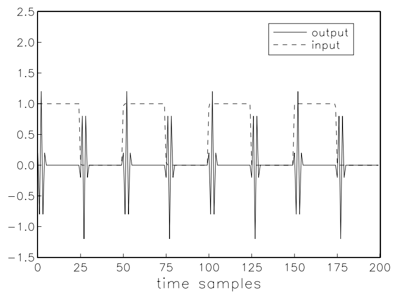
consists of four impulses of the indicated strengths occurring at the indicated time instances. Note that the first two impulses are separated by more than the duration of the filter, whereas the last two are separated by less. Using your function `myconv`, compute the filter output y_n for $0 \le n \le 120$ and plot it on the same graph with x_n. Comment on the resulting output with regard to linearity and time invariance.

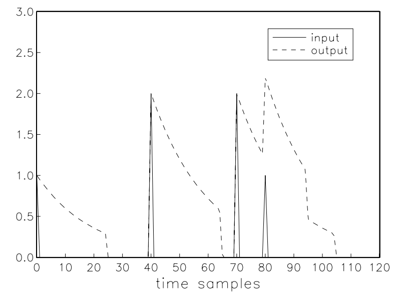
In generating the above input signal, you may find the following C function `delta.c` useful:

```
/* delta.c - delta function */

double delta(n)
int n;
{
       if (n == 0)
              return 1;
       else
              return 0;
}
```









3.3. Filtering of Noisy Signals

A signal $x(n)$ is the sum of a desired signal $s(n)$ and interference $v(n)$:

$$x(n) = s(n) + v(n)$$

where

$$s(n) = \sin(\omega_2 n)\,, \quad v(n) = \sin(\omega_1 n) + \sin(\omega_3 n)$$

with

$$\omega_1 = 0.05\pi\,, \quad \omega_2 = 0.20\pi\,, \quad \omega_3 = 0.35\pi \qquad \text{[radians/sample]}$$

In order to remove $v(n)$, the signal $x(n)$ is filtered through a bandpass FIR lowpass filter that is designed to pass the frequency ω_2 and reject the interfering frequencies ω_1, ω_3. An example of such a filter of order $M = 100$ can be designed with the methods of Chapter 11 of [1] (using a Hamming-window design) and has impulse response:

$$h(n) = w(n)\left[\frac{\sin\big(\omega_b(n - M/2)\big) - \sin\big(\omega_a(n - M/2)\big)}{\pi(n - M/2)}\right], \quad 0 \le n \le M$$

where $\omega_a = 0.15\pi$, $\omega_b = 0.25\pi$, and $w(n)$ is the Hamming window:

$$w(n) = 0.54 - 0.46\cos\left(\frac{2\pi n}{M}\right), \quad 0 \le n \le M$$

It has an effective passband $[\omega_a, \omega_b] = [0.15\pi, 0.25\pi]$. To avoid a computational issue at $n = M/2$, you may use MATLAB's built-in function `sinc`, which is defined as follows:

$$\text{sinc}(x) = \frac{\sin(\pi x)}{\pi x}$$

Lab Procedure

a. On the same graph plot $x(n)$ and $s(n)$ versus n

b. Filter $x(n)$ through the filter $h(n)$ using MATLAB's built-in function `filter`, and plot the filtered output $y(n)$, together with $s(n)$. Apart from an overall delay introduced by the filter, $y(n)$ should resemble $s(n)$ after the M initial transients.

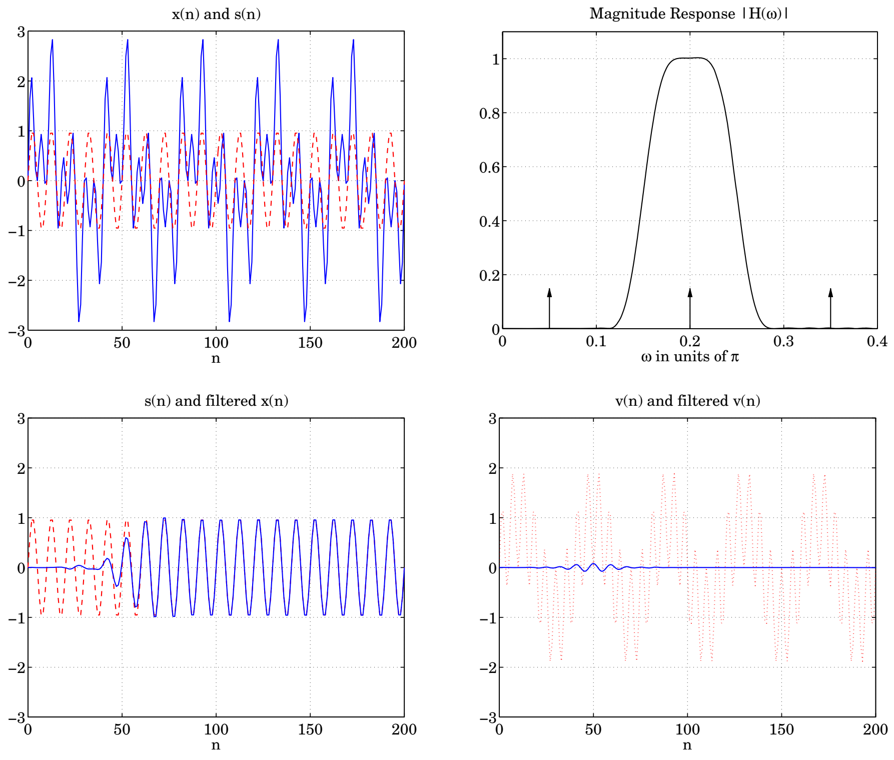
c. To see what happened to the interference, filter the signal $v(n)$ separately through the filter and plot the output, on the same graph with $v(n)$ itself.

d. Using the built-in MATLAB function `freqz`, or the textbook function `dtft`, calculate and plot the magnitude response of the filter over the frequency interval $0 \le \omega \le 0.4\pi$:

$$|H(\omega)| = \left|\sum_{n=0}^{M} h(n)e^{-j\omega n}\right|$$

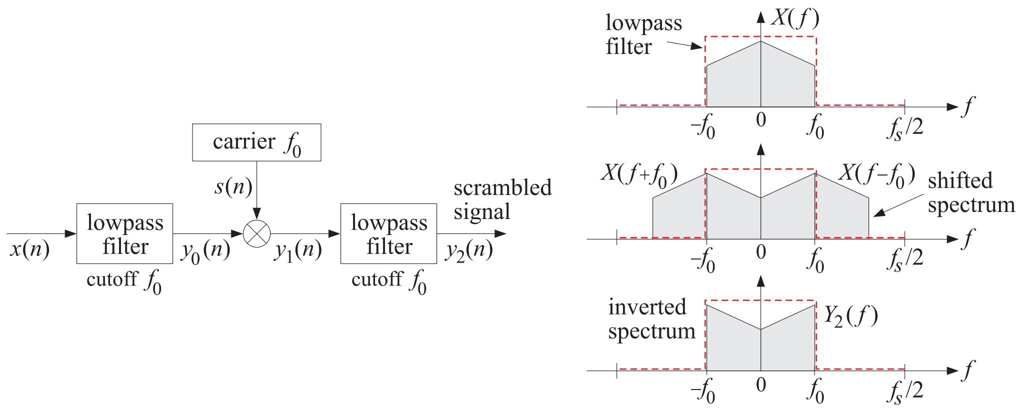
Indicate on that graph the frequencies $\omega_1, \omega_2, \omega_3$.

e. Redesign the filter with $M = 200$ and repeat parts (a)–(d). Discuss the effect of choosing a longer filter length.



3.4. Voice Scrambler in MATLAB

A simple voice scrambler works by spectrum inversion. It is not the most secure way of encrypting speech, but we consider it in this lab as an application of low pass filtering and AM modulation. The main operations are depicted below.



First, the sampled speech signal $x(n)$ is filtered by a lowpass filter $h(n)$ whose cutoff frequency f_0 is high enough not to cause distortions of the speech signal (the figure depicts an ideal filter). The sampling rate f_s is chosen such that $4f_0 < f_s$. The filtering operation can be represented by the convolutional equation:

$$y_0(n) = \sum_m h(m)x(n-m) \tag{3.2}$$

Next, the filter output $y_0(n)$ modulates a cosinusoidal carrier signal whose frequency coincides with the filter's cutoff frequency f_0, resulting in the signal:

$$y_1(n) = s(n)y_0(n), \quad \text{where} \quad s(n) = 2\cos(\omega_0 n), \quad \omega_0 = \frac{2\pi f_0}{f_s} \tag{3.3}$$

The multiplication by the carrier signal causes the spectrum of the signal to be shifted and centered at $\pm f_0$, as shown above. Finally, the modulated signal $y_1(n)$ is passed through the same filter again which removes the spectral components with $|f| > f_0$, resulting in a signal $y_2(n)$ with inverted spectrum. The last filtering operation is:

$$y_2(n) = \sum_m h(m)y_1(n-m) \tag{3.4}$$

To unscramble the signal, one may apply the scrambling steps (3.2)–(3.4) to the scrambled signal itself. This works because the inverted spectrum will be inverted again, recovering in the original spectrum.

Lab Procedure

a. Explain why the factor 2 is needed in the carrier definition $s(n) = 2\cos(\omega_0 n)$. Explain why f_0 must be chosen such that $4f_0 < f_s$ in designing the lowpass filter.

b. A practical FIR lowpass filter with cutoff frequency f_0 can be designed by a similar Hamming windowing method as that used for the bandpass filter of the previous section:

$$h(n) = w(n)\,\frac{\sin\big(\omega_0(n-M/2)\big)}{\pi(n-M/2)}, \quad 0 \le n \le M$$

where $\omega_0 = 2\pi f_0/f_s$, and $w(n)$ is the Hamming window:

$$w(n) = 0.54 - 0.46\cos\left(\frac{2\pi n}{M}\right), \quad 0 \le n \le M$$

Design such a filter using the following values: $f_s = 16$ kHz, $f_0 = 3.3$ kHz, and filter order $M = 100$. Then, using the built-in MATLAB function `freqz`, or the textbook function `dtft`, calculate and plot the magnitude response of the filter over the frequency interval $-f_s/2 \le f \le f_s/2$:

$$|H(f)| = \left| \sum_{n=0}^{M} h(n) e^{-2\pi j f n/f_s} \right|$$

c. Repeat part (b) with $M = 200$.

d. Write a MATLAB function named `scrambler.m` that implements the operations of Eqs. (3.2)–(3.4). It must have usage:

```
y = scrambler(h,w0,x);
```

where `h,w0,x` are the filter vector, carrier frequency in rads/sample, and input speech vector, and `y` is the scrambled output speech representing the signal $y_2(n)$.

Internally, the required filtering operations must be implemented with the built-in function `filter`, and the multiplication operation of Eq. (3.3) must be vectorized.

e. Write a MATLAB program that does the following: (a) designs the lowpass filter $h(n)$ with parameters f_s, f_0, M, (b) reads a vector $x(n)$ of input samples from a wavefile using the function `wavread`, (c) sends it through the function `scrambler`, (d) plays the output through the sound card using the function `sound`, (e) then, unscrambles the scrambled signal and plays it back using `sound`.

Apply your program to some wave files using the parameter values of part (b). To get an idea of what sort of output to expect for this part, we have included the following files in this lab:

```
JB.wav
JBs.wav
scramblex.p
```

The second file is the scrambled version of the first. Play them in your computer to hear what they sound like. Next, run the p-coded scramble example file, `scramblex`, in MATLAB and enter the first filename and you will hear its scrambled version. Then, run it again, and enter the second filname and you will hear its scrambled version, which is the original unscrambled one. (Note that a p-coded file is an encrypted m-file and can be run just like an m-file.)

3.5. 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.

Lab 4 – Delays and FIR Filtering

4.1. Introduction

In the last lab you studied filtering by convolution, which is a block processing method. In this lab you will study sample by sample processing methods for FIR filters and implement them on the TMS320C6713 processor. Once you know how to implement a multiple delay on a sample by sample basis, it becomes straightforward to implement FIR and IIR filters. Multiple delays are also the key component in various digital audio effects, such as reverb.

Delays can be implemented using linear or circular buffers, the latter being more efficient, especially for audio effects. The theory behind this lab is developed in Ch. 4 of the text [1] for FIR filters, and used in Ch. 8 for audio effects.

4.2. Delays Using Linear and Circular Buffers

A D-fold delay, also referred to as a delay line, has transfer function $H(z)= z^{-D}$ and corresponds to a time delay in seconds:

$$T_D = DT = \frac{D}{f_s} \quad \Rightarrow \quad D = f_s T_D \tag{4.1}$$

where T is the time interval between samples, related to the sampling rate by $f_s = 1/T$. A block diagram realization of the multiple delay is shown below:

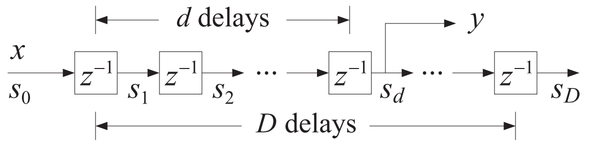


Fig. 4.1 Tapped delay line.

There are D registers whose contents are the "internal" states of the delay line. The dth state s_d, i.e., the content of the dth register, represents the d-fold delayed version of the input, that is, at time n we have: $s_d(n)= x(n-d)$, for $d = 1,\dots,D$; the case $d = 0$ corresponds to the input $s_0(n)= x(n)$.

At each time instant, all D contents are available for processing and can be "tapped" out for further use (e.g., to implement FIR filters). For example, in the above diagram, the dth tap is being tapped, and the corresponding transfer function from the input x to the output $y = s_d$ is the partial delay z^{-d}.

The D contents/states s_d, $d = 1,2,\dots,D$, and the input $s_0 = x$ must be stored in memory in a $(D+1)$-dimensional array or buffer. But the manner in which they are stored and retrieved depends on whether a linear or a circular buffer is used. The two cases are depicted below.

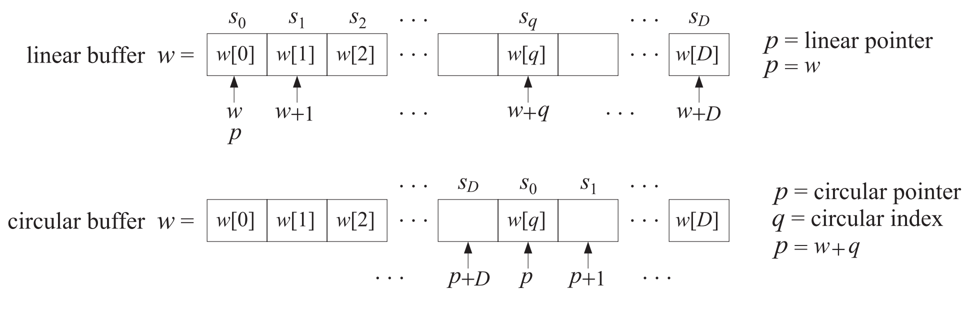


Fig. 4.2 Linear and circular buffers.

In both cases, the buffer can be created in C by the declaration:

```
float w[D+1];
```

Its contents are retrieved as $w[i]$, $i = 0, 1, \dots, D$. Thinking of w as pointer, the contents can also be retrieved by $*(w + i) = w[i]$, where $*$ denotes the de-referencing operator in C.

In the linear buffer case, the states are stored in the buffer sequentially, or linearly, that is, the ith state is:

$$s_i = w[i] = *(w + i)\,, \quad i = 0, 1, \dots, D$$

At each time instant, after the contents s_i are used, the delay-line is updated in preparation for the next time instant by shifting its contents to the right from one register to the next, as suggested by the block diagram in Fig. 4.1. This follows from the definition $s_i(n) = x(n - i)$, which implies for the next time instant $s_i(n+1) = x(n+1-i) = s_{i-1}(n)$. Thus, the current s_{i-1} becomes the next s_i. Since $s_i = w[i]$, this leads to the following updating algorithm for the buffer contents:

```
for $i = D$ down to $i = 1$, do:
    $w[i] = w[i-1]$
```

where the shifting is done from the right to the left to prevent the over-writing of the correct contents. It is implemented by the C function `delay()` of the text [1]:

```
// delay.c - linear buffer updating
// --------------------------------

void delay(int D, float *w)
{
   int i;

   for (i=D; i>=1; i--)
      w[i] = w[i-1];
}

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

For large values of D, this becomes an inefficient operation because it involves the shifting of large amounts of data data from one memory location to the next. An alternative approach is to keep the data unshifted but to shift the beginning address of the buffer to the left by one slot.

This leads to the concept of a circular buffer in which a movable pointer p is introduced that always points somewhere within the buffer array, and its current position allows one to retrieve the states by $s_i = *(p + i)$, $i = 0, 1, \dots, D$. If the pointer $p + i$ exceeds the bounds of the array to the right, it gets wrapped around to the beginning of the buffer.

To update the delay line to the next time instant, the pointer is left-shifted, i.e., by the substitution $p = p - 1$, or, $--p$, and is wrapped to the right if it exceeds the array bounds to the left. Fig. 4.3 depicts the contents and pointer positions at two successive time instants for the linear and circular buffer cases for $D = 3$. In both cases, the states are retrieved by $s_i = *(p + i)$, $i = 0, 1, 2, 3$, but in the linear case, the pointer remains frozen at the beginning of the buffer, i.e., $p = w$, and the buffer contents shift forwards, whereas in the circular case, p shifts backwards, but the contents remain in place.

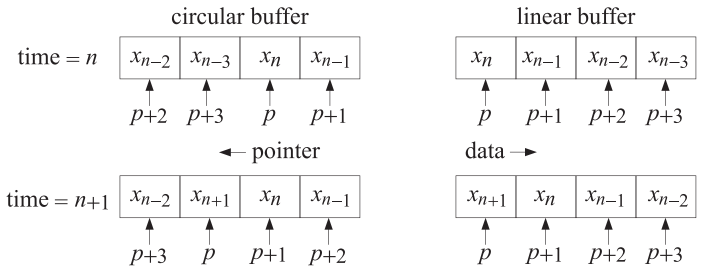


Fig. 4.3 Buffer contents at successive time instants for $D = 3$.

In the text [1], the functions `tap()` and `cdelay()` are used for extracting the states s_i and for the circular back-shifting of the pointer. Although these two functions could be used in the CCS environment, we prefer instead to use a single function called `pwrap()` that calculates the new pointer after performing the required wrapping. The function is declared in the common header file `dsplab.h` and defined in the file `dsplab.c`. Its listing is as follows:

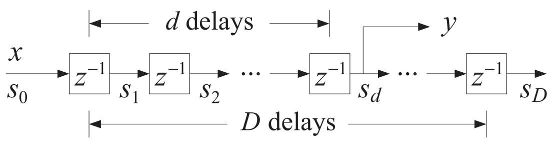
```
// pwrap.c - pointer wrapping relative to circular buffer
// Usage: p_new = pwrap(D,w,p)
// -----------------------------------------------------

float *pwrap(int D, float *w, float *p)
{
   if (p > w+D)
      p -= D+1;

   if (p < w)
      p += D+1;

   return p;
}

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

The ith state s_i and the updating of the delay-line can be obtained by the function calls:

$$s_i = *\text{pwrap}(D, w, p + i)\,, \quad i = 1, 2, \dots, D$$

$$p_{\text{next}} = \text{pwrap}(D, w, --p)$$

We will use this function in the implementation of FIR filters and in various audio effects. It will allow us to easily translate a sample processing algorithm expressed in pseudo-code to the actual C code. As an example, let us consider the circular buffer implementation of the partial delay z^{-d}. The block diagram of Fig. 4.1 and the pseudo-code computational algorithm are as follows:



```
for each input x do:
    $y = s_d = *(p + d)$
    $*p = x$
    $p--$
```

We may translate this into C by the following operations using `pwrap`:

```
y = *pwrap(D,w,p+d);          // delay output
*p = x;                       // delay-line input
p = pwrap(D,w,--p);           // backshift circular buffer pointer
```

In the last line, we must pre-decrement the pointer, that is, `--p`, instead of post-decrementing it, `p--`. By comparison, the linear buffer implementation, using a $(D+1)$-dimensional buffer, is as follows:

```
y = w[d];                      // delay output
w[0] = x;                      // delay-line input
for (i=D; i>=0; i--)           // update linear buffer
   w[i] = w[i-1];
```

An alternative approach to circular buffers is working with circular indices instead of circular pointers. The pointer p always points to some element of the buffer array w, that is, there is a unique integer q such that $p = w + q$, with corresponding content $*p = w[q]$. This is depicted in Fig. 4.2. The index q is always bound by the limits $0 \le q \le D$ and wrapped modulo-$(D+1)$ if it exceeds these limits.

