**ECEN4002/5002   Digital Signal Processing Laboratory**

Spring 2002

# Laboratory Exercise #5

## *Signal Synthesis*

## Introduction

Up to this point we have been developing and implementing signal *processing* algorithms: we start with an analog input signal, sample it, perform some LTI process, then reconstruct an analog signal. Another type of DSP application involves signal *synthesis* algorithms: the process calculates the output signal from scratch, perhaps with some low bandwidth control signals.

Signal synthesis is useful in quite a few DSP applications. Examples include communications systems, test and simulation equipment, music and entertainment, and so forth.

In this experiment you will work on two different types of synthesis routines. The first type involves generating the output signal directly from a mathematical equation or sequence of discrete-time operations. There are a large number of numerical procedures that have been developed over the years that are suitable for use in the DSP environment, but we will be considering only a simple random number generator. The second type of synthesis involves the use of a stored waveform—a look-up table, or *wavetable*—that is read over and over to create a periodic output signal.

## Algorithmic signal synthesis

A pseudo-random sequence is a binary sequence produced by a finite state machine (FSM). Since the state machine is finite, its output is actually periodic. However, the period can be so large that it is not detectable. By carefully selecting the characteristics of the pseudo-random algorithm it is often possible to achieve a sequence that goes through every possible combination of $n$ bits ($2^n$-1). Such a cycle is called a "maximum length" pseudo-random sequence, and looks statistically very much like a random process.

A computationally cheap method for generating a pseudo-random sequence is the *linear congruential* method. The computation is expressed as

$$y[n]=(a\cdot y[n-1] + c)_{\text{mod } M},$$

where $a$ is a constant multiplier, $c$ is a constant offset, and $M$ is the modulus. The initial value to start the sequence, y[0], is called the *seed* of the random generator. The choice of $a$, $c$, and $M$ must be made with care to ensure that a maximal length output sequence is obtained for any choice of the seed, y[0]. Note that if $c$ is chosen to be zero, then the seed must not also be zero. It is common to choose $M$ to be equal to the word length of the processor (e.g., 24 bits for the 563xx) so that the modulo operation is achieved simply by taking the appropriate portion of the accumulator.

It is important to keep in mind that a linear congruential generator is a deterministic algorithm, and the sequence of output values is exactly determined from the seed and the constant parameters. It is often found that the bit patterns are correlated from one output to the next, especially the least-significant bits. This means that if you try to simulate the outcome of a coin flip by testing the LSB of the random word, the result may have a periodic pattern of some kind rather than a more random distribution. The moral of this story is to be extremely skeptical of simple algorithmic random number generators!

## ⇨Exercise A: Construct a uniformly distributed random number generator

Starting with a nonzero seed number in the A accumulator, do an *integer* multiply by a constant *a* (need to do a right shift after the multiply to convert from fractional to integer form), then throw away the most significant part of the result by moving A0 into A. Note that we are taking *c* to be zero, but you can also try this with a non-zero *c* value. If the constant *a* is 'good' then the iteration of this scheme will produce a reasonable approximation to a random sequence, which is uniform on the interval $-1 < y < 1$. What is the theoretical mean and variance of this sequence? You will have to experiment with the choice of the constant *a*. For some values, the result will be noticeably nonwhite (pitched or repetitive) to the ear. You might also choose to do some reading in a numerical methods book to find suggestions for good *a* values. For your report, include an example of a ‘bad’ value of *a* and a seed that results in a short repetitive sequence, and a ‘good’ value of *a* and a seed that gives a hissy, white-sounding sequence.

For the ‘good’ value of *a*, obtain several lon g segments of output values using the file I/O method of the debugger (or the simulator), and include in your report a histogram and empirical statistics obtained for a ‘good’ value of *a*. Do the results appear to be random in a practical sense?

*Optional*: If you have time, try running your noise generator through one of the filters (FIR, IIR, EQ, etc.) you wrote for the previous lab assignments. Modify one of your pass programs so that you ignore the analog input signal and just generate the noise samples as the input to your filter routine.

**Additional exercise for Graduate Students:**
The *central limit theorem* states that the sum of independent random variables tends toward a Gaussian distribution as the size of the sum increases. Try using your uniform random number generator to simulate white Gaussian noise (WGN). Take the sum of 12 consecutive values of your uniform random generator and normalize to get mean 0 and variance 0.0625 (the standard deviation is 0.25). This will form one output value of your pseudo WGN generator. In other words, for every output sample you must generate 12 values of the pseudo uniform random sequence. Include in your report a histogram of your generated WGN, and comment on the results.

## Wavetable synthesis of periodic signals

Sinusoids are conceptually the opposite in complexity to white noise signals. White noise is perfectly unpredictable, by definition, and has the maximum amount of complexity. Sinusoids, on the other hand, are perfectly predictable given just a few sample values. Musically interesting signals lie somewhere between these two extremes. Musical signals are often predictable over short intervals of time, but they must be able to surprise the listener to some extent in order to remain intriguing and interesting.

The basic building block for periodic signal synthesis is the *wavetable* synthesizer. The shape of a desired waveform is stored in memory (a "wavetable") so that the samples in the table can be extracted and sent to the DAC to produce the desired sound. The wavetable can be filled with samples recorded from a natural sound (a process known as "sampling" to electronic musicians) or with arbitrary pre-computed samples.

Naturally, there are complications. When the samples are obtained by recording natural instruments, there must be a way to control the playback duration independently of its recorded duration. Also, it is usually necessary to make recordings of the original sound at many different pitches and volumes to give the synthesizer a comparable range of expressiveness. The timbre of an instrument changes with loudness in ways not recreated by simple scaling of its waveform. However, it is not practical to record a different sample for each desired pitch because the memory requirements would be prohibitive. Furthermore, such an approach would make "pitch bending" and vibrato impossible. The basic problem is that the waveform in the wavetable corresponds to exactly one pitch.

