# Lab 2 – Wavetable Generators, AM/FM Modulation

## 2.1. Lab Tasks

The concept of a wavetable is introduced and applied first to the generation of sinusoidal signals of different frequencies and then to square waves. AM and FM examples are constructed by combining two wavetables. Ring modulation and tremolo audio effects are studied as special cases of AM modulation.

## 2.2. 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:

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

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

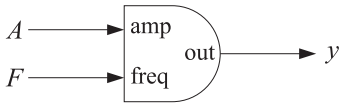
$$w[n] = \begin{cases} +1\,, & \text{if} \quad 0 \le n < D/2 \\ -1\,, & \text{if} \quad D/2 \le n < D \end{cases}$$

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:

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

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

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




(2.1)

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.

## 2.3. Sinusoidal Wavetable

The following program, `sinex.c`, 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. In order to generate $f = 1$ kHz, the step size will be $c = D \cdot f/f_s = 4000 \cdot 1/8 = 500$ samples.

In this example, we will not use the function `wavgen` but rather apply the generation algorithm of Eq. (2.1) explicitly. In addition, we will save the output samples in a buffer array of length $N = 128$ and inspect the generated waveform both in the time and frequency domains using CCS's graphing capabilities.

```
// sinex.c - sine wavetable example
//
// 332:348 DSP Lab - Spring 2012 - S. J. Orfanidis
// ---------------------------------------------------------------------------

#include "dsplab.h"          // DSK initialization declarations and function prototypes
#include <math.h>
//#define PI 3.141592653589793

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
#define N 128                // buffer length

short fs=8;                  // fs = 8 kHz
float c, A=5000, f=1;        // f = 1 kHz
float w[D];                  // wavetable buffer
float buffer[N];             // buffer for plotting with CCS
int q=0, k=0;

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

void main()                  // main program executed first
{
  int n;
  float PI = 4*atan(1);

  for (n=0; n<D; n++) w[n] = sin(2*PI*n/D);     // load wavetable with one period

  c = D*f/fs;                // step into wavetable buffer

  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);      // wait for interrupts
}
```

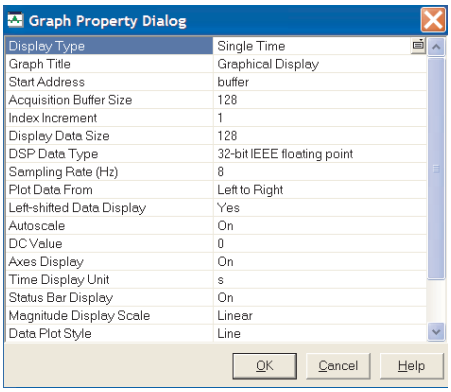
```
// ---------------------------------------------------------------------------

interrupt void isr()
{
   yL = (short) (A * w[q]);                  // generate sinusoidal output
   q = (int) (q+c); if (q >= D) q = 0;       // cycle over wavetable in steps c

   buffer[k] = (float) yL;                   // save into buffer for plotting
   if (++k >= N) k=0;                        // cycle over buffer

   write_outputs(yL,yL);                     // audio output
}

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

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 and sends it to the codec, and saves the sample into a buffer (only the last $N$ generated samples will be in present in that buffer).

### Lab Procedure

a. Create a project for this program and run it. The amplitude was chosen to be $A = 5000$ in order to make the wavetable output audible. Hold the processor after a couple of seconds (SHIFT-F5).

b. Using the keyboard shortcut, "ALT-V RT", or the menu commands *View -> Graph -> Time/Frequency*, open a graph-properties window as that shown below:




Select the starting address to be, `buffer`, set the sampling rate to 8 and look at the time waveform. Count the number of cycles displayed. Can you predict that number from the fact that $N$ samples are contained in that buffer? Next right-click on the graph and select "Properties", and choose "FFT Magnitude" as the plot-type. Verify that the peak is at $f = 1$ kHz.

c. Reset the frequency to 500 Hz. Repeat parts (a,b).

d. 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.

e. 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.)

f. Replace the following two lines in the `isr()` function:

```
yL = (short) (A * w[q]);
q = (int) (q+c); if (q >= D) q = 0;
```

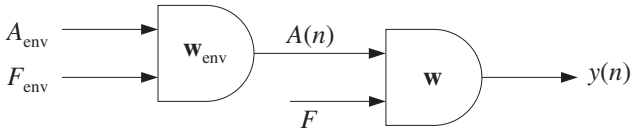
by a single call to the function wavgen, and repeat parts (a,b).

g. Replace the sinusoidal table of part (f) with a square wavetable that has period 4000 and is equal to $+1$ for the first half of the period and $-1$ for the second half. Run the program with frequency $f = 1$ kHz and $f = 200$ Hz.

h. Next, select the sampling rate to be $f_s = 96$ kHz and for the sinusoid case, start with the frequency $f = 8$ kHz and keep increasing it by 2 kHz at a time till about 20 kHz to determine the highest frequency that you can hear—each time repeating parts (a,b).

## 2.4. AM Modulation

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




The AM-modulated signal is of the form:

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

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

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

#include "dsplab.h"          // DSK initialization declarations and function prototypes
#include <math.h>
//#define PI 3.141592653589793

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;
  float PI = 4*atan(1);

  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,yR);

  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:

$$f = 200 \text{ Hz}, \quad f_{\text{env}} = 100 \text{ Hz}$$
$$f = 200 \text{ Hz}, \quad f_{\text{env}} = 190 \text{ Hz}$$
$$f = 200 \text{ Hz}, \quad f_{\text{env}} = 200 \text{ Hz}$$

## 2.5. FM Modulation

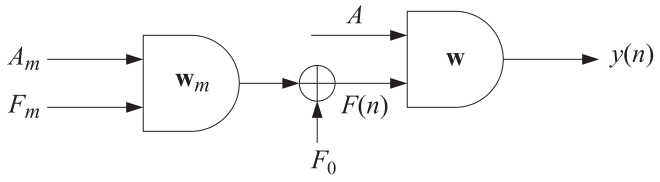
The third program, `fmex.c`, illustrates FM modulation in which the frequency of a sinusoid is time-varying. The generated signal is of the form:

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

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

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

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



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

#include "dsplab.h"          // DSK initialization declarations and function prototypes
#include <math.h>
//#define PI 3.141592653589793

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;
  float PI = 4*atan(1);

  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,yR);

  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

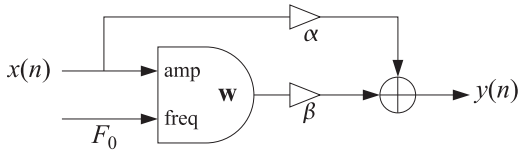
will hear a square wave whose frequency switches between a high and a low value in each second.

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:

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

## 2.6. 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:

$$y(n) = \alpha x(n) + \beta x(n)\cos(2\pi F_0 n) = x(n)\left[\alpha + \beta\cos(2\pi F_0 n)\right] \tag{2.2}$$

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

$$y(n) = x(n)\cos(2\pi F_0 n) \tag{2.3}$$

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

$$y(n) = x(n) + \beta x(n)\cos(2\pi F_0 n) = x(n)\left[1 + \beta\cos(2\pi F_0 n)\right] \tag{2.4}$$

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,yR);

  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.

## 2.7. Scrambler as Ring Modulator

In the frequency domain, Eq. (2.3) is equivalent to frequency translation:

$$Y(f) = \frac{1}{2}\Big[X(f - f_0) + X(f + f_0)\Big] \tag{2.5}$$

As $f_0$ is chosen closer and closer to the Nyquist frequency $f_s/2$, the shifted replicas begin to resemble the inverted spectrum of $X(f)$. In particular, if $f_0 = f_s/2$, then,

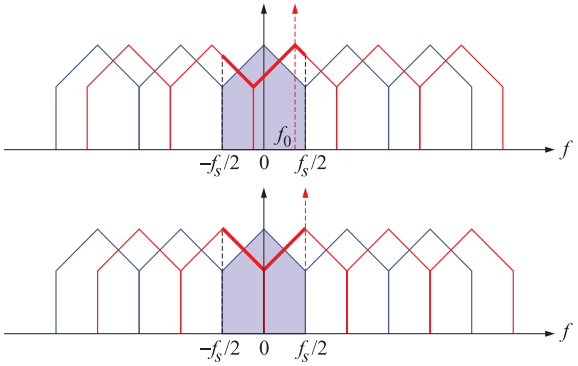
$$Y(f) = \frac{1}{2}\Big[X(f - f_s/2) + X(f + f_s/2)\Big]$$

Using the periodicity property $X(f \pm f_s) = X(f)$, we then obtain the equivalent expressions:

$$Y(f) = \frac{1}{2}\Big[X(f - f_s/2) + X(f + f_s/2 - f_s)\Big] = X(f - f_s/2)\,, \quad 0 \le f \le \frac{f_s}{2}$$

$$Y(f) = \frac{1}{2}\Big[X(f - f_s/2 + f_s) + X(f + f_s/2)\Big] = X(f + f_s/2)\,, \quad -\frac{f_s}{2} \le f \le 0$$

which imply that the positive (negative) frequency part of $Y(f)$ is equal to the negative (positive) frequency part of $X(f)$, in other words, $Y(f)$ is the inverted version of $X(f)$. This is depicted below.




Because in this case $F_0 = f_0/f_s = (f_s/2)/f_s = 1/2$, the carrier waveform is simply the alternating sequence of $\pm 1$:

$$\cos(2\pi F_0 n) = \cos(\pi n) = (-1)^n$$

and the modulator output becomes

$$y(n) = (-1)^n x(n) \tag{2.6}$$

### Lab Procedure

Modify the `template.c` program to implement the frequency-inversion or scrambling operation of Eq. (2.6). This can be done easily by introducing a global index:

```
int q = 1;
```

and keep changing its sign at each interrupt call, i.e., after reading the left/right codec inputs, define the corresponding codec outputs by:

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

q = -q;
```

Compile and run this program. Send the wave file `JB.wav` into it. First comment out the line `q = -q`, and hear the file as pass through. Then, enable the line, recompile, and hear the scrambled version of the file.

The scrambled version was recorded with MATLAB and saved into another wave file, `JBm.wav`. If you play that through the scrambler program, it will get unscrambled. In Labs 3 & 4, we will implement the frequency inversion in alternative ways.

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

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

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

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

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