The textbook functions `tap2()` and `cdelay2()`, and their corresponding MATLAB versions given in the Appendix of [1], implement this approach. Again, however, we prefer to use the following function, `qwrap()`, also included in the common file `dsplab.c`, that calculates the required wrapped value of the circular index:

```
// qwrap.c - circular index wrapping
// Usage: q_new = qwrap(D,q);
// -------------------------------------

int qwrap(int D, int q)
{
       if (q > D)
              q -= D + 1;

       if (q < 0)
              q += D + 1;

       return q;
}

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

In terms of this function, the above d-fold delay example is implemented as follows:

```
y = w[qwrap(D,q+d)];           // delayed output
w[q] = x;                      // delay-line input
q = qwrap(D,--q);              // backshift pointer index
```

We note that in general, the ith state is:

$$s_i = *(p+i) = *(w+q+i) = w[q+i]$$

where $q+i$ must be wrapped as necessary. Thus, the precise way to extract the ith state is:

$$s_i = w\big[\mathrm{qwrap}(D, q+i)\big], \quad i = 1, 2, \dots, D$$

Lab Procedure

A complete C program that implements the above d-fold delay example on the TMS320C6713 processor is given below:

```
// delay1.c - multiple delay example using circular buffer pointers (pwrap version)
// ----------------------------------------------------------------------------------

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

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

#define D 8000               // max delay in samples (TD = D/fs = 8000/8000 = 1 sec)
short fs = 8;                // sampling rate in kHz
float w[D+1], *p, x, y;      // circular delay-line buffer, circular pointer, input, output
int d = 4000;                // must be d <= D

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

void main()                  // main program executed first
{
   int n;

   for (n=0; n<=D; n++)      // initialize circular buffer to zero
      w[n] = 0;
   p = w;                    // initialize pointer

  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);         // use 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
{
   read_inputs(&xL, &xR);    // read left and right input samples from codec

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

   y = *pwrap(D,w,p+d);      // delayed output - pwrap defined in dsplab.c
   *p = x;                   // delay-line input
   p = pwrap(D,w,--p);       // backshift pointer

   yL = yR = (short) y;

   write_outputs(yL,yR);     // write left and right output samples to codec

   return;
}

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

Note the following features. The sampling rate is set to 8 kHz, therefore, the maximum delay $D = 8000$ corresponds to a delay of 1 sec, and the partial delay $d = 4000$, to $1/2$ sec. The circular buffer array `w` has dimension $D + 1 = 8001$ and its scope is global within this file. It is initialized to zero within `main()` and the pointer p is initialized to point to the beginning of w, that is, $p = w$.

The left/right input samples, which are of the **short int** type, are cast to **float**, while the **float** output is cast to **short int** before it is sent out to the codec.

a. Create and build a project for this program. Then, run it. Give the system an impulse by lightly tapping the table with the mike, and listen to the impulse response. Then, speak into the mike.

   Bring the mike near the speaker and then give the system an impulse. You should hear repeated echoes. If you bring the mike too close to the speakers the output goes unstable. Draw a block diagram realization that would explain the effect you are hearing. Experimentally determine the distance at which the echoes remain marginally stable, that is, neither die out nor diverge. (Technically speaking, the poles of your closed-loop system lie on the unit circle.)

b. Change the sampling rate to 16 kHz, recompile and reload keeping the value of d the same, that is, $d = 4000$. Listen to the impulse response. What is the duration of the delay in seconds now?

c. Reset the sampling rate back to 8 kHz, and this time change d to its maximum value $d = D = 8000$. Recompile, reload, and listen to the impulse response. Experiment with lower and lower values of d and listen to your delayed voice until you can no longer distinguish a separate echo. How many milliseconds of delay does this correspond to?

d. Set $d = 0$, recompile and reload. This should correspond to no delay at all. But what do you hear? Can you explain why? Can you fix it by changing the program? Will your modified program still work with $d \neq 0$? Is there any good reason for structuring the program the way it was originally?

e. In this part you will profile the computational cost of the sample processing algorithm. Open the source file `delay1.c` in a CCS window. Locate the `read_inputs` line in the `isr()`, then right-click on that line and choose *Toggle Software Breakpoint*; a red dot will appear in the margin. Do the same for the `write_outputs` line. From the top menu of the CCS window, choose *Profile -> Clock -> View*; a little yellow clock will appear on the right bottom status line of CCS. When you compile, load, and run your program, it will stop at the first breakpoint, with a yellow arrow pointing to it. Double-click on the profile clock to clear the number of cycles, then type `F5` to continue running the program and it will stop at the second breakpoint. Read and record the number of cycles shown next to the profile clock.

f. Write a new program, called `delay2.c`, that makes use of the function `qwrap` instead of `pwrap`. Repeat parts (a) and (e).

g. Next, write a new program, called `delay3.c`, that uses linear buffers. Its `isr()` will be as follows:

```
interrupt void isr()
{
   int i;

   read_inputs(&xL, &xR);

   x = (float) xL;

   w[0] = x;                    // delay-line input
   y = w[d];                    // delay output
   for (i=D; i>=0; i--)         // update linear buffer
      w[i] = w[i-1];

   yL = yR = (short) y;

   write_outputs(yL,yR);

   return;
}
```

Build the project. You will find that it may not run (because the data shifts require too many cycles that over-run the sampling rate). Change the program parameters D, d to the following values $D = 2000$ and $d = 1000$. Rebuild and run the program. Repeat part (e) and record the number of cycles. Change the parameters D, d of the program `delay1.c` to the same values, and repeat part (e) for that. Comment on the required number of samples using the linear vs. the circular buffer implementation.

4.3. FIR Comb Filters Using Circular Buffers

More interesting audio effects can be derived by combining several multiple delays. An example is the FIR comb filter defined by Eq. (8.2.8) of the text [1]:

$$y_n = x_n + a\,x_{n-D} + a^2 x_{n-2D} + a^3 x_{n-3D}$$

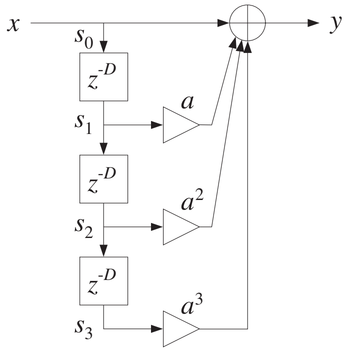
Its transfer function is given by Eq. (8.2.9):

$$H(z) = 1 + az^{-D} + a^2 z^{-2D} + a^3 z^{-3D}$$

Its impulse response has a very sparse structure:

$$\mathbf{h} = [1, \underbrace{0, 0, \dots, 0}_{D-1 \text{ zeros}}, a, \underbrace{0, 0, \dots, 0}_{D-1 \text{ zeros}}, a^2, \underbrace{0, 0, \dots, 0}_{D-1 \text{ zeros}}, a^3]$$

The comb-like structure of its frequency response and its zero-pattern on the z-plane are depicted in Fig. 8.2.5 of [1]. Instead of implementing it as a general FIR filter, a more efficient approach is to program the block diagram directly by using a single delay line of order $3D$ and tapping it out at taps 0, D, $2D$, and $3D$. The block diagram realization and corresponding sample processing algorithm are:



```
for each input x do:
    $s_0 = x$
    $s_1 = *(p + D)$
    $s_2 = *(p + 2D)$
    $s_3 = *(p + 3D)$
    $y = s_0 + as_1 + a^2 s_2 + a^3 s_3$
    $*p = s_0$
    $--p$
```

The translation of the sample processing algorithm into C is straightforward and can be incorporated into the following `isr()` function to be included in your main program:

```
interrupt void isr()
{
   float s0, s1, s2, s3, y;              // states & output

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

   s0 = (float) xL;                      // work with left input only
   s1 = *pwrap(3*D,w,p+D);               // extract states relative to p
   s2 = *pwrap(3*D,w,p+2*D);             // note, buffer length is 3D+1
   s3 = *pwrap(3*D,w,p+3*D);
   y = s0 + a*s1 + a*a*s2 + a*a*a*s3;    // output sample
   *p = s0;                              // delay-line input
   p = pwrap(3*D,w,--p);                 // backshift pointer

   yL = yR = (short) y;

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

   return;
}
```

Lab Procedure

Set the sampling rate to 8 kHz and the audio source to microphone. Choose the delay to be $D = 4000$, corresponding to $T_D = 0.5$ sec, so that the total duration of the filter is $3T_D = 1.5$ sec, and set $a = 0.5$.

a. Write a C program called `comb.c` that incorporates the above interrupt service routine. You will need to globally declare/define the parameters D, a, p, as well as the circular buffer w to be a $3D+1$ dimensional float array. Make sure you initialize the buffer to zero inside `main()`, as was done in the previous example, and also initialize $p = w$.

   Build and run this project. Listen to the impulse response of the filter by tapping the table with the mike. Speak into the mike. Bring the mike close to the speakers and get a closed-loop feedback.

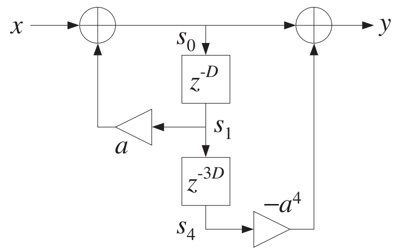
b. Keeping the delay D the same, choose $a = 0.2$ and run the program again. What effect do you hear? Repeat for $a = 0.1$. Repeat with $a = 1$.

c. Set the audio input to LINE and play your favorite wave or MP3 song into the input. Experiment with reducing the value of D in order to match your song's tempo to the repeated echoes. Some wave files are in the directory `c:\dsplab\wav` (e.g., try `dsummer`, `take5`.)

d. The FIR comb can also be implemented *recursively* using the geometric series formula to rewrite its transfer function in the recursive form as shown in Eq. (8.2.9) of the text:

$$H(z) = 1 + az^{-D} + a^2z^{-2D} + a^3z^{-3D} = \frac{1 - a^4z^{-4D}}{1 - az^{-D}}$$

   This requires a $(4D+1)$-dimensional delay-line buffer w. The canonical realization and the corresponding sample processing algorithm are shown below:



```
for each input x do:
    $s_1 = *(p + D)$
    $s_4 = *(p + 4D)$
    $s_0 = x + as_1$
    $y = s_0 - a^4 s_4$
    $*p = s_0$
    $--p$
```

Write a new program, `comb2.c`, that implements this algorithm. Remember to define the buffer to be a $(4D+1)$-dimensional float array. Using the values $D = 1600$ (corresponding to a 0.2 sec delay) and $a = 0.5$, recompile and run both the `comb.c` and `comb2.c` programs and listen to their outputs.

In general, such recursive implementations of FIR filters are more prone to the accumulation of round-off errors than the non-recursive versions. You may want to run these programs with $a = 1$ to observe this sensitivity.

4.4. FIR Filters with Linear and Circular Buffers

The sample-by-sample processing implementation of FIR filters is discussed in Sect. 4.2 of the text [1]. For an order-M filter, the input/output convolutional equation can be written as the dot product of the filter-coefficient vector $\mathbf{h} = [h_0, h_1, \dots, h_M]^T$ with the state vector $\mathbf{s}(n) = [x_n, x_{n-1}, \dots, x_{n-M}]^T$:

$$y_n = \sum_{m=0}^{M} h(m)x(n-m) = [h_0, h_1, \dots, h_M]\begin{bmatrix} x(n) \\ x(n-1) \\ \vdots \\ x(n-M) \end{bmatrix} = \mathbf{h}^T\mathbf{s}(n)\,, \quad \mathbf{s}(n) = \begin{bmatrix} x(n) \\ x(n-1) \\ \vdots \\ x(n-M) \end{bmatrix}$$

We note that the ith component of the state vector is $s_i(n) = x(n-i)$, $i = 0, 1, \dots, M$, and therefore, the states are the tap outputs of a multiple delay-line with M delays. Thus, the definition of the delay line and its time updating remains the same as in the previous sections. To realize the FIR filter, we must use the tapped outputs s_i from the delay line to calculate the dot product, and then update the delay line to the next time instant. The following C function from the text [1] implements the linear buffer case:

```
// fir.c - FIR filter in direct form with linear buffer
// Usage: y = fir(M, h, w, x);
// ----------------------------------------------------------------

float fir(int M, float *h, float *w, float x)
{
   int i;
   float y;                        // y=output sample

   w[0] = x;                       // read current input sample x

   for (y=0, i=0; i<=M; i++)       // process current output sample
       y += h[i] * w[i];           // dot-product operation

   for (i=M; i>=1; i--)            // update states for next call
      w[i] = w[i-1];               // done in reverse order

   return y;
}

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

The circular-buffer implementation can be carried out by the textbook function `cfir()` that uses a circular pointer, or, by the function `cfir2()` that uses a circular index. In this lab. we have replaced `cfir()` by the following function `firc()`, which makes use of the pointer-wrapping function `pwrap()`:

```
// firc.c - FIR filter implemented with circular pointer
// Usage: y = firc(M, h, w, &p, x);
// ----------------------------------------------------------------------

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

float firc(int M, float *h, float *w, float **p, float x)
{
   int i;
   float y;
   **p = x;                         // read input sample x

   for (y=0, i=0; i<=M; i++) {
      y += (*h++) * (**p);          // i-th state s[i] = *pwrap(M,w,*p+i)
      *p = pwrap(M,w,++*p);         // pointer to state s[i+1] = *pwrap(M,w,*p+i+1)
      }

   *p = pwrap(M,w,--*p);            // update circular delay line

   return y;
}

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

Because the pointer p changes at each call, it is passed by address, and therefore, it is declared as a pointer to pointer in the function. Similarly, we replaced `cfir2()` by `firc2()`, which uses `qwrap()`:

```
// firc2.c - FIR filter implemented with circular index
// Usage: y = firc2(M, h, w, &q, x);
// ---------------------------------------------------------------------

int qwrap(int, int);                                      // defined in dsplab.c

float firc2(int M, float *h, float *w, int *q, float x)
{
   int i;
   float y;

   w[*q] = x;                        // read input sample x

   for (y=0, i=0; i<=M; i++) {
      y += *h++ * w[*q];             // i-th state s[i] = w[*q]
      *q = qwrap(M,++*q);            // pointer to state s[i+1] = w[*q+1]
      }

   *q = qwrap(M,--*q);               // update circular delay line

   return y;
}

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

Lab Procedure

As discussed in the voice scrambler example of Lab-3, a lowpass FIR filter of order M and cutoff frequency f_0 can be designed using the Hamming window approach by the following equations:

$$w(n) = 0.54 - 0.46\cos\left(\frac{2\pi n}{M}\right), \quad h(n) = w(n)\,\frac{\sin\big(\omega_0(n - M/2)\big)}{\pi(n - M/2)}, \quad 0 \le n \le M$$

where $\omega_0 = 2\pi f_0/f_s$, and $w(n)$ is the Hamming window.

a. Design such a filter with MATLAB using the following values: $f_s = 8$ kHz, $f_0 = 2$ kHz, and filter order $M = 50$. Then, using the built-in MATLAB function `freqz`, or the textbook function `dtft`, calculate and plot in dB the magnitude response of the filter over the frequency interval $0 \le f \le 4$ kHz.

b. The designed 51-long impulse response coefficient vector **h** can be exported into a data file, `h.dat`, in a form that is readable by a C program by the following MATLAB command:

```
C_header('h.dat', 'h', 'M', h);
```

where `C_header` is a MATLAB function in the directory `c:\dsplab\common`. A few lines of the resulting data file are shown below:

```
// h.dat  -  FIR impulse response coefficients
// exported from MATLAB using C_header.m
// ------------------------------------------

#define M 50        // filter order

float h[M+1] = {
    0.001018591635788,
   -0.000000000000000,
   -0.001307170629617,
    0.000000000000000,
    0.002075061044252,
   -0.000000000000000,
      --- etc. ---
   -0.001307170629617,
   -0.000000000000000,
    0.001018591635788
   };

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

The following complete C program called `firex.c` implements this example on the C6713 processor. The program reads the impulse response vector from the data file `h.dat`, and defines a linear 51-dimensional buffer array. The FIR filtering operation is based on the function `fir()`.

```
// firex.c - FIR filtering example
// ------------------------------------------------------------------------------------

#include "dsplab.h"          // DSK initialization declarations and function prototypes

float fir(int, float *, float *, float);
// float firc(int, float *, float *, float **, float);
// float firc2(int, float *, float *, int *, float);

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

#include "h.dat"             // defines M=50, and contains M+1 = 51 filter coefficients

float w[M+1];                // delay line buffer
int on = 1;                  // turn filter on or off
// float *p;
// int q;

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

void main()
{
  int i;

  for (i=0; i<=M; i++) w[i] = 0;
  // p=w;
  // q=0;

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

  sampling_rate(8);          // 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()
{
  float x, y;                 // filter input & output
```

```
   read_inputs(&xL, &xR);

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

      y = fir(M,h,w,x);
      // y = firc(M,h,w,&p,x);
      // y = firc2(M, h, w, &q, x);

      yL = (short)(y);
      }
   else
      yL = xL;

  write_outputs(yL,yL);

  return;
}

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

Create and build a project for this program. You will need to add the file `fir.c` to the project. Using the following MATLAB code (same as in the aliasing example of Lab-2), generate a signal consisting of a 1-kHz segment, followed by a 3-kHz segment, followed by another 1-kHz segment, where all segments have duration of 1 sec:

```
fs = 8000; f1 = 1000; f2 = 3000; f3 = 1000;
L = 8000; n = (0:L-1);
A = 1/5;                         % adjust playback volume

x1 = A * cos(2*pi*n*f1/fs);
x2 = A * cos(2*pi*n*f2/fs);
x3 = A * cos(2*pi*n*f3/fs);

sound([x1,x2,x3], fs);
```

First, set the parameter `on=0` so that the filtering operation is bypassed. Send the above signal into the line input of the DSK and listen to the output. Then, set `on=1` to turn the filter on, recompile and run the program, and send the same signal in. The middle 3-kHz segment should not be heard, since it lies in the filter's stopband.

c. Create breakpoints at the `read_inputs` and `write_outputs` lines of the `isr()` function, and start the profile clock. Run the program and record the number of cycles between reading the input samples and writing the computed outputs.

d. Uncomment the appropriate lines in the above program to implement the circular buffer version using the function `firc()`. You will need to add it to your project. Recompile and run your program with the same input.

   Then, repeat part (c) and record the number of computation cycles.

e. Repeat part (d) using the circular-index function `firc2()`.

In comparing the computational costs of the various implementations, you will notice that for this example, the linear buffer version is far more efficient, in contrast to what you observed in the case of multiple delays. The reason is that the function `pwrap()` gets called for each tap of the FIR filter, whereas in the case of multiple delays, it was essentially called once.

The circular buffer implementation of FIR filters can indeed be made far more efficient than the linear buffer case if one used an assembly language version that takes advantage of the built-in circular addressing capability of the C6713 processor.

4.5. Voice Scrambler in C

In this lab, you will study the real-time implementation of the voice scrambler that you designed in Lab-3. We recall that the lowpass filter was designed with the parameters $f_s = 16$ kHz, $f_0 = 3.3$ kHz, and filter order $M = 100$ using the Hamming design method:

$$h(n) = w(n)\,\frac{\sin\big(\omega_0(n-M/2)\big)}{\pi(n-M/2)}\,, \quad 0 \le n \le M \tag{4.2}$$

where $\omega_0 = 2\pi f_0/f_s$, and $w(n)$ is the Hamming window:

$$w(n) = 0.54 - 0.46\cos\left(\frac{2\pi n}{M}\right), \quad 0 \le n \le M \tag{4.3}$$

The following C program, `scrambler.c`, forms the basis of this lab. It is a variation of that discussed in the Chassaing-Reay text [2].

```
// scrambler.c - voice scrambler example
// ------------------------------------------------------------------------------------------

#include "dsplab.h"                  // initialization declarations and function prototypes
#include <math.h>
#define PI 3.14159265                // 3.14159265358979

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

#define M 100                        // filter order
#define L 160                        // carrier period, note L*f0/fs = 160*3.3/16 = 33 cycles

float h[M+1], w1[M+1], w2[M+1];      // filter coefficients and delay-line buffers
int n=0;                             // time index for carrier, repeats with period L
int on=1;                            // turn scrambler on (off with on=0)

float w0, f0 = 3.3;                  // f0 = 3.3 kHz
short fs = 16;                       // fs = 16 kHz

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

void main()
{
  int i;
  float wind;

  w0 = 2*PI*f0/fs;                               // carrier frequency in rads/sample

  for (i=0; i<=M; i++)  {                        // initialize buffers & design filter
      w1[i] = w2[i] = 0;
      wind = 0.54 - 0.46 * cos(2*PI*i/M);        // Hamming window
      if (i==M/2)
         h[i] = w0/PI;
      else
         h[i] = wind * sin(w0*(i-M/2)) / (PI*(i-M/2));
      }

  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
{
   int i;
   float y;

   read_inputs(&xL, &xR);      // read left and right input samples from codec

   if (on) {
      y = (float)(xL);                 // work with left input only

      w1[0] = y;                       // first filter
      for (y=0, i=0; i<=M; i++)
         y += h[i] * w1[i];
      delay(M,w1);

      y *= 2*cos(w0*n);                // multiply y by carrier
      if (++n >= L) n = 0;



      w2[0] = y;                       // second filter
      for (y=0, i=0; i<=M; i++)
         y += h[i] * w2[i];
      delay(M,w2);

      yL = (short)(y);
      }
   else
      yL = xL;                         // pass through if on=0

  write_outputs(yL,yL);

  return;
}

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

Two separate buffers, w_1, w_2, are used for the two lowpass filters. The filter coefficients are computed on the fly within `main()` using Eqs. (4.2) and (4.3). A linear buffer implementation is used for both filters. The sinusoidal carrier signal is defined by:

$$s[n]= 2\cos(\omega_0 n)\,, \quad \omega_0 = \frac{2\pi f_0}{f_s}$$

Since $f_s/f_0 = 16/3.3$ samples/cycle, it follows that the smallest number of samples containing an integral number of cycles will be:

$$L = \frac{16}{3.3}\cdot 33 = 160 \text{ samples}$$

that is, these 160 samples contain 33 cycles and will keep repeating. Therefore, the time index n of $s[n]$ is periodically cycled over the interval $0 \le n \le L-1$.

Lab Procedure

a. Create and build a project for this program. The parameter `on=1` turns the scrambler on or off. Create a GEL file for this parameter and open it when you run the program.

b. Play the following two wave files through program:

```
JB.wav
JBs.wav
```

When you play the second, which is a scrambled version of the first, it will get unscrambled.

c. Open MATLAB and generate three sinusoids of frequencies 300 Hz, 3000 Hz, and 300 Hz, sampled at a rate of 16 kHz, each of duration of 1 second, and concatenate them to form a 3-second signal. Then play this out of the PCs sound card using the `sound()` function. For example, the following MATLAB code will do this:

```
fs = 16000; f1 = 300; f2 = 3000; f3 = 300;
L = 16000; n = (0:L-1);
A = 1/5;                          % adjust playback volume

x1 = A * cos(2*pi*n*f1/fs);
x2 = A * cos(2*pi*n*f2/fs);
x3 = A * cos(2*pi*n*f3/fs);

sound([x1,x2,x3], fs);
```

Play this signal through the DSK with the scrambler off. Then, play it with the scrambler on. What are the frequencies in Hz of the scrambled signal that you hear? Explain this in your report.

d. Instead of actually computing the cosine function at each call of `isr()`, a more efficient approach would be to pre-compute the L repeating samples of the carrier $s[n]$ and keep re-using them. This can be accomplished by replacing the two modulation instructions in `isr()` by:

```
y *= s[n];              // multiply y by carrier
if (++n >= L) n = 0;
```

where $s[n]$ must be initialized within `main()` to the L values, $s[n] = 2\cos(\omega_0 n)$, $n = 0, 1, \dots, L-1$. Re-write the above program to take advantage of this suggestion. Test your program.

4.6. 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.

Lab 5 – DTFT/DFT/FFT

5.1. DTFT Examples

In this part, we consider the analytical computation of the DTFT of some periodic signals, the computation of their DFT, and their representation as sums of sinusoids.

Lab Procedure

a. Consider the periodic square-wave of period 10,

$$s(n) = [\underbrace{1,1,1,1,1,-1,-1,-1,-1,-1}_{\text{one period}}, \cdots]$$

where the dots represent the repetition of the basic period. Prove analytically that the DTFT of one basic period is given by

$$S(e^{j\omega}) = \sum_{n=0}^{9} s(n)e^{-j\omega n} = \frac{(1-e^{-5j\omega})^2}{1-e^{-j\omega}}$$

b. By evaluating this at the 10 DFT frequencies $\omega_k = 2\pi k/10$, $k = 0,1,\dots,9$, determine the 10-point DFT of one basic period, i.e., the values $S(k) = S(e^{j\omega_k})$. [*Hint:* $e^{j\pi} = -1$]

c. By inserting the result of part (b) into the inverse DFT formula,

$$x(n) = \frac{1}{N}\sum_{k=0}^{N-1} X(k)e^{j\omega_k n}$$

prove analytically that the above square wave admits the following expansion in terms of sinusoids:

$$s(n) = \frac{0.4}{\sin\left(\dfrac{\omega_1}{2}\right)}\sin\left(\omega_1 n + \frac{\omega_1}{2}\right) + \frac{0.4}{\sin\left(\dfrac{\omega_3}{2}\right)}\sin\left(\omega_3 n + \frac{\omega_3}{2}\right) + 0.2\cos(\omega_5 n)$$

and verify that this expression agrees with the given square-wave for $n = 0,1,\dots,9$.

d. Repeat parts (a–c) for the following square-wave:

$$s(n) = [\underbrace{0,1,1,1,1,0,-1,-1,-1,-1}_{\text{one period}}, \cdots]$$

and show that in this case,

$$S(e^{j\omega}) = \sum_{n=0}^{9} s(n)e^{-j\omega n} = \frac{(e^{-j\omega}-e^{-5j\omega})(1-e^{-5j\omega})}{1-e^{-j\omega}}$$

and

$$s(n) = \frac{0.4}{\tan\left(\dfrac{\omega_1}{2}\right)}\sin(\omega_1 n) + \frac{0.4}{\tan\left(\dfrac{\omega_3}{2}\right)}\sin(\omega_3 n)$$

e. For both square waves of parts (a) and (d), make a plot of their DTFT magnitude, $|S(e^{j\omega})|$, versus digital frequency in the interval $0 \le \omega \le 2\pi$. Superimpose on the same graph the corresponding 10-point DFT values plotted with circle markers.

These particular periodic square wave signals will be used again in hardware Lab-8.

5.2. Filtering of Periodic Signals

If a causal sinusoidal signal of frequency ω is sent to the input of a (strictly stable) filter, the output will tend in the long run, after the transients die out, to a steady-state sinusoidal signal of the same frequency ω, but modified in amplitude by the frequency response of the filter:

$$x(n)=e^{j\omega n}u(n) \quad \Rightarrow \quad y(n)=H(e^{j\omega})e^{j\omega n}u(n)+y_{\text{trans}}(n)\longrightarrow H(e^{j\omega})e^{j\omega n}u(n)=\text{steady-state}$$

where $u(n)$ is the unit-step function. This is the most important property of linear systems. The sinusoid $e^{j\omega n}$ is not necessarily a periodic signal in the discrete-time variable n unless the frequency has the form $\omega = 2\pi k/N$, for integer values of k, N, and in this case, it is periodic with period N, that is, $e^{j\omega(n+N)} = e^{j\omega n}$. A more general periodic discrete-time signal $s(n)$ with period N is specified by its sample values of one period,

$$s(n)=[\underbrace{s_0, s_1, s_2\dots, s_{N-1}}_{\text{one period}}, \underbrace{s_0, s_1, s_2\dots, s_{N-1}}_{\text{one period}}, \cdots]$$

and it can be represented by the inverse DFT formula as a sum of sinusoids at the DFT frequencies $\omega_k = 2\pi k/N$, $k = 0, 1, \dots, N-1$,

$$s(n)=\frac{1}{N}\sum_{k=0}^{N-1}S(k)e^{j\omega_k n} \tag{5.1}$$

where $S(k)$ is the N-point DFT of one period of the periodic signal:

$$S(k)=\sum_{n=0}^{N-1}s(n)e^{-j\omega_k n}$$

If a periodic signal such as $s(n)$ is sent to the input of a stable filter, then after the filter transients die out, the steady-state output signal will also be periodic with the same period N, and given in its sinusoidal form by the inverse DFT:

$$s_{\text{out}}(n)=\frac{1}{N}\sum_{k=0}^{N-1}H(e^{j\omega_k})S(k)e^{j\omega_k n} \tag{5.2}$$

This follows by applying the sinusoidal response of the filter to the individual terms of Eq. (5.1). Thus, the following computational steps allow one to determine the samples of the output period:

$$\begin{array}{ll}
\text{compute FFT of input period:} & \mathbf{S} = \text{fft}(\mathbf{s}, N) \\
\text{evaluate filter at DFT frequencies:} & \mathbf{H} = \big[H(e^{j\omega_0}), H(e^{j\omega_1}), \dots, H(e^{j\omega_{N-1}})\big] \\
\text{point-by-point multiplication:} & \mathbf{S}_{\text{out}} = \mathbf{H}\,.\!*\,\mathbf{S} \\
\text{compute IFFT:} & \mathbf{s}_{\text{out}} = \text{ifft}(\mathbf{S}_{\text{out}}, N)
\end{array} \tag{5.3}$$

Lab Procedure

a. Write a MATLAB function that implements the steps in Eq. (5.3), with syntax:

```
s_out = periodic_output(b,a,s);
```

where `b,a` are the numerator and denominator coefficient vectors of the filter, and `s,s_out` represent one period of the input and output signals. All the operations inside this function must be vectorized. To help you debug your program, the following answer is given:

$$\mathbf{s} = [3, 6, 3], \quad H(z)=\frac{2+z^{-1}}{1+0.5z^{-3}} \quad \Rightarrow \quad \mathbf{s}_{\text{out}} = [6, 10, 8]$$

b. Apply your function to the period-8 square wave:

$$s(n) = [\underbrace{1, 1, 1, 1, -1, -1, -1, -1}_{\text{one period}}, \cdots] \tag{5.4}$$

which is sent into the filter

$$H(z) = \frac{1 + z^{-1} + z^{-2}}{1 + 0.5z^{-4}}$$

and compute the corresponding length-8 output period $\mathbf{s}_{\text{out}}$.

c. Repeat the calculation of $\mathbf{s}_{\text{out}}$ by performing the operations of Eq. (5.3) by hand using 8-point FFTs and showing all calculations explicitly.

d. Generate an input signal $x(n)$ from Eq. (5.4) consisting of 5 periods only and filter it through $H(z)$ using the function `filter`, i.e.,

```
y = filter(b,a,x);
```

On two separate graphs, make stem plots of the signals $x(n)$ and $y(n)$. Observe how the output signal converges to the computed output period as the transients die out.

e. For a filter $H(z) = B(z)/A(z)$, write a MATLAB function that estimates the effective time constant of the filter, say, n_{eff}, i.e., the number of time samples that elapse until the filter transients effectively die out and the output settles into its steady-state. The filter $H(z)$ is arbitrary, but must be assumed to be stable with all its poles strictly inside the unit circle. The function must have syntax:

```
n_eff = time_constant(b,a);
```

The function must also handle the case of an FIR filter, i.e., `a=[1]`.

Apply your function to the example of part (b) and verify that the chosen 5 periods were sufficiently long to let the transients die out.

5.3. Circular Convolution

The modulo-N circular convolution of two sequences $\mathbf{h}$, $\mathbf{x}$ is defined as the modulo-N wrapping of ordinary linear convolution. See Sect. 9.9.1 of the text [1] for more information on this definition.

Lab Procedure

a. Recent versions of MATLAB have a function called, `datawrap()`, that can perform such mod-N wrapping. Using `datawrap()` and the function `conv()` only, write a MATLAB function, say, `circonv()`, that implements circular convolution. It must have syntax:

```
y = circonv(h,x,N);
```

where the two vectors `h,x` may have arbitrary lengths, not necessarily equal to N.

b. Let $\mathbf{h} = [1, 2, -1, 1]$ and $\mathbf{x} = [1, 1, 2, 1, 2, 2, 1, 1]$. Using your function, calculate the mod-4 circular convolution of the two sequences. Repeat for the mod-8 case.

c. Repeat part (a) by performing all calculations by hand, i.e., ordinary convolution, followed by wrapping modulo-N.

d. Circular convolution is equivalent to the DFT/FFT operations, expressed in MATLAB notation:

```
y = ifft(fft(h,N).*fft(x,N),N);
```

where all FFTs have length N.

Carry out part (a) by performing the required 4-point or 8-point FFTs by hand.

[One word of caution, for the case $N = 4$, you must pre-wrap the length-8 signal $\mathbf{x}$ modulo-4 before doing the circular convolution, otherwise MATLAB will truncate $\mathbf{x}$ to the length-4 signal $[1, 1, 2, 1]$ in performing its 4-point FFT.]

5.4. Spectral Analysis by DFT/FFT

The purpose of this lab is to use the DFT and FFT to compute spectra and study the tradeoffs between frequency resolution and frequency leakage.

Zero Padding and Windowing

Given a length-L signal x_n, $0 \le n \le L-1$, its N-point DFT (assuming $N \ge L$) can be computed by padding x_n with zeros until it has length N, that is,

$$\underbrace{[x_0, x_1, \dots, x_{L-1}]}_{L} \longrightarrow \underbrace{[x_0, x_1, \dots, x_{L-1}, 0, 0, \dots, 0]}_{N} \tag{5.5}$$

This extended signal can then be passed into an N-point DFT or FFT routine. If the signal x_n is to be prewindowed by a non-rectangular window, this operation must be done *before* it is extended to length N. For example, the length-L *Hamming* window is

$$w_n = 0.54 - 0.46\cos\left(\frac{2\pi n}{L-1}\right), \qquad n = 0, 1, \dots, L-1 \tag{5.6}$$

The Hamming windowed signal will be obtained by replacing x_n by $x_n w_n$ for $0 \le n \le L-1$; afterwards, it may be padded to length N for the purpose of computing its DFT:

$$[x_0, x_1, \dots, x_{L-1}] \longrightarrow [x_0 w_0,\ x_1 w_1,\ \dots,\ x_{L-1} w_{L-1}]$$

$$\longrightarrow [x_0 w_0,\ x_1 w_1,\ \dots,\ x_{L-1} w_{L-1}, 0, 0, \dots, 0] \tag{5.7}$$

Lab Procedure

The following signal consists of three sinusoids of frequencies $f_1 = 4$ kHz, $f_2 = 5$ kHz, and $f_3 = 6$ kHz:

$$x(t) = \cos(2\pi f_1 t) + \cos(2\pi f_2 t) + \cos(2\pi f_3 t)$$

where t is in milliseconds. A 5 msec portion of this signal is sampled at a rate of 20 kHz, and the resulting $L = 100$ samples x_n, $n = 0, 1, \dots, L-1$ are saved for further processing.

a. Plot x_n versus n in the range $0 \le n \le L-1$.

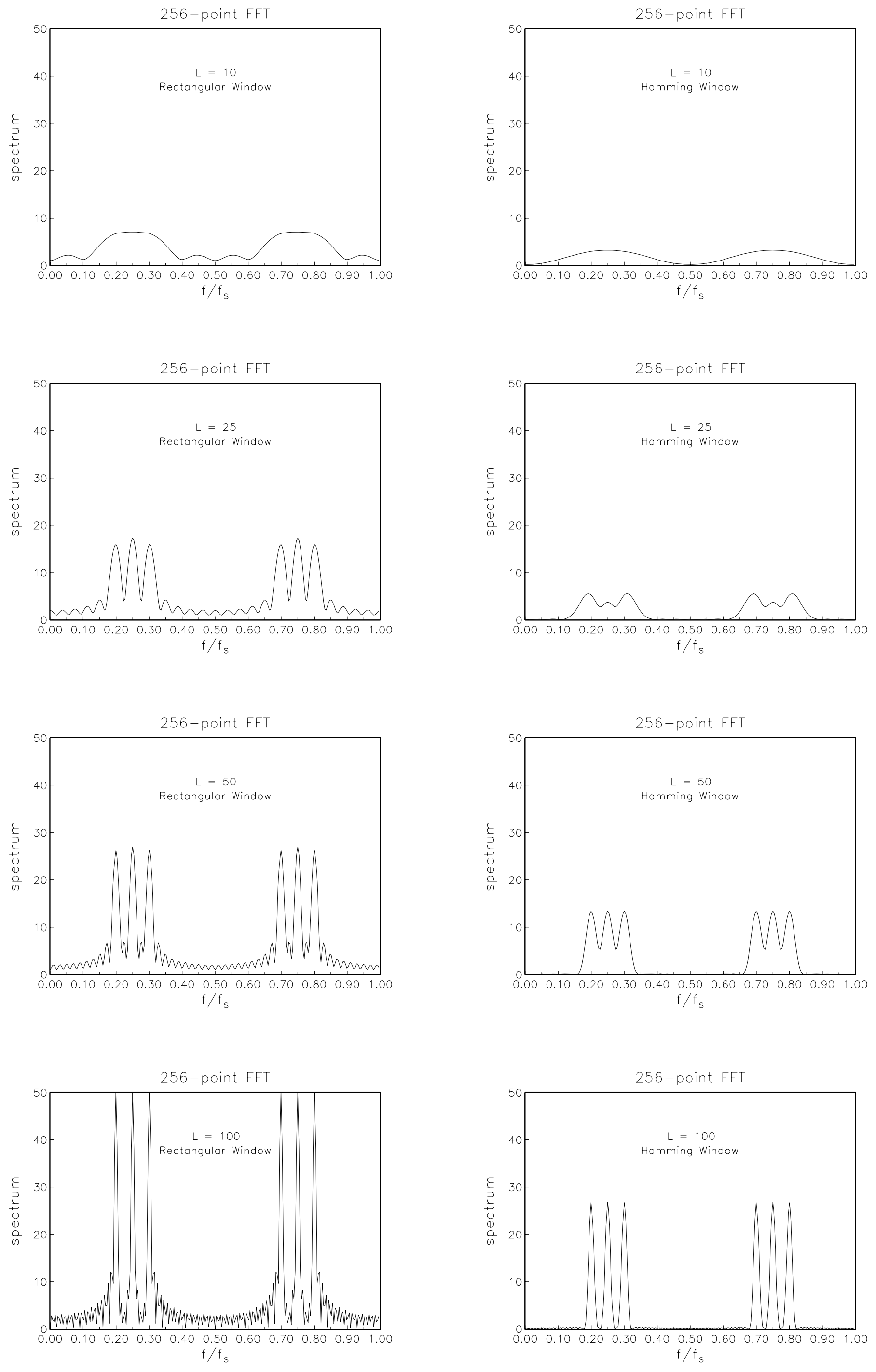
b. Using the routine `fft`, compute the 256-point DFT of the length-L signal x_n. Plot the magnitude of the DFT over one complete Nyquist interval with respect to the normalized frequency f/f_s, that is, over the range $0 \le f/f_s \le 1$. Use vertical scales from $[0, 50]$.

c. Using the routine `ifft`, compute the 256-point inverse FFT of the result in part (b) and verify that you recover the original time signal (including the zeros that were padded at its tail).

d. Window the L samples x_n using a length-L Hamming window as in Eqs. 5.6 and (5.7). Plot the windowed signal $x_n w_n$ versus n using the same vertical scales as in part (a).

e. Repeat part (b) for the Hamming windowed signal $x_n w_n$. Discuss the tradeoff between reducing the sidelobe levels (smaller sidelobes) and worsening of frequency resolution (peaks are less sharp).

f. Repeat parts (b) and (e) when L is reduced to $L = 50$, $L = 25$, and $L = 10$. For both the windowed and unwindowed cases, discuss the worsening of the frequency resolution as the data record becomes shorter and shorter.



g. Consider the following analog signal consisting of three sinusoids:

$$x(t) = \cos(2\pi f_1 t) + 10^{-3}\cos(2\pi f_2 t) + \cos(2\pi f_3 t)$$

where $f_1 = 1.5$, $f_2 = 2.5$, and $f_3 = 3.5$ kHz. The middle term represents a weak sinusoid whose presence we wish to detect by sampling $x(t)$ and computing its DTFT spectrum. The sampling rate is 10 kHz.

It is desired to assess and compare the use of the rectangular, Hamming, and Kaiser windows for this spectral analysis problem. Because the middle sinusoid is 60 dB below the other two, in order to detect its presence, we must use a window that has sidelobes that are suppressed by at least 60 dB. To get some extra margin, we take the relative sidelobe level to be 10 dB deeper than required, that is, $R = 60 + 10 = 70$ dB and choose its width to be $\Delta f = 0.2$ kHz (i.e., one-fifth the minimum frequency separation.)

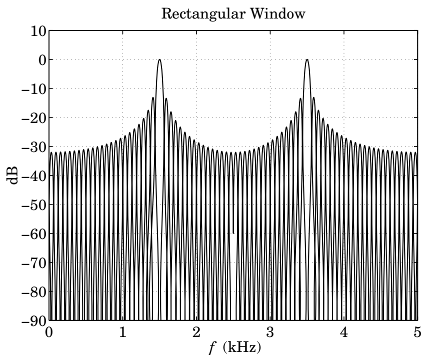
Calculate the length L of a Kaiser window with the above values of R and width Δf.

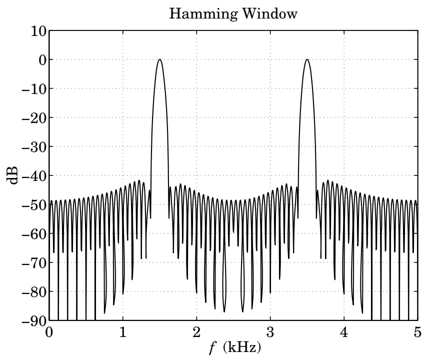
Let $x(t_n)$, $n = 0, 1, \dots, L-1$ be the collected L samples of the above signal. Window this signal with the following three windows of length L: (a) rectangular, (b) Hamming, and (c) Kaiser.

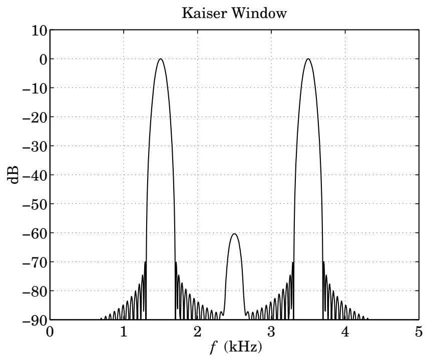
For each case, calculate the corresponding DTFT of the length-L sampled signal at 2000 equally-spaced frequencies in the Nyquist interval $0 \le f \le 5$ kHz. Normalize each DTFT to unity magnitude at its maximum. Plot each normalized DTFT in dB scales over the range $0 \le f \le 5$ kHz.

Observe how the Kaiser window allows the weak sinusoid to rise above the sidelobes, whereas in the rectangular and Hamming window cases the weak sinusoid is buried under the sidelobes.

The Kaiser window can be evaluated with the MATLAB functions `kparm2` and `kwind` of the text [1]. See also the discussion in Sections 10.2.2 of [1] for its application to spectrum estimation problems.







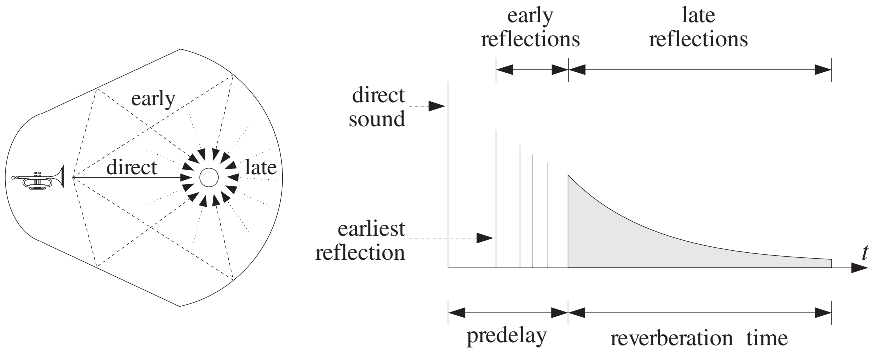
5.5. References

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

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:



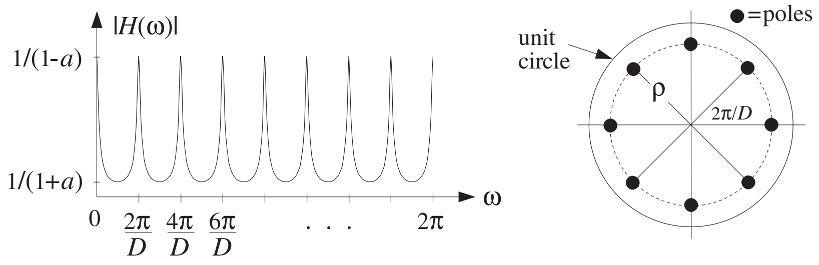
The early reflections correspond to the first few reflections off the walls of the room. As the waves continue to bounce off the walls, their density increases and they disperse, arriving at the listener from all directions. This is the late reflection part.

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.

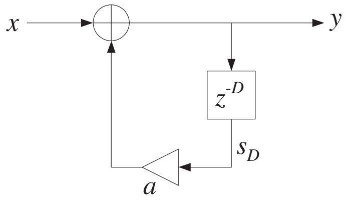
In this and several other labs, we discuss how to emulate such reverberation characteristics using DSP filtering algorithms. A *plain reverberator* can be used as an elementary building block for more complicated reverberation algorithms. It is given by Eq. (8.2.12) of the text [1] and shown in Fig.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

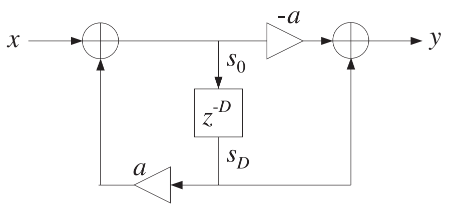
Like the plain reverberator, an allpass reverberator can be used as an elementary building block for building more complicated reverberation algorithms. It is given by Eq. (8.2.25) of the text [1] and shown in Fig. 8.2.17. Its I/O equation and transfer function are:

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

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:

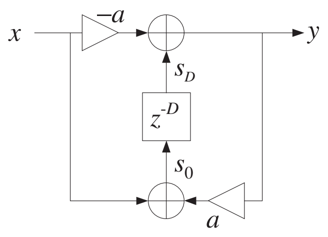
```
// --------------------------------------------------------------------------
// allpass.c - allpass reverb with circular delay line - canonical realization
// --------------------------------------------------------------------------

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

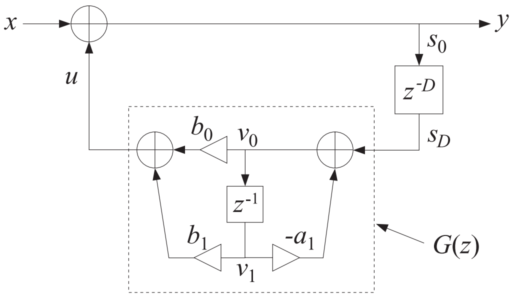
The lowpass reverberator of this experiment is shown in Figs. 8.2.20 and 8.2.21 of Ref. [1]. The feedback gain a of the plain reverb is replaced by a lowpass filter $G(z)$, so that one obtains the new transfer function by the replacement:

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

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:

$$G(z)=\frac{b_0+b_1z^{-1}}{1+a_1z^{-1}}$$

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



```
for each input sample x do:
    $s_D = *(p+D)$
    $v_0 = av_1 + s_D$
    $u = b_0v_0 + b_1v_1$
    $y = x + u$
    $v_1 = v_0$
    $*p = y$
    $--p$
```

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.

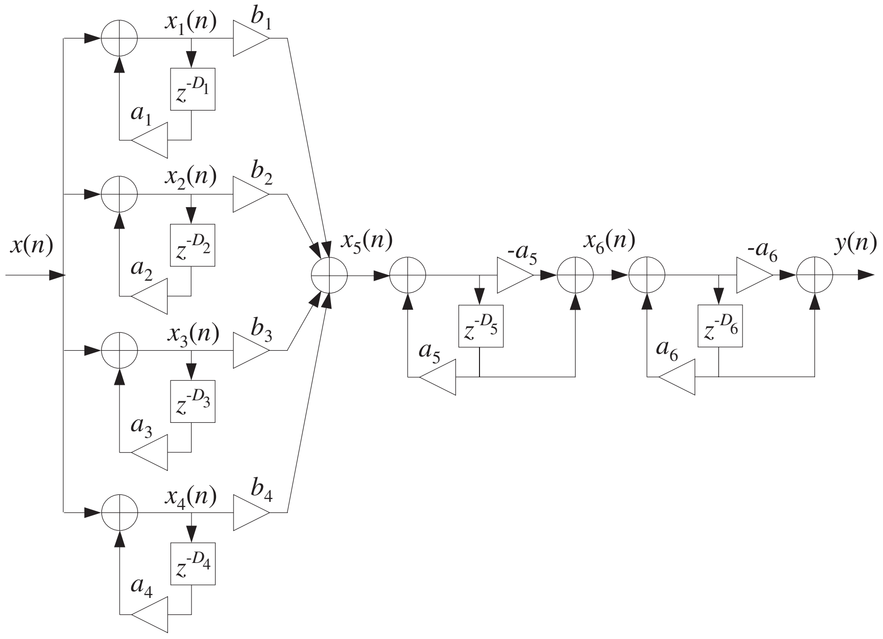
b. Try the case $D = 20$, $a = 0$, $b_0 = b_1 = 0.495$. You will hear a guitar-like sound. Repeat for $D = 100$. What do you hear?

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);
```

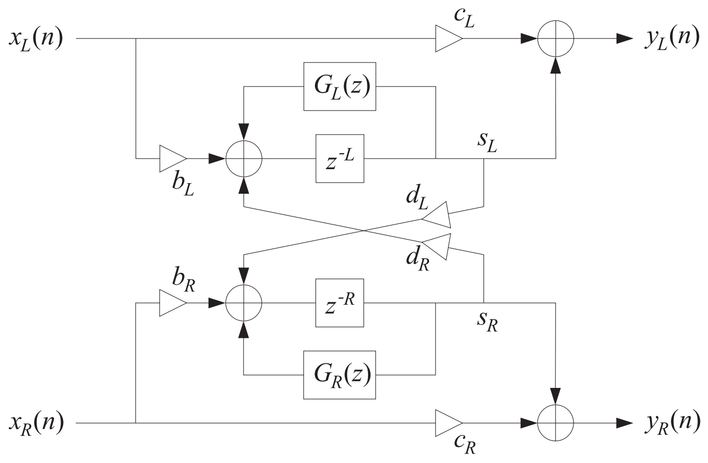
b. What are the feedback delays of each unit in msec? Replace all the delays by double their values, compile, and run again. Compare the output with that of part (a). Repeat when you triple all the delays. (Note that you can just replace the constant definitions by `#define D1 1759*2`, etc.)

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.

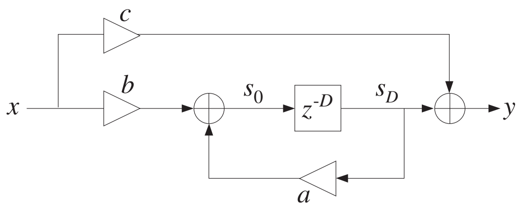
c. Next, introduce some self-feedback, such as $a_L = a_R = 0.2$. Repeat part (a). Vary all the parameters at will to see what you get.

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:

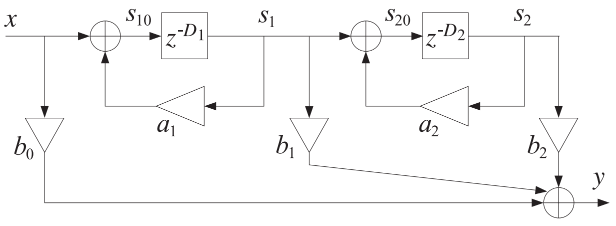
$$H(z) = \frac{1}{1 - az^{-D}}$$

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:

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

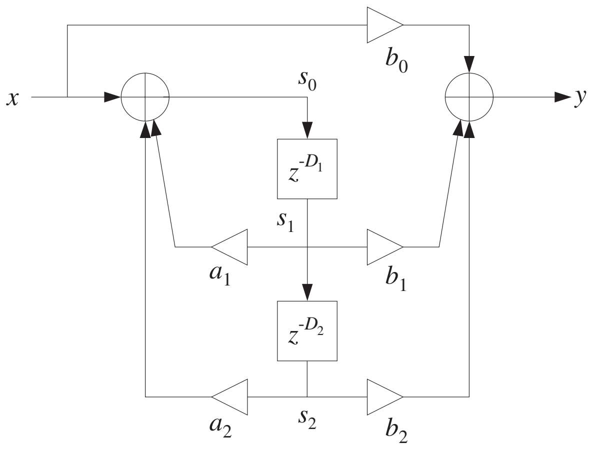
Listen to its impulse response and speak into the mike. Then select LINE input and play a wave file (e.g., `dsummer`) through it.

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:
    $s_1 = *(p + D_1)$
    $s_2 = *(p + D_1 + D_2)$
    $y = b_0x + b_1s_1 + b_2s_2$
    $s_0 = x + a_1s_1 + a_2s_2$
    $*p = s_0$
    $--p$
```

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:

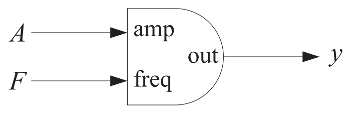
$$w[n] = \begin{cases} +1, & \text{if} \quad 0 \le n < D/2 \\ -1, & \text{if} \quad D/2 \le n < D \end{cases}$$

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:
    $y = A\,w[q]$
    $q = (q + c)\,\mathrm{mod}(D)$
```



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.

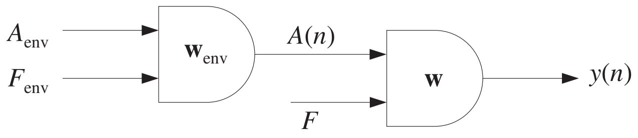
b. Create a GEL file with a slider for the value of the frequency over the interval $0 \le f \le 1$ kHz in steps of 100 Hz. Open the slider and run the program while changing the frequency with the slider.

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:

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

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

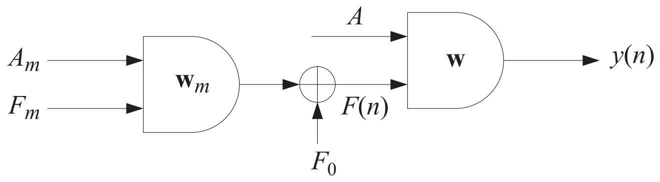
The third program, `fmex.c`, illustrates FM modulation in which the frequency of a sinusoid is time-varying. The generated signal is of the form:

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

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 $A_m = 0.3 f_0$, so that $0.7 f_0 \le f(t) \le 1.3 f_0$. 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

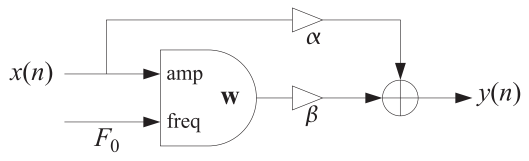
a. Compile, run, and hear the program with the following three choices of the modulation depth: $A_m = 0.3f_0$, $A_m = 0.8f_0$, $A_m = f_0$, $A_m = 0.1f_0$. Repeat these cases when the center frequency is changed to $f_0 = 1000$ Hz.

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

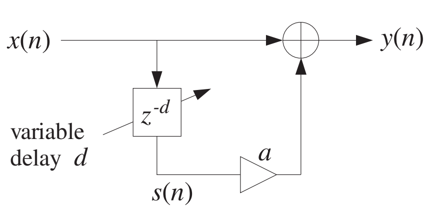
a. Modify the `amex.c` project to implement the ring modulator/tremolo effect. Set the carrier frequency to $f_0 = 400$ Hz and $\alpha = \beta = 1$. Compile, run, and play a wavefile with voice in it (e.g., `dsummer`.)

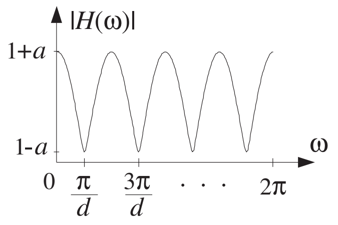
b. Experiment with higher and lower values of f_0.

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 π/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/`.

Lab 7 – IIR Filtering Applications

The first part of this lab illustrates, via a simple IIR filter example, the principles of signal enhancement and noise reduction. The second part illustrates the interplay between steady-state and transient response and the trade-off between time constant and sharpness of filter specifications. The third part looks at some examples of dynamics processors for audio signals, such as compressors, limiters, expanders, and noise gates.

7.1. Signal Enhancement and Noise Reduction

Consider a noisy sinusoidal signal of frequency $f_0 = 500$ Hz sampled at a rate of $f_s = 10$ kHz:

$$x(n) = \cos(\omega_0 n) + v(n) \tag{7.1}$$

where $\omega_0 = 2\pi f_0/f_s$ is the digital frequency and $v(n)$ the noise.

It is desired to design a filter $H(\omega)$ to extract the desired signal $s(n) = \cos(\omega_0 n)$ from the available noisy signal $x(n)$. Such a filter must have two properties: First, it must remove the noise component $v(n)$ as much as possible, and second, it must let the desired signal $s(n)$ pass through unchanged, except possibly for a time delay.

The second of these requirements is met by designing a bandpass filter whose passband coincides with the passband of the desired signal. The noise component is typically a white noise signal whose power is spread equally over the entire frequency axis. After filtering, the overall noise power will be reduced because only the power that resides within the passband of the filter will survive the filtering process.

In our example, the passband of the desired signal is just the frequency ω_0. Therefore, we must design a bandpass filter centered at ω_0 with unity gain at that frequency, namely, $|H(\omega_0)| = 1$. The simplest possible choice for such a filter is a resonator 2-pole filter with poles at $Re^{\pm j\omega_0}$, where R must be chosen to be $0 < R < 1$ for stability. The transfer function will be of the form

$$H(z) = \frac{G}{(1 - Re^{j\omega_0}z^{-1})(1 - Re^{-j\omega_0}z^{-1})} = \frac{G}{1 + a_1 z^{-1} + a_2 z^{-2}} \tag{7.2}$$

where $a_1 = -2R\cos(\omega_0)$ and $a_2 = R^2$. The gain G is fixed so as to ensure $|H(\omega_0)| = 1$. This gives the following expression for G:

$$G = (1 - R)(1 - 2R\cos(2\omega_0) + R^2)^{1/2} \tag{7.3}$$

In the time domain, the filter is described by the difference equation:

$$y(n) = -a_1 y(n-1) - a_2 y(n-2) + Gx(n) \tag{7.4}$$

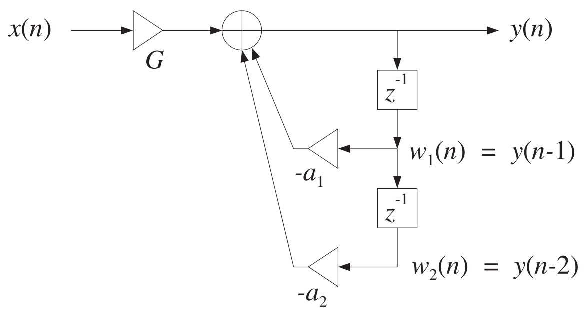
The impulse response of the filter can be shown to be:

$$h(n) = \frac{G}{\sin(\omega_0)} R^n \sin(\omega_0 n + \omega_0), \qquad n = 0, 1, 2, \dots \tag{7.5}$$

Finally, the magnitude response squared can be shown to be:

$$|H(\omega)|^2 = \frac{G^2}{[1 - 2R\cos(\omega - \omega_0) + R^2][1 - 2R\cos(\omega + \omega_0) + R^2]} \tag{7.6}$$

A block diagram implementing the difference equation (7.4) and the transfer function (7.2) is shown below:



The best way to program the difference equation (7.4) is via its sample processing algorithm. To derive it, define the delayed output signals

$$w_1(n)=y(n-1), \qquad w_2(n)=y(n-2)$$

Then, Eq. (7.4) reads

$$y(n)=-a_1w_1(n)-a_2w_2(n)+Gx(n) \tag{7.7a}$$

Once $y(n)$ is computed, the values of $w_1(n)$ and $w_2(n)$ may be updated to the next time instant by

$$\begin{aligned} w_2(n+1) &= w_1(n) \\ w_1(n+1) &= y(n) \end{aligned} \tag{7.7b}$$

These are easily derived from their definition, that is, $w_2(n+1)=y(n+1-2)=y(n-1)=w_1(n)$ and $w_1(n+1)=y(n+1-1)=y(n)$. Equations (7.7), lead to the following simple computational algorithm:

```
For each input sample x do:                    (7.8)
    $y = -a_1w_1 - a_2w_2 + Gx$
    $w_2 = w_1$
    $w_1 = y$
```

Lab Procedure

First, prove Eqs. (7.3) through (7.6). Then, for each of the three values $R = 0.95$, $R = 0.97$, $R = 0.99$, do the following:

a. Plot the magnitude response squared $|H(f)|^2$ over the frequency range $0 \le f \le 5$ kHz. *Suggestion*: For plotting purposes, you may compute $|H(f)|^2$ at 500 equally-spaced frequencies in that range. *Note*: $\omega = 2\pi f/f_s$.

b. Compute the impulse response $h(n)$ of the filter by sending a unit impulse at the input of the difference equation (7.4) and iterating forward in time (with zero initial conditions). Compare the computed values with the values obtained from the formula (7.5). Plot the quantity $h(n)/G$ versus n in the range $0 \le n \le 300$. Use the vertical range of $[-4, 4]$ for your plot.

c. Using MATLAB's built-in random-number generator function, `randn`, generate 300 samples of the white noise signal $v(n)$ and the corresponding noisy sinusoid $x(n)$ of Eq. (7.1). For example, to generate a row of 300 zero-mean unit-variance samples, do:

```
seed = 12345;              % may change this to any integer
randn('state',seed);       % initialize the generator
v = randn(1,300);          % v is a row vector of length 300
```

Then, using the sample processing algorithm (7.8), filter $x(n)$ through the filter $H(z)$ and compute and plot the resulting output signal $y(n)$ versus n in the range $0 \le n \le 300$.

On the same graph, place the desired noise-free signal $s(n)$. Again, use vertical range of $[-4, 4]$. Discuss the trade-off between the sharpness of the bandpass filter (i.e., the goodness of its passband) and the speed of response (i.e., how quickly you reach steady-state) in the filtered signal.

d. Filter the noise signal $v(n)$ separately through this bandpass filter and compute the corresponding filtered output noise, say $y_v(n)$. On two separate graphs, plot $v(n)$ and $y_v(n)$ versus n. Explain why the filtered noise looks more like a sinusoid than noise.

e. It can be shown [1] that for a zero-mean white-noise input $v(n)$ of variance σ_v^2 going through a stable causal filter $h(n)$, the output noise signal $y_v(n)$ will have variance $\sigma_{y_v}^2$ given by the ratio:

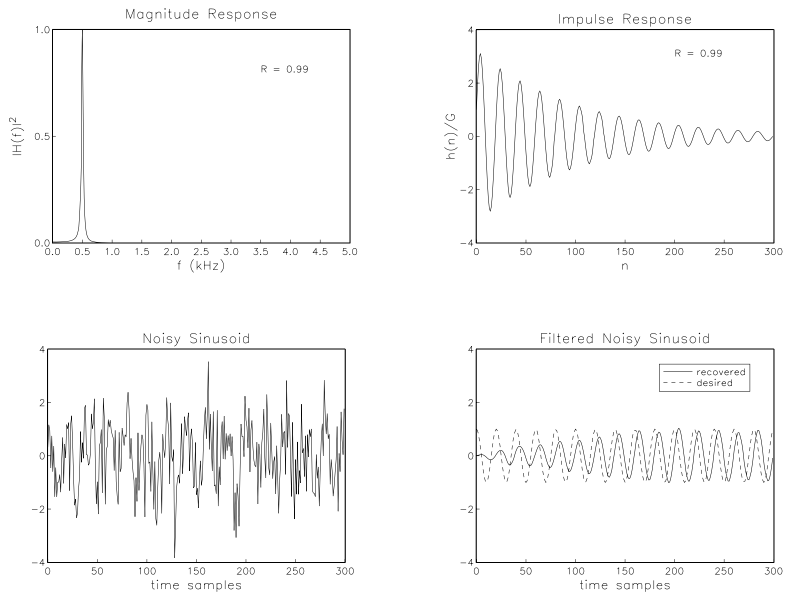
$$\frac{\sigma_{y_v}^2}{\sigma_v^2} = \sum_{n=0}^{\infty} h^2(n) \tag{7.9}$$

The right-hand-side is referred to as the "noise-reduction-ratio" (NRR) because it quantifies the effect of the filter on the input noise. For the particular impulse response of Eq. (7.5), prove that the NRR is given by,

$$\sum_{n=0}^{\infty} h^2(n) = \frac{1 + R^2}{(1 + R)(1 + R^2 + 2R\cos\omega_0)} \tag{7.10}$$

[*Hint:* Write $h(n)$ in the form $h(n) = A_1 p_1^n + A_2 p_2^n$, where p_1, p_2 are the filter poles, then, square and sum the resulting three terms using the infinite geometric series.]

Using the particular sequences $v(n), y_v(n)$ that you generated in part (d), and the built-in MATLAB standard-deviation function, `std()`, calculate an estimated value for the left-hand-side of Eq. (7.9), and compare it with the theoretical value of Eq. (7.10).



7.2. Transient and Steady-State Properties

In Chap. 6 of Ref. [1], the sinusoidal response of a second-order filter with poles at p_1, p_2 is shown to have the following exact form, for $n \ge 0$:

$$x(n) = \cos(\omega_0 n) \quad \Rightarrow \quad y(n) = |H(\omega_0)|\cos(\omega_0 n + \theta_0) + B_1 p_1^n + B_2 p_2^n$$

where the phase shift θ_0 is value of the phase response of the filter at ω_0, and B_1, B_2 depend on the particulars of the transfer function. In this part, you will study how well the steady-state term represents the output of a short-duration signal and the effect of the transient terms on the time constant of the filter.

Consider the following signal of duration of six seconds defined as three concatenated two-second unity-amplitude sinusoidal signals of frequencies $f_1 = 4$, $f_2 = 8$, and $f_2 = 12$ Hz:

$$x(t) = \begin{cases} \cos(2\pi f_1 t), & 0 \le t < 2 \text{ sec} \\ \cos(2\pi f_2 t), & 2 \le t < 4 \text{ sec} \\ \cos(2\pi f_3 t), & 6 \le t < 6 \text{ sec} \end{cases}$$

This signal is sampled at a rate of $f_s = 400$ samples/sec. The following two filters, operating at the rate f_s, are *notch* filters that have been designed to have a notch at $f_2 = 8$ Hz, therefore, they will knock out the middle portion of $x(t)$:

$$H_1(z) = \frac{0.969531 - 1.923772\,z^{-1} + 0.969531\,z^{-2}}{1 - 1.923772\,z^{-1} + 0.939063\,z^{-2}}$$

$$H_2(z) = \frac{0.996088 - 1.976468\,z^{-1} + 0.996088\,z^{-2}}{1 - 1.936468\,z^{-1} + 0.992177\,z^{-2}}$$

The first filter has a 3-dB width of $\Delta f = 4$ Hz, and the second, a width of $\Delta f = 0.5$ Hz. The magnitude responses $|H(f)|$ are shown below. We will learn in Chap. 11 how to design such filters. They have been designed by the MATLAB function `parmeq` from the text [1], invoked with the parameters:

```
[a,b] = parmeq(1, 0, 1/sqrt(2), 2*pi*f2/fs, 2*pi*Df/fs);
```

where `a,b` are the denominator and numerator filter coefficient row vectors. In this experiment, we study the interplay between notch width and transient time constant. The first filter has a wide width and a short time constant, whereas the second filter has a narrow width and a long time constant. The notch filters designed by `parmeq` have the following general form, derive in Ch. 11 of [1]:

$$H(z) = \left(\frac{1}{1+\beta}\right) \frac{1 - 2\cos\omega_0\, z^{-1} + z^{-2}}{1 - 2\left(\dfrac{\cos\omega_0}{1+\beta}\right) z^{-1} + \left(\dfrac{1-\beta}{1+\beta}\right) z^{-2}} \tag{7.11}$$

where the notch frequency and 3-dB notch width are f_0 and Δf and

$$\omega_0 = \frac{2\pi f_0}{f_s}, \quad \Delta\omega = \frac{2\pi \Delta f}{f_s}, \quad \beta = \tan\left(\frac{\Delta\omega}{2}\right) \tag{7.12}$$

Lab Procedure

a. Draw the canonical realization of each filter and write down its sample processing algorithm using linear delay-line buffers. Because these are second-order sections, they may be implemented using the functions `sos.c`, or, `sos.m` of [1].

b. Calculate the 40-dB time constants of both of these filters in seconds.

c. Let $x(t_n)$ denote the sampled input $x(t)$. Plot $x(t_n)$ versus t_n over the period of 6 seconds. Using the function `sos`, filter this input through $H_1(z)$ and plot the output $y(t_n)$ versus t_n.

d. Notice how quickly the middle portion of $x(t_n)$ is notched out. Notice also that the f_1 and f_3 portions no longer have unity-amplitudes. Verify that the (steady-state) amplitudes are given respectively by the magnitude response numbers $|H_1(f_1)|$ and $|H_1(f_3)|$.

   (You can do that either by plotting horizontal lines of such heights $|H_1(f_1)|$ and $|H_1(f_3)|$ over the corresponding signal portions, or, by using MATLAB's function `max` to determine the maximum values of the two portions and then comparing them with the calculated response values.)

   Do you observe a phase shift? Is the observed transient response consistent with the calculation of the time constant of part (b)?

e. Repeat questions (c,d) for the second filter $H_2(z)$.

f. On two separate graphs, plot the magnitude responses $|H_1(f)|$ and $|H_2(f)|$ versus f in the range $0 \le f \le 20$ Hz. The expected graphs are shown below. The values at f_1 and f_3 have been indicated on these graphs.

g. For each filter, calculate the corresponding left and right 3-dB frequencies, say, f_L and f_R, and indicate them on your graphs of part (f) by connecting them with a horizontal segment at the 3-dB level. Verify that the difference $f_R - f_L$ is equal to the given 3-dB widths of 4 Hz and 0.5 Hz, respectively.

   (This is a hard question. You can determine these frequencies analytically if you read Sect. 11.3 of the text, otherwise, you can determine them by trial and error.)

h. Next, consider the following two *peaking* filters, which are complementary to the above notch filters. They have a peak at f_2 and the same 3-dB widths of 4 and 0.5 Hz, respectively:

$$H_1(z) = \frac{0.030469(1 - z^{-2})}{1 - 1.923772\,z^{-1} + 0.939063\,z^{-2}}$$

$$H_2(z) = \frac{0.003912(1 - z^{-2})}{1 - 1.976468\,z^{-1} + 0.992177\,z^{-2}}$$

   They can also be designed with the `parmeq` function:

```
[a,b] = parmeq(0, 1, 1/sqrt(2), 2*pi*f2/fs, 2*pi*Df/fs);
```

   The general design equations for such filters are as follows, where f_0 and Δf are the peak frequency and 3-dB peak width:

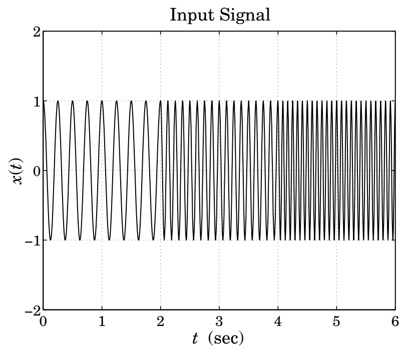
$$H(z) = \left(\frac{\beta}{1+\beta}\right) \frac{1 - z^{-2}}{1 - 2\left(\frac{\cos\omega_0}{1+\beta}\right)z^{-1} + \left(\frac{1-\beta}{1+\beta}\right)z^{-2}} \tag{7.13}$$

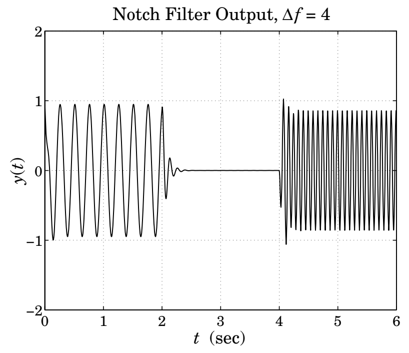
   where

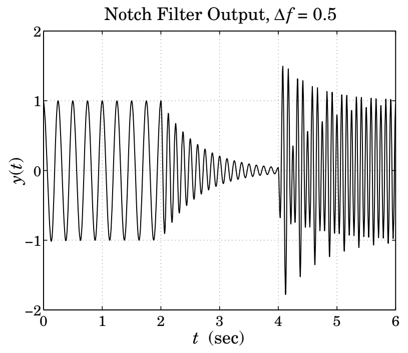
$$\omega_0 = \frac{2\pi f_0}{f_s}\,, \quad \Delta\omega = \frac{2\pi\Delta f}{f_s}\,, \quad \beta = \tan\left(\frac{\Delta\omega}{2}\right)$$

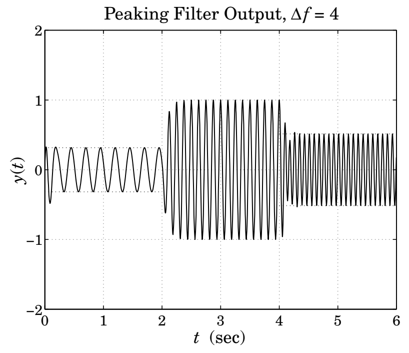
   Repeat questions (a-g) for these filters. The peaking filter is supposed to extract the middle portion of the input and remove the f_1 and f_3 portions. Discuss how well each filter accomplishes this goal and correlate what you see in the time-dependence of the output signals with the magnitude responses of the peaking filters.

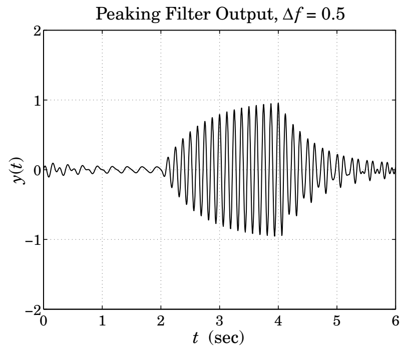
In Lab-8 you will work with similar notch/peaking filters, as well as parametric audio equalizer filters, and program them on the C6713 processor.

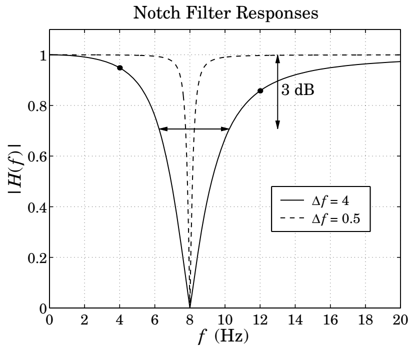

Input Signal
x(t)
t (sec)

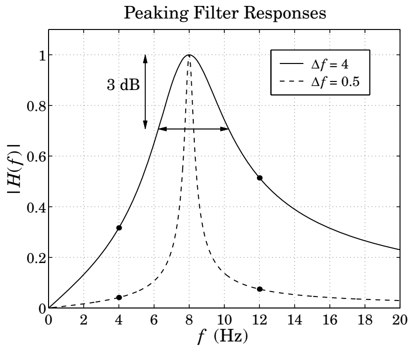

Notch Filter Output, Δf = 4
y(t)
t (sec)


Notch Filter Output, Δf = 0.5
y(t)
t (sec)


Peaking Filter Output, Δf = 4
y(t)
t (sec)


Peaking Filter Output, Δf = 0.5
y(t)
t (sec)


Notch Filter Responses
3 dB
Δf = 4
Δf = 0.5
|H(f)|
f (Hz)


Peaking Filter Responses
3 dB
Δf = 4
Δf = 0.5
|H(f)|
f (Hz)

7.3. Compressors, Limiters, Expanders, and Gates

This experiment is based on Sect. 8.2.5 of Ref. [1]. See Refs. [94–151] included therein for further information on dynamics processors, digital audio effects, and computer music in general. That section is reproduced here.

Compressors, limiters, expanders, and gates have a wide variety of uses in audio signal processing. Compressors attenuate strong signals; expanders attenuate weak signals. Because they affect the dynamic range of signals, they are referred to as *dynamics processors*.

Compressors are used mainly to *decrease* the dynamic range of audio signals so that they fit into the dynamic range of the playback or broadcast system; for example, for putting a recording on audio tape. But there are several other applications, such as announcers "ducking" background music, "de-essing" for eliminating excessive microphone sibilance, and other special effects [2].

Expanders are used for *increasing* the dynamic range of signals, for noise reduction, and for various special effects, such as reducing the sustain time of instruments [2].

A typical steady-state input/output relationship for a compressor or expander is as follows, in absolute and decibel units:

$$y = y_0\left(\frac{x}{x_0}\right)^{\rho} \quad \Rightarrow \quad 20\log_{10}\left(\frac{y}{y_0}\right) = \rho\, 20\log_{10}\left(\frac{x}{x_0}\right) \tag{7.14}$$

where x is here a constant input, x_0 a desired threshold, and ρ defines the compression or expansion ratio. A compressor is effective only for $x \ge x_0$ and has $\rho < 1$, whereas an expander is effective for $x \le x_0$ and has $\rho > 1$. Fig. 7.1 shows these relationships in dB, so that a 1 dB change in the input causes ρ dB change in the output, that is, ρ is the slope of the input/output straight lines.

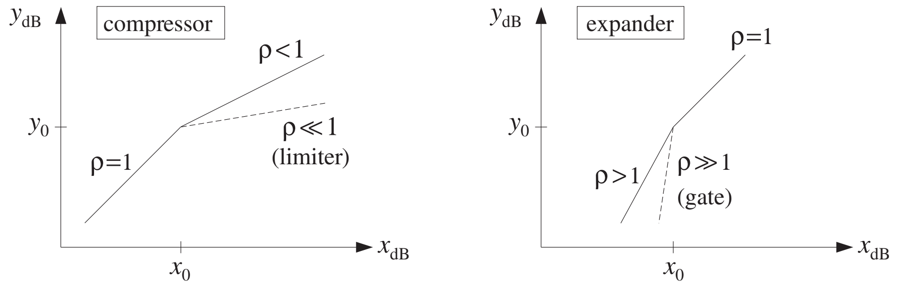


Fig. 7.1 Input/output relationship of compressor or expander.

Typical practical values are $\rho = 1/4$–$1/2$ for compression, and $\rho = 2$–4 for expansion. *Limiters* are extreme forms of compressors that prevent signals from exceeding certain maximum thresholds; they have very small slope $\rho \ll 1$, for example, $\rho = 1/10$. *Noise gates* are extreme cases of expanders that infinitely attenuate weak signals, and therefore, can be used to remove weak background noise; they have very large slopes $\rho \gg 1$, for example, $\rho = 10$.

The I/O equation (7.14) is appropriate only for constant signals. Writing $y = Gx$, we see that the effective gain of the compressor is a nonlinear function of the input of the form $G = G_0 x^{\rho-1}$. For time-varying signals, the gain must be computed from a *local average* of the signal which is representative of the signal's level.

A model of a compressor/expander is shown in Fig. 7.2. The level detector generates a *control signal* c_n that controls the gain G_n of the multiplier through a nonlinear gain processor. Depending on the type of compressor, the control signal may be (1) the instantaneous *peak* value $|x_n|$, (2) the *envelope* of x_n, or (3) the *root-mean-square* value of x_n. A simple model of the envelope detector is as follows:

$$c_n = \lambda c_{n-1} + (1-\lambda)|x_n| \tag{7.15}$$

The difference equation for c_n acts as a *rectifier* followed by a lowpass filter. The time constant of this filter, $n_{\text{eff}} = \ln\epsilon/\ln\lambda$, controls the time to rise or fall to a new input level. The time to rise to a level above the threshold (where the compressor is active) is called the *attack* time constant. This time may be increased further by introducing a delay D in the detector's input, that is, $|x_{n-D}|$. The time to drop to a level below the threshold (where the compressor is inactive) is called the *release* time.

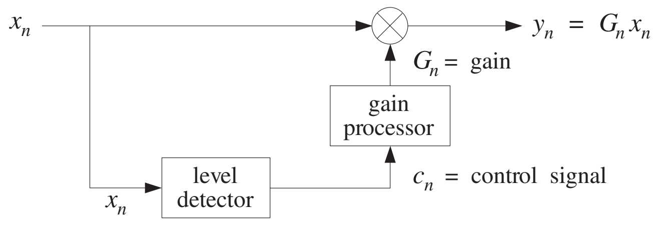


Fig. 7.2 Compressor/expander dynamics processor.

For $\lambda = 0$, Eq. (7.15) becomes an instantaneous peak detector. This case is useful when the compressor is used as a limiter. If in Eq. (7.15) the absolute value $|x_n|$ is replaced by its square, $|x_n|^2$, the control signal will track the *mean-square* value of the input.

The gain processor is a nonlinear function of the control signal imitating the I/O equation (7.14). For a compressor, we may define the gain function to be:

$$f(c)=\begin{cases}(c/c_0)^{\rho-1}, & \text{if } c \geq c_0\\ 1, & \text{if } c \leq c_0\end{cases} \tag{7.16}$$

where c_0 is a desired *threshold* and $\rho < 1$. For an expander, we have $\rho > 1$ and:

$$f(c)=\begin{cases}1, & \text{if } c \geq c_0\\ (c/c_0)^{\rho-1}, & \text{if } c \leq c_0\end{cases} \tag{7.17}$$

Thus, the gain G_n and the final output signal y_n are computed as follows:

$$\begin{aligned} G_n &= f(c_n) \\ y_n &= G_n x_n \end{aligned} \tag{7.18}$$

Compressors/expanders are examples of *adaptive signal processing* systems, in which the filter coefficients (in this case, the gain G_n) are time-dependent and adapt themselves to the nature of the input signals. The level detector (7.15) serves as the "adaptation" equation and its attack and release time constants are the "learning" time constants of the adaptive system; the parameter λ is called the "forgetting factor" of the system.

As a simulation example, consider a sinusoid of frequency $\omega_0 = 0.15\pi$ rads per sample whose amplitude changes to the three values $A_1 = 2$, $A_2 = 4$, and $A_3 = 0.5$ every 200 samples, as shown in Fig. 7.3, that is, $x_n = A_n \cos(\omega_0 n)$, with:

$$A_n = A_1(u_n - u_{n-200}) + A_2(u_{n-200} - u_{n-400}) + A_3(u_{n-400} - u_{n-600})$$

A compressor is used with parameters $\lambda = 0.9$, $c_0 = 0.5$, and $\rho = 1/2$ (that is, 2:1 compression ratio). The output y_n is shown in Fig. 7.3; the control signal c_n and gain G_n in Fig. 7.4.

The first two sinusoids A_1 and A_2 lie above the threshold and get compressed. The third one is left unaffected after the release time is elapsed. Although only the stronger signals are attenuated, the overall reduction of the dynamic range will be *perceived* as though the weaker signals also got amplified. This property is the origin of the popular, but somewhat misleading, statement that compressors attenuate strong signals and amplify weak ones.

Jumping between the steady-state levels A_1 and A_2 corresponds to a 6 dB change. Because both levels get compressed, the steady-state output levels will differ by $6\rho = 3$ dB. To eliminate some of the overshoots, an appropriate delay may be introduced in the main signal path, that is, computing the output by $y_n = G_n x_{n-d}$.

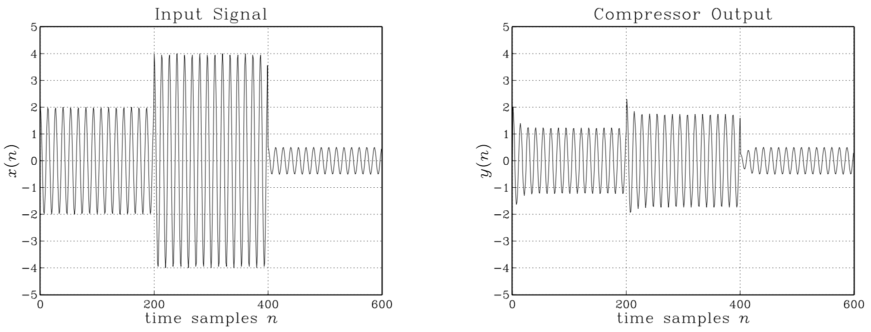


Fig. 7.3 Compressor input and output signals ($\rho = 1/2$, $\lambda = 0.9$, $c_0 = 0.5$).

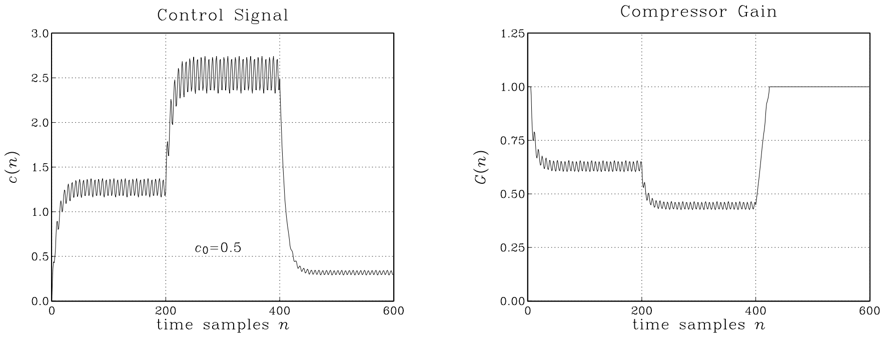


Fig. 7.4 Compressor control signal and gain ($\rho = 1/2$, $\lambda = 0.9$, $c_0 = 0.5$).

Another improvement is to smooth further the nonlinear gain $g_n = f(c_n)$ by a lowpass filter, such as an L-point smoother, so that the final gain is computed by:

$$G_n = \frac{1}{L}\big[g_n + g_{n-1} + \cdots + g_{n-L+1}\big] \tag{7.19}$$

The overall model for a compressor/expander can be summarized as the following sample processing algorithm, expressed with the help of the routine `fir`:

```
for each input sample x do:                          (7.20)
    $c = \lambda c_1 + (1-\lambda)|x|$
    $c_1 = c$
    $G = \text{fir}(M, \mathbf{h}, \mathbf{w}, f(c))$
    $y = Gx$
```

where $\mathbf{h}$ is the impulse response of the L-point smoother, $M = L - 1$ is its order, $\mathbf{w}$ is its L-dimensional delay-line buffer, and c_1 represents c_{n-1}.

Figure 7.5 shows the output signal and compressor gain using a seven-point smoother. The initial transients in G_n are caused by the input-on transients of the smoother.

Figure 7.6 shows the output signal and compressor gain of a limiter, which has a 10:1 compression ratio, $\rho = 1/10$, and uses also a seven-point smoother. The threshold was increased here to $c_0 = 1.5$, so that only A_2 lies above it and gets compressed.

Figure 7.7 shows an example of an expander, with parameters $\lambda = 0.9$, $c_0 = 0.5$, $\rho = 2$, and gain function computed by Eq. (7.17) and smoothed by a seven-point smoother. Only A_3 lies below the threshold and gets attenuated. This causes the overall dynamic range to increase. Although the expander affects only the weaker signals, the overall increase in the dynamic range is perceived as making the stronger signals louder and the weaker ones quieter.

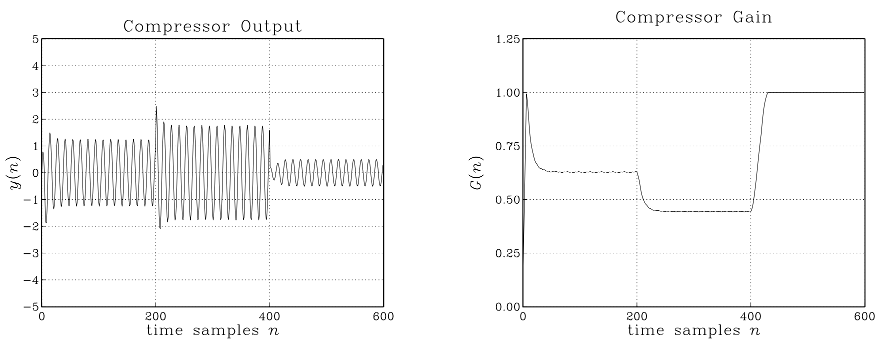


Fig. 7.5 Compressor output with smoothed gain ($\rho = 1/2$, $\lambda = 0.9$, $c_0 = 0.5$).

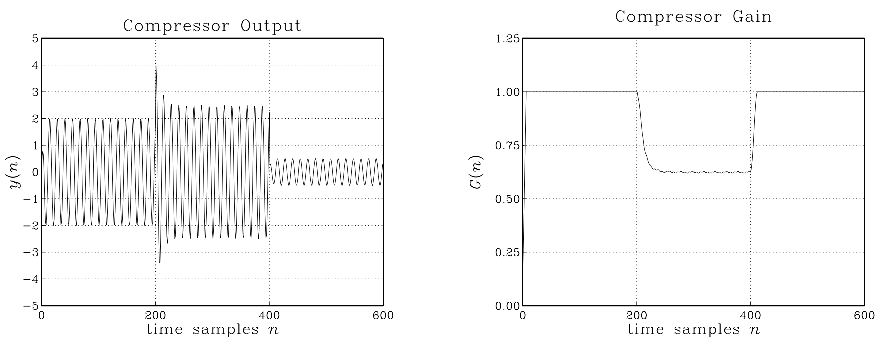


Fig. 7.6 Limiter output with smoothed gain ($\rho = 1/10$, $\lambda = 0.9$, $c_0 = 1.5$).

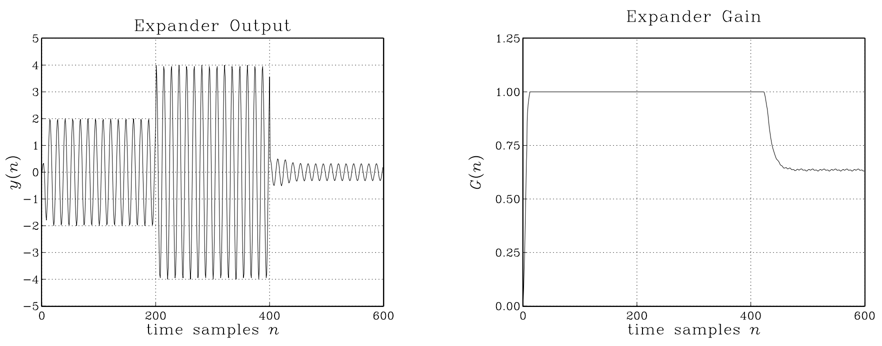


Fig. 7.7 Expander output and gain ($\rho = 2$, $\lambda = 0.9$, $c_0 = 0.5$).

Finally, Fig. 7.8 shows an example of a noise gate implemented as an expander with a 10:1 expansion ratio, $\rho = 10$, having the same threshold as Fig. 7.7. It essentially removes the sinusoid A_3, which might correspond to unwanted noise.

It is possible to modify the algorithm (7.20) to allow different attack and release time constants [3,4]. However, in this lab we will assume they are the same.

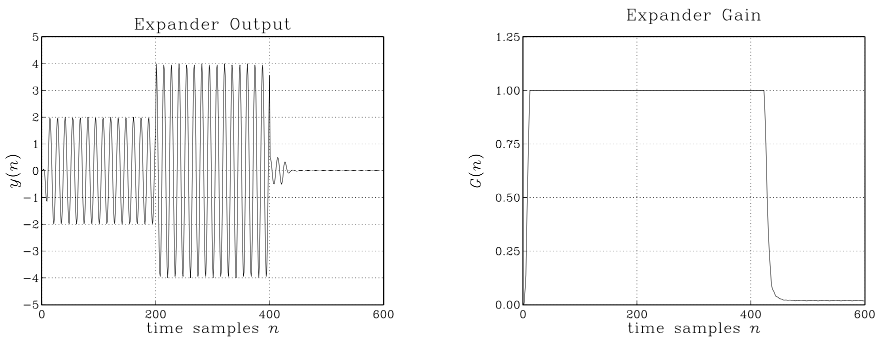


Fig. 7.8 Noise gate output and gain ($\rho = 10$, $\lambda = 0.9$, $c_0 = 0.5$).

Lab Procedure

a. Consider a sinusoid $x(t)= A\cos(2\pi f t)$. Show that its mean-square average over one period, and its absolute average are given by:

$$\overline{x^2(t)} = \frac{1}{2}A^2\,, \quad \overline{|x(t)|} = \frac{2}{\pi}A \tag{7.21}$$

Thus, the peak values $[A_1, A_2, A_3]= [2, 4, 0.5]$ of the three sinusoidal segments will correspond to the following mean absolute values, which will be compared to the assumed compressor threshold:

$$[A_1,\, A_2,\, A_3]= [2,\, 4,\, 0.5] \quad\Rightarrow\quad \frac{2}{\pi}[A_1,\, A_2,\, A_3]= [1.27,\, 2.55,\, 0.32] \tag{7.22}$$

b. Write MATLAB programs to reproduce all the graphs in Figures 7.3–7.8. Include the graphs and your MATLAB code in your report.

Your programs must implement Eq. (7.20) on a sample by sample basis, and therefore, you may not use MATLAB's built-in function `filter()`, but you may use the function, `fir.m`, of the text [1], or write your own version.

Are the plots of the control signal $c(n)$, which measures the average value of $|x(n)|$, consistent with the values of Eq. (7.22)?

c. Redo the compressor plots of Figs. 7.3–7.4, when $\rho = 1/4$, with the other parameters staying the same.

d. Redo the compressor plots of Figs. 7.3–7.4, when the threshold is set to $c_0 = 1.3$, with $\rho = 1/2$ and $\lambda = 0.9$. Explain why only the middle segment gets compressed.

7.4. References

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

[2] B. Hurtig, "The Engineer's Notebook: Twelve Ways to Use Dynamics Processors," *Electronic Musician*, **7**, no. 3, 66 (1991). See also, B. Hurtig, "Pumping Gain: Understanding Dynamics Processors," *Electronic Musician*, **7**, no. 3, 56 (1991).

[3] G. W. McNally, "Dynamic Range Control of Digital Audio Signals," *J. Audio Eng. Soc.*, **32**, 316 (1984), available online from `http://www.bbc.co.uk/rd/publications/rdreport_1983_17.shtml`

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

Lab 8 – Notch, Peak, and Equalizer Filters

8.1. Periodic Notch Filters

In this experiment, we demonstrate the use of filtering for canceling periodic interference. The input signal is of the form:

$$x(n) = s(n) + v(n)$$

where $s(n)$ is the microphone or line input and $v(n)$ a periodic interference signal generated internally by the DSP using a wavetable generator.

When the noise is periodic, its energy is concentrated at the harmonics of the fundamental frequency, f_1, $2f_1$, $3f_1$, and so on. To cancel the entire noise component, we must use a filter with multiple notches at these harmonics.

As discussed in Section 8.3.2 of the text [1], a simple design can be given when the period of the noise is an integral multiple of the sampling period, that is, $T_1 = DT$, which implies that the fundamental frequency $f_1 = 1/T_1$ and its harmonics will be:

$$f_1 = \frac{f_s}{D}, \quad f_k = kf_1 = k\frac{f_s}{D}, \quad k = 0, 1, 2, \dots \tag{8.1}$$

or, in units of radians per sample:

$$\omega_1 = \frac{2\pi}{D}, \quad \omega_k = k\omega_1 = \frac{2\pi k}{D}$$

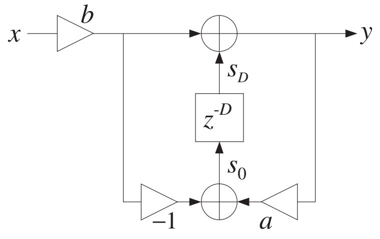
These are recognized as the D-th root-of-unity frequencies. The corresponding notch filter, designed by Eqs. (8.3.26) and (8.3.27) of the text [1], has the form:

$$H(z) = b\,\frac{1 - z^{-D}}{1 - az^{-D}} \tag{8.2}$$

where the parameters a, b depend on the 3-dB notch width Δf as follows:

$$\beta = \tan\left(\frac{D\Delta\omega}{4}\right), \quad \Delta\omega = \frac{2\pi\Delta f}{f_s}, \quad a = \frac{1-\beta}{1+\beta}, \quad b = \frac{1}{1+\beta} \tag{8.3}$$

The numerator of Eq. (8.2) has zeros, notches, at the Dth root-of-unity frequencies (8.1). To avoid potential overflows, we use the transposed realization of this transfer function. Its block diagram and sample processing algorithm are shown below:



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

A quick way to understand the transposed realization is to write:

$$H(z) = \frac{Y(z)}{X(z)} = b\,\frac{1 - z^{-D}}{1 - az^{-D}},$$

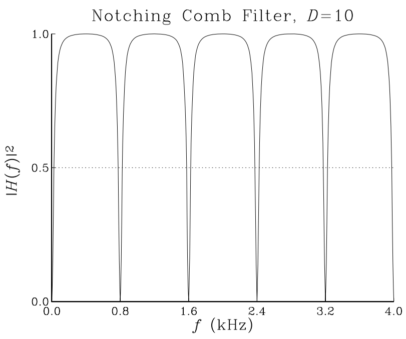
from where we obtain the I/O equation on which the block diagram is based:

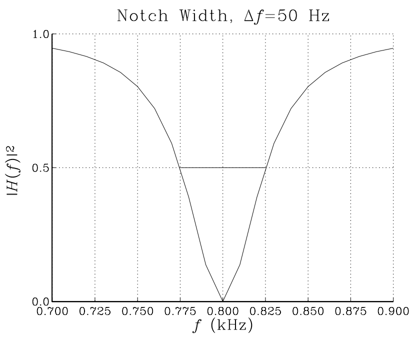
$$Y(z) = bX(z) + z^{-D}\big(aY(z) - bX(z)\big)$$

In the experiment, we take the fundamental period to be 800 Hz and the sampling rate 8 kHz. Thus, the period D is:

$$D = \frac{f_s}{f_1} = \frac{8000\text{ Hz}}{800\text{ Hz}} = 10$$

The width of the notches is taken to be $\Delta f = 50$ Hz. The design equations (8.3.27) give the parameter values $b = 0.91034$, $a = 0.82068$. The magnitude response of this filter plotted over the right-half of the Nyquist interval is shown below, together with a magnified view of the notch width at 800 Hz:





Using a square wavetable, the program `multinotch.c` listed below generates a square wave of period $D = 10$ and adds it to the microphone or line input. The resulting signal is then filtered by the above multi-notch filter, removing the periodic noise. The filtering operation can be bypassed in order to hear the desired signal plus the noise. The particular square wave of period 10 generated by the program has the form:

$$v(n) = [\underbrace{1,1,1,1,1,-1,-1,-1,-1,-1}_{\text{one period}},\dots]$$

It contains only odd harmonics. As discussed in Example 1.4.6 and Section 9.7 of the text [1], all harmonics that lie outside the Nyquist interval are wrapped inside the interval and get aliased with the harmonics within the interval. Thus, the above periodic signal contains only the harmonics $\omega_1 = 2\pi/10$, $\omega_3 = 3\omega_1 = 6\pi/10$, and $\omega_5 = 5\omega_1 = 10\pi/10 = \pi$. In fact, we can show using the techniques of Section 9.7 of the text [1] that the signal $v(n)$ can be expressed in the alternative sinusoidal form, obtained from the 10-point DFT of one period of the square wave:

$$v(n) = \frac{0.4}{\sin(\frac{\omega_1}{2})}\sin(\omega_1 n + \frac{\omega_1}{2}) + \frac{0.4}{\sin(\frac{\omega_3}{2})}\sin(\omega_3 n + \frac{\omega_3}{2}) + 0.2\cos(\omega_5 n)$$

Thus, the filter acts to remove these three odd harmonics. It may appear puzzling that the Fourier series expansion of this square wave does not contain exclusively sine terms, as it would in the continuous-time case. This discrepancy can be traced to the discontinuity of the square wave. In the continuous-time case, any finite Fourier series sinusoidal approximation to the square wave will vanish at the discontinuity points. Therefore, a more appropriate discrete-time square wave might be of the form:

$$v(n) = [\underbrace{0,1,1,1,1,0,-1,-1,-1,-1}_{\text{one period}},\dots]$$

Again, using the techniques of Section 9.7, we find for its inverse DFT expansion:

$$v(n) = \frac{0.4}{\tan(\frac{\omega_1}{2})}\sin(\omega_1 n) + \frac{0.4}{\tan(\frac{\omega_3}{2})}\sin(\omega_3 n)$$

where now only pure sines (as opposed to sines and cosines) appear. The difference between the above two square waves represents the effect of the discontinuities and is given by

$$v(n) = [1,0,0,0,0,-1,0,0,0,0,\dots]$$

Its discrete Fourier series is the difference of the above two and contains only cosine terms:

$$v(n) = 0.4\cos(\omega_1 n) + 0.4\cos(\omega_3 n) + 0.2\cos(\omega_5 n)$$

The following program, `multinotch.c`, implements this example:

```
// multinotch.c - periodic notch filter
// ------------------------------------------------------------------------------------

#include "dsplab.h"          // DSK initialization declarations and function prototypes
#include <math.h>
#define PI 4*atan(1.0)

short xL, xR, yL, yR;        // left and right input and output samples from/to codec
int on = 1;                  // turn filter on, use with GEL file on.gel

#define D 10
float w[D+1], *p;
float v[D] = {1,1,1,1,1,-1,-1,-1,-1,-1};      // square wavetable of period D
//float v[D] = {0,1,1,1,1,0,-1,-1,-1,-1};     // alternative square wavetable
float A = 1000;                               // noise strength
int q;                                        // square-wavetable circular index

float fs=8000, f1=800, Df=50;       // fundamental harmonic and notch width in Hz
float be, Dw, a, b;                 // filter parameters

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

void main()
{
   int i;

   for (i=0; i<=D; i++) w[i] = 0;          // initialize filter's buffer
   p = w;                                  // initialize circular pointer
   q = 0;                                  // initialize square wavetable index

   Dw = 2*PI*Df/fs;                        // design multinotch filter
   be = tan(D*Dw/4);
   a = (1-be)/(1+be); b = 1/(1+be);        // be=0.098491, b=0.910339, a=0.820679

   initialize();              // initialize DSK board and codec, define interrupts
   sampling_rate(8);          // 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()
{
   float x,y,sD;

   read_inputs(&xL, &xR);

   x = (float)(xL);                // work with left input only
   x = x + A*v[q];                 // add square-wave noise
   q = qwrap(D-1,++q);             // wrap q mod-D, noise period is D

   if (on) {
      sD = *pwrap(D,w,p+D);        // filter's transposed realization
      y = b*x + sD;
      *p = a*y - b*x;
      p = pwrap(D,w,--p);

      yL = (short)(y);             // output with filtered noise
      }
   else
      yL = (short) x;              // output noisy input

   write_outputs(yL,yL);

   return;
}

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

Lab Procedure

a. Compile and run the program with the filter off. Play a wave file or speak into the mike and listen to the interference. Repeat with the filter on. Repeat with the filter on, but using the second wavetable. Turn off the interference by setting its amplitude $A = 0$ and listen to the effect the filter has on the line or mike input. [You may use the GEL file, `on.gel`, to turn the filter on and off in real time.]

b. Estimate the 60-dB time constant (in seconds) of the filter in part (a). Redesign the notch filter so that its 3-dB width is now $\Delta f = 1$ Hz. What is the new time constant? Run the new filter and listen to the filter transients as the steady-state gradually takes over and suppresses the noise. Turn off the square wave, recompile and run with MIC input. Listen to the impulse response of the filter by lightly tapping the mike on the table. Can you explain what you are hearing?

c. Generate a square wave with a fundamental harmonic of 1000 Hz, but leave the filter (with 50 Hz width) unchanged (you will need to use two different D's for that). Repeat part (a). Now the interference harmonics do not coincide with the filter's notches and you will still hear the interference.

d. Design the correct multi-notch filter that should be used in part (c). Run your new program to verify that it does indeed remove the 1000 Hz interference.

8.2. Single-Notch Filters

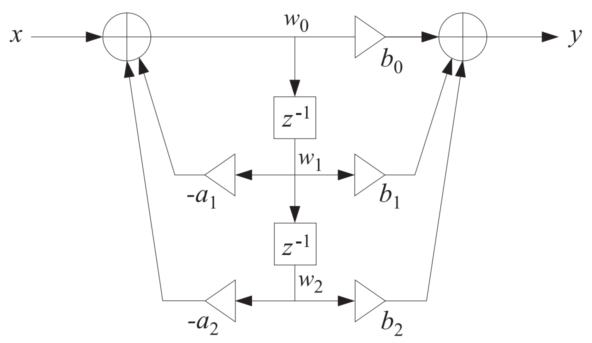
A single-notch filter with notch frequency f_0 and 3-dB notch width Δf can be designed using Eq. (11.3.5) of the text [1], and implemented by the MATLAB function `parmeq.m` of [1]:

$$H(z) = \left(\frac{1}{1+\beta}\right)\frac{1 - 2\cos\omega_0\, z^{-1} + z^{-2}}{1 - 2\left(\dfrac{\cos\omega_0}{1+\beta}\right)z^{-1} + \left(\dfrac{1-\beta}{1+\beta}\right)z^{-2}} \tag{8.4}$$

where

$$\omega_0 = \frac{2\pi f_0}{f_s}\,,\quad \Delta\omega = \frac{2\pi\Delta f}{f_s}\,,\quad \beta = \tan\left(\frac{\Delta\omega}{2}\right) \tag{8.5}$$

Such filters may be realized in their canonical form (also known as direct-form-2) using linear delay-line buffers as shown below:



```
for each x do:
    $w_0 = x - a_1 w_1 - a_2 w_2$
    $y = b_0 w_0 + b_1 w_1 + b_2 w_2$
    $w_2 = w_1$
    $w_1 = w_0$
```
(8.6)

The block diagram implements the second-order transfer function:

$$H(z)=\frac{b_0+b_1z^{-1}+b_2z^{-2}}{1+a_1z^{-1}+a_2z^{-2}}$$

Circular buffers and other realization forms, such as the transposed form, are possible, but the canonical realization will suffice for these experiments.

The following program, `notch0.c`, implements such a notch filter. The program takes as inputs the parameters $f_s, f_0, \Delta f$, performs the design within `main()` before starting filtering, and implements the filter in its canonical form (8.6) within its `isr()` function:

```
// notch0.c - single notch filter
// ------------------------------------------------------------------------------------

#include "dsplab.h"          // DSK initialization declarations and function prototypes
#include <math.h>
#define PI 4*atan(1.0)

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

float w[3], a[3], b[3];      // filter state vector and coefficients
float be, c0;                // filter parameters

int on = 1;

float fs=8000, f0=800, Df=50;     // notch frequency and width in Hz

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

void main()
{
   c0 = cos(2*PI*f0/fs);                                 // design notch filter
   be = tan(PI*Df/fs);
   a[0] = 1; a[1] = -2*c0/(1+be); a[2] = (1-be)/(1+be);
   b[0] = 1/(1+be); b[1] = -2*c0/(1+be); b[2] = 1/(1+be);
   w[1] = w[2] = 0;                                      // initialize filter states

   initialize();             // initialize DSK board and codec, define interrupts
   sampling_rate(8);         // 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()
{
   float x, y;                // filter input & output

   read_inputs(&xL, &xR);

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

      w[0] = x - a[1]*w[1] - a[2]*w[2];
      y = b[0]*w[0] + b[1]*w[1] + b[2]*w[2];
      w[2] = w[1]; w[1] = w[0];

      yL = (short)(y);
      }
   else
      yL = xL;

   write_outputs(yL,yL);
   return;
}

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

The canonical realization can be implemented by its own C function, `can.c`, that can be called within an ISR. The following version is based on that given in the text [1]:

```
// ----------------------------------------------------------------
// can.c - IIR filtering in canonical form
// usage: y = can(M, b, a, w, x);
// ----------------------------------------------------------------

float can(int M, float *b, float *a, float *w, float x)
{
       int i;
       float y = 0;

       w[0] = x;                          // current input sample

       for (i=1; i<=M; i++)               // input adder
              w[0] -= a[i] * w[i];

       for (i=0; i<=M; i++)               // output adder
              y += b[i] * w[i];

       for (i=M; i>=1; i--)               // reverse updating of w
              w[i] = w[i-1];

       return y;                          // current output sample
}
// ----------------------------------------------------------------
```

Thus, the ISR for `notch0.c` can be replaced by

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

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

   read_inputs(&xL, &xR);

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

   if (on) {
      y = can(2, b, a, w, x);     // filter using canonical form
      yL = (short) y;
      }
   else
      yL = (short) x;

   write_outputs(yL,yL);

   return;
}

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

Lab Procedure

a. Compile and run the program `notch0.c`. Using MATLAB, generate a three-second signal consisting of three one-second portions of 1500, 800, 1500 Hz sinusoids, e.g., using the code,

```
f1 = 1500; f2 = 800; f3 = 1500; fs = 8000;
L=8000; n = (0:L-1);
A = 1/5;

x1 = A * cos(2*pi*n*f1/fs);
x2 = A * cos(2*pi*n*f2/fs);
x3 = A * cos(2*pi*n*f3/fs);

sound([x1,x2,x3], fs);
```

and send it to the LINE input of the DSK. Run the program with the filter off and then on to hear the filtering action. Redesign the filter so that the notch width is $\Delta f = 2$ Hz and replay the above signal with the filter on/off, and listen to the audible filter transients.

b. Redo part (a), but now use the function, `can.c`, within the ISR.

8.3. Double-Notch Filters

Using a single-notch filter with notch frequency at $f_1 = 800$ Hz in the square-wavetable experiment, instead of the multi-notch filter, would not be sufficient to cancel completely the square wave interference. The third and higher harmonics will survive it. Assuming the same width $\Delta f = 50$ Hz, the transfer function of Eq. (8.4) becomes explicitly:

$$H_1(z) = \frac{0.980741 - 1.586872z^{-1} + 0.980741z^{-2}}{1 - 1.586872z^{-1} + 0.961481z^{-2}} \tag{8.7}$$

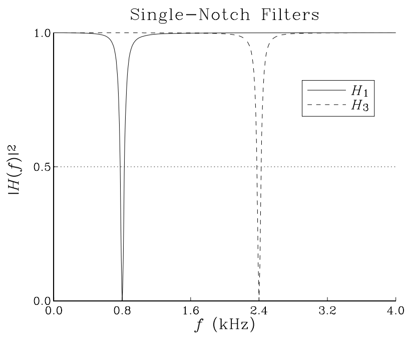
A similar design with a notch at $f_3 = 3f_1 = 2400$ Hz gives:

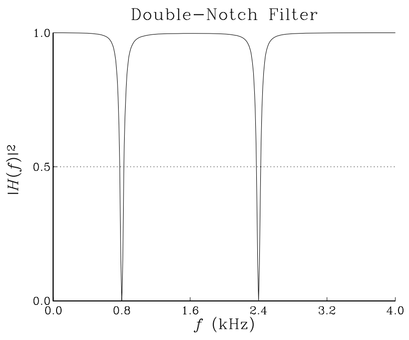
$$H_3(z) = \frac{0.980741 + 0.606131z^{-1} + 0.980741z^{-2}}{1 + 0.606131z^{-1} + 0.961481z^{-2}} \tag{8.8}$$

The cascade of the two is a fourth-order filter of the form $H_{13}(z) = H_1(z)H_3(z)$ with coefficients obtained by convolving the coefficients of filter-1 and filter-3:

$$H_{13}(z) = \frac{0.961852(1 - z^{-1} + z^{-2} - z^{-3} + z^{-4})}{1 - 0.980741z^{-1} + 0.961111z^{-2} - 0.942964z^{-3} + 0.924447z^{-4}} \tag{8.9}$$

The magnitude responses of the two single-notch filters $H_1(z)$, $H_3(z)$ and of the double-notch filter $H_{13}(z)$ are shown below:





Lab Procedure

a. Modify the program, `multinotch.c`, so that it uses only the filter $H_1(z)$ that has a single notch at $f_1 = 800$ Hz. Run it with the filter off and then turn the filter on. Do you hear the partial suppression of the interference?

b. Next, modify the program to use both filters $H_1(z)$ and $H_3(z)$ in cascade. This can be implemented by the following ISR function:

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

interrupt void isr()
{
   float x, y, y1;                // filter inputs & outputs

   read_inputs(&xL, &xR);

   x = (float)(xL);               // work with left input only
   x = x + A*v[q];                // add square-wave noise
   q = qwrap(D-1,++q);            // update square-wavetable index

   if (on) {
      y1 = can(2, b1, a1, w1, x);      // filter noisy input by H1
      y  = can(2, b3, a3, w3, y1);     // filter output of H1 by H3
      yL = (short) y;
      }
   else
      yL = (short) x;             // output noisy input

   write_outputs(yL,yL);

   return;
}

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

where `b1,a1,w1` are the numerator, denominator, and state vectors of filter $H_1(z)$, and similarly, `b3,a3,w3` are those of filter $H_3(z)$. These coefficients must be designed within `main()`.

Run your program using the first square-wavetable choice. Listen to the suppression of the harmonics at f_1 and f_3. However, the harmonic at $f_5 = 4000$ Hz (i.e., the Nyquist frequency) can still be heard.

c. Next, run your program using the second square-wavetable choice. You will hear no interference at all because your square wave now has harmonics only at f_1 and f_3 which are canceled by the double-notch filter.

d. Modify your program to use the combined 4th-order filter of Eq. (8.9) implemented by a single call to `can` (with $M = 4$). Verify that it behaves similarly to that of part (c). You may use the approximate numerical values of the filter coefficients shown in Eq. (8.9).

e. Can you explain theoretically why in the numerator of $H_{13}(z)$ you have the polynomial with alternating coefficients $(1 - z^{-1} + z^{-2} - z^{-3} + z^{-4})$?

8.4. Peaking Filters

Peaking or resonator filters are used for enhancing a desired sinusoidal component. They are discussed in Ch.11 of the text [1] and you have used them in Lab-7. With a peak at f_0, 3-dB peak width of Δf, and unity peak gain, the corresponding transfer function is given by Eq. (11.3.18) of [1]:

$$H(z) = \left(\frac{\beta}{1+\beta}\right)\frac{1 - z^{-2}}{1 - 2\left(\dfrac{\cos\omega_0}{1+\beta}\right)z^{-1} + \left(\dfrac{1-\beta}{1+\beta}\right)z^{-2}} \tag{8.10}$$

where

$$\omega_0 = \frac{2\pi f_0}{f_s}\,, \quad \Delta\omega = \frac{2\pi\Delta f}{f_s}\,, \quad \beta = \tan\left(\frac{\Delta\omega}{2}\right)$$

Lab Procedure

a. Modify the program, `notch0.c`, to implement the design and operation of such a peaking filter. For example, the function `main()` can be replaced by:

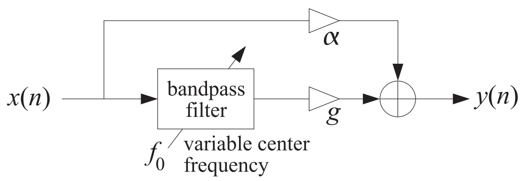
```
// ----------------------------------------------------------------------------------
void main()
{
   c0 = cos(2*PI*f0/fs);                                  // design notch filter
   be = tan(PI*Df/fs);
   a[0] = 1; a[1] = -2*c0/(1+be); a[2] = (1-be)/(1+be);
   b[0] = be/(1+be); b[1] = 0; b[2] = -be/(1+be);
   w[1] = w[2] = 0;                                       // initialize filter states

   initialize();              // initialize DSK board and codec, define interrupts
   sampling_rate(8);          // 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()
}
// ----------------------------------------------------------------------------------
```

Generate the same 1500-800-1500 Hz three-second signal as input. Design your peaking filter to have a peak at $f_0 = 1500$ Hz, so that it will let through the first and last one-second portions of your input. For the 3-dB width choose first $\Delta f = 5$ Hz. Can you explain what you hear? Can you hear the filter transients?

b. Repeat part (a) with $\Delta f = 50$ Hz.

8.5. Wah-Wah Filters and Phasers

A *wah-wah filter* is a bandpass filter with variable center frequency, which is usually controlled by a pedal. The output of the filter is mixed with the input to produce the characteristic whistling or voice-like sound of this effect.



The center frequency f_0 is usually varied within the voice-frequency range of 300–3000 Hz and the bandwidth of the filter is typically of the order of 200 Hz.

A *phaser* is a very similar effect, which uses a notch filter, instead of a bandpass filter, and the notch frequency is varied in a similar fashion.

In this experiment, you will implement a wah-wah filter using the peaking resonator filter of Eq. (8.10) of the previous section, and a phaser filter using the notch filter of Eq. (8.4). The center peaking or notch frequency f_0 will be chosen to vary sinusoidally with a sweep-frequency f_{sweep} as follows:

$$f_0(n) = f_c + \Delta f \cdot \sin(2\pi F_{\text{sweep}} n)\,, \quad F_{\text{sweep}} = \frac{f_{\text{sweep}}}{f_s} \tag{8.11}$$

so that f_0 varies between the limits $f_c \pm \Delta f$. The following program, `wahwah.c`, implements a wah-wah filter that uses a wavetable generator to calculate f_0 from Eq. (8.11).

```
// wahwah.c - wah-wah filter
// ------------------------------------------------------------------------------------

#include "dsplab.h"          // DSK initialization declarations and function prototypes
#include <math.h>
#define PI 4*atan(1.0)

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

float w[3], a[3], b[3];      // filter state vector and coefficients

int on = 1;

float fs=8000, fc=1000, Df=500, Bw=200;
float be, c0, f0, alpha=0.2, g=1.5;

#define Ds 8000                      // smallest sweep frequency is fs/Ds = 1 Hz
float v[Ds], fsweep = 1;             // wavetable buffer and sweep frequency of 1 Hz
int q;

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

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

void main()
{
   int i;

   be = tan(PI*Bw/fs);                                  // bandwidth parameter
   b[0] = be/(1+be); b[1] = 0; b[2] = -be/(1+be);      // filter coefficients
   a[0] = 1; a[2] = (1-be)/(1+be);

   w[1] = w[2] = 0;                                     // initialize filter states

   q=0;                                                 // wavetable index
   for (i=0; i<Ds; i++) v[i] = sin(2*PI*i/Ds);         // initialize wavetable

   initialize();              // initialize DSK board and codec, define interrupts
   sampling_rate(8);          // 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()
{
   float x, y;                         // filter input & output

   read_inputs(&xL, &xR);

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

      f0 = fc + wavgen(Ds, v, Df, fsweep/fs, &q);        // variable center frequency
      c0 = cos(2*PI*f0/fs);

      a[1] = -2*c0/(1+be);                               // filter coefficient

      w[0] = x - a[1]*w[1] - a[2]*w[2];                  // filtering operation
      y = b[0]*w[0] + b[1]*w[1] + b[2]*w[2];             // filter output
      w[2] = w[1]; w[1] = w[0];                          // update filter states

      y = alpha * x + g * y;                             // mix with input

      yL = (short)(y);
      }
   else
      yL = xL;

   write_outputs(yL,yL);

   return;
}

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

The 3-dB width (in Hz) of the filter is denoted here by B_w to avoid confusion with the variation width Δf of f_0. Thus, the bandwidth parameter β of Eq. (8.10), and some of the filter coefficients that do not depend on f_0, can be fixed within `main()`, i.e.,

$$\beta = \tan\left(\frac{\pi B_w}{f_s}\right), \quad b_0 = \frac{\beta}{1+\beta}, \quad b_1 = 0, \quad b_2 = -\frac{\beta}{1+\beta}, \quad a_0 = 1, \quad a_2 = \frac{1-\beta}{1+\beta}$$

The denominator coefficient $a_1 = -2\cos(\omega_0)/(1+\beta)$ is computed on the fly within `isr()` at each sampling instant and then the filtering operation is implemented in its canonical form. A phaser can be implemented in a similar fashion, except now the numerator coefficient $b_1 = -2\cos(\omega_0)$ must also be computed on the fly in addition to a_1. The other coefficients can be pre-computed as follows:

$$\beta = \tan\left(\frac{\pi B_w}{f_s}\right), \quad b_0 = b_2 = \frac{\beta}{1+\beta}, \quad a_0 = 1, \quad a_2 = \frac{1-\beta}{1+\beta}$$

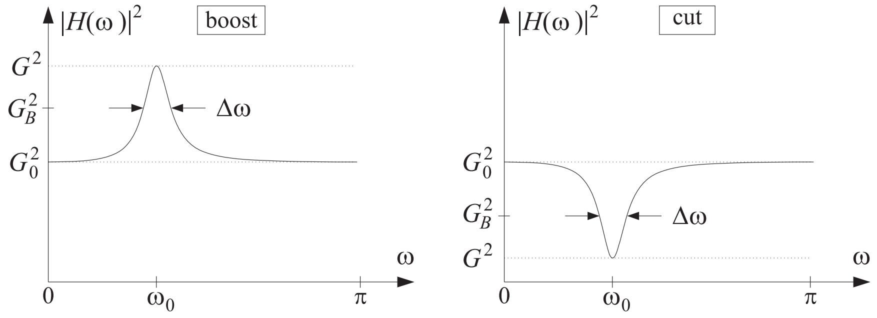
Lab Procedure

a. Create a project for this program and run it on a wave file (e.g., `caravan`, `dsummer`, `trials`). Rerun it with $f_c = 2000$, $\Delta f = 500$ Hz. Repeat with $f_c = 500$, $\Delta f = 500$ Hz.

b. Modify the program to implement a phaser and run it for the above choices of $f_c, \Delta f$.

8.6. Parametric Equalizer Filters

Parametric audio equalizer filters are used to boost or cut the frequency content of an input signal around some center frequency f_0 with a bandwidth of Δf, as shown below.



The design of such filters is discussed in Sect. 11.4 of the text [1]. In terms of the desired boost/cut gain G, bandwidth gain G_B, and reference gain G_0, the transfer function is given by:

$$H(z) = \frac{\left(\dfrac{G_0 + G\beta}{1+\beta}\right) - 2\left(\dfrac{G_0\cos\omega_0}{1+\beta}\right)z^{-1} + \left(\dfrac{G_0 - G\beta}{1+\beta}\right)z^{-2}}{1 - 2\left(\dfrac{\cos\omega_0}{1+\beta}\right)z^{-1} + \left(\dfrac{1-\beta}{1+\beta}\right)z^{-2}} \tag{8.12}$$

where the gains are in absolute units (not dB), and,

$$\omega_0 = \frac{2\pi f_0}{f_s}\,, \quad \Delta\omega = \frac{2\pi\Delta f}{f_s}\,, \quad \beta = \sqrt{\frac{G_B^2 - G_0^2}{G^2 - G_B^2}}\,\tan\left(\frac{\Delta\omega}{2}\right) \tag{8.13}$$

Some possible choices for the bandwidth gain (i.e., the level at which Δf is measured) are:

$$G_B = \sqrt{GG_0} \quad \text{and} \quad G_B^2 = \frac{1}{2}(G^2 + G_0^2) \tag{8.14}$$

In practice, several such 2nd-order filters are used in cascade to cover a desired portion of the audio band. Higher-order designs also exist, but we will not consider them in this lab.

A program implementing a single EQ filter can be structured along the same lines as `notch0.c`. The filter may be designed on the fly within `main()`. For example,

```
void main()
{
   if (Gdb==0)                                                // no boost gain
      { a[0]=b[0]=1; a[1]=a[2]=b[1]=b[2]=0; }                 // H(z) = 1
   else {                                                     // design filter
      G = pow(10.0, Gdb/20);                                  // boost gain
      GB = sqrt(G);                                           // bandwidth gain
      //GB = sqrt((1+G*G)/2);                                 // alternative GB gain
      be = sqrt((GB*GB-1)/(G*G-GB*GB)) * tan(PI*Df/fs);       // bandwidth parameter
      c0 = cos(2*PI*f0/fs);                                   // f0 is center frequency
      a[0] = 1; a[1] = -2*c0/(1+be); a[2] = (1-be)/(1+be);
      b[0] = (1+G*be)/(1+be); b[1] = -2*c0/(1+be); b[2] = (1-G*be)/(1+be);
      }
   w[1] = w[2] = 0;                                           // initialize filter states

   initialize();              // initialize DSK board and codec, define interrupts
   sampling_rate(8);          // 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()
}
```

where the reference gain is assumed to be $G_0 = 1$, and the boost/cut gain is to be specified in dB, so that the absolute gain is computed by the `pow()` function, that is,

$$G = 10^{G_{\text{dB}}/20}$$

One of the two choices (8.14) can be made for the bandwidth gain G_B. The `isr()` function for this filter is identical to that of `notch0.c`.

Lab Procedure

a. Complete the above EQ program. Select the parameters $f_s = 8000$, $f_0 = 800$, and $\Delta f = 50$ Hz and LINE input. Choose the initial value of the boost gain to be $G_{\text{dB}} = -50$ dB, that is, a cut. Compile and run the program. Send as input your usual 3-sec 1500-800-1500 Hz signal from MATLAB.

b. Keep increasing the boost gain G_{dB} to about 15 dB and playing the same signal through until overflow effects begin to be heard. How high is the maximum gain you can set?

c. Repeat parts (a,b), for the alternative definition of the bandwidth gain G_B.

d. The GEL file, `eq.gel`, allows you to interactively select the parameters $f_0, G_{\text{dB}}, \Delta f$. However, for these to have an effect, you will need to move the design equations for the filter inside `isr()`— not an attractive choice. For example,

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

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

   read_inputs(&xL, &xR);

   if (on) {

      if (Gdb==0)
         { a[0]=b[0]=1; a[1]=a[2]=b[1]=b[2]=0; }
      else {
         G = pow(10.0, Gdb/20);
         GB = sqrt(G);
         //GB = (1+G*G)/2;
         be = sqrt((GB*GB-1)/(G*G-GB*GB)) * tan(PI*Df/fs);
         c0 = cos(2*PI*f0/fs);
         a[0] = 1; a[1] = -2*c0/(1+be); a[2] = (1-be)/(1+be);
         b[0] = (1+G*be)/(1+be); b[1] = -2*c0/(1+be); b[2] = (1-G*be)/(1+be);
         }

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

      w[0] = x - a[1]*w[1] - a[2]*w[2];
      y = b[0]*w[0] + b[1]*w[1] + b[2]*w[2];
      w[2] = w[1]; w[1] = w[0];

      yL = (short)(y);

      }
   else
      yL = xL;

   write_outputs(yL,yL);

   return;
}

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

Run this program on some wave file, open the sliders for $f_0, G_{\text{dB}}, \Delta f$, and experiment with varying them in real time. One can easily modify the above ISR so that the filter is not being re-designed at every sampling instant, but rather, say every 10 msec, or so.

Profile this version of ISR and that of part (a) to determine the number of cycles between breakpoints set at the read-inputs and write-outputs statements.

8.7. References

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

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