A simple technique for transposing the pitch is to skip samples. Reading every other sample will have the effect of doubling the frequency; reading every third will triple it, and so on. Skipping an integral number of samples represents a crude decimation. The effective *phase counter*, which keeps track of the current phase (look-up location) of the synthesized waveform, increments by any integer value rather than just 1. The limitation of this crude technique is that a doubling of the frequency--the result of applying the smallest decimation factor--represents transposition by an octave. What do you do if you need an intermediate musical pitch? The answer is to use a *fractional* phase counter and more precise interpolation.

Frequency control of a wavetable synthesizer involves a phase look-up accumulator (LUA) and a sample increment (SI), both with an integer part and a fractional part. This is depicted in Figure 1. When the fractional part of the LUA is zero, the synthesizer simply reads the specified sample out of the wavetable. A nonzero fractional part of the LUA indicates that we need a sample in between two of the samples actually stored in the wavetable. It should be clear that what we need is to perform sample rate conversion on the stored wavetable: we want to re-sample the stored waveform so that the amount of time it takes to play one cycle of the signal is more (lower frequency) or less (higher frequency) than the original samples in the wavetable.

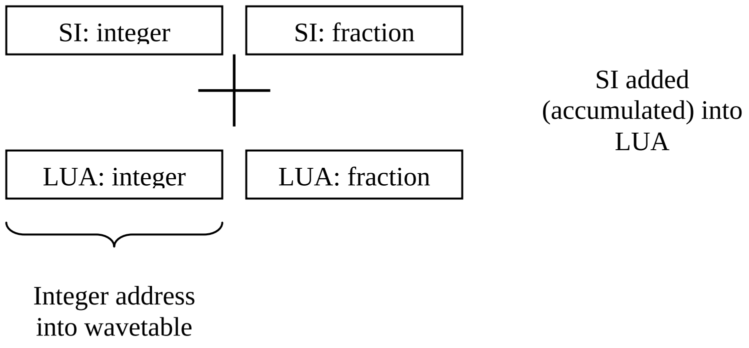
SI: integer
SI: fraction
SI added (accumulated) into LUA
LUA: integer
LUA: fraction
Integer address into wavetable

***Figure 1:*** *Look-Up Accumulator (phase counter) for wavetable synthesizer.*

Various interpolation algorithms can be used to handle the case of a non-zero fractional value in the LUA. The simplest method ignores the fractional part of the LUA and just reads the location specified by the integer part (simple truncation). Linear interpolation provides a superior result, but at the expense of a bit more computation. Linear interpolation means that we approximate the true value of the continuous waveform between two stored samples as a simple straight line. The real waveform is generally not exactly a straight line, so our interpolation will not be perfect. Nevertheless, the linear approach is inexpensive to compute and often quite satisfactory.

To implement linear interpolation it is necessary to read two values out of the wavetable, namely the current integer part of the LUA and the integer part plus 1. The fractional part of the LUA indicates what fraction of the way between the two samples is the desired location of the output sample. This is depicted in Figure 2. We use the equation for the line connecting the two samples in order to compute the output value.

$$y[L+fract] = y[L] + fract\cdot(y[L+1]-y[L])$$

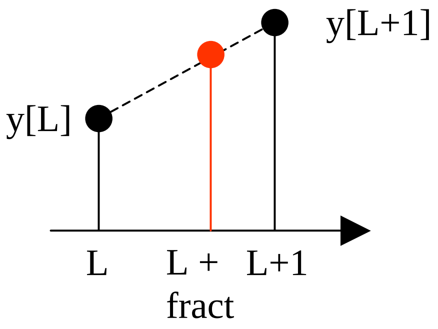

***Figure 2:*** *Linear interpolation scheme.*

The best wavetable synthesizers use even more sophisticated interpolation techniques, ultimately approaching ideal Shannon re-sampling.

Besides frequency control, we will also want amplitude control of the synthesized waveform. This means that the sample values obtained or interpolated from the wavetable will be multiplied by an amplitude scaling factor. If the amplitude scaling is made to be time varying, then the control is known as an *amplitude envelope* of the signal. The amplitude envelope gives the output signal a more natural and gradual change in loudness rather than an abrupt ‘on/off’ characteristic. The amplitude envelope can be created by using linear segments or some other functional representation, or by using another look-up table.

## ⇨Exercise B: Construct a wavetable generator

Devise a way to get one period of a sine wave (amplitude less than 1) loaded into Y memory, and then write a wavetable signal generator module that can produce an accurate sine wave in the frequency range from near zero to at least 3 kHz. The technique must use linear interpolation between the samples loaded into the sine wave data buffer, and run at an output sampling frequency of 48 kHz. Your LUA and SI values should have 24-bits in the fractional part and up to 24 bits in the integer part, so think about using A0 for the fractional part and A1 for the integer part. After adding the SI to the LUA you will need to test whether the look-up address (integer part of the LUA) has gone beyond the end of the wavetable and “wrap” it accordingly while keeping the fractional part intact.

Use the location $700 in X memory to store the frequency in hertz as an integer. In other words, if you are asked to produce a 1 kHz sine wave, you will enter the value 1000 decimal in this location and then start your program. You will need to do some computation on the contents of X:$700 in the initialization part of your code in order to get the proper integer part and fractional part of the sample increment (SI) for your circular buffer manager routines.

You probably want to create the sine wave table for this exercise using an “include” file organized as a series of “`dc`” assembler statements—much like what you did for the FIR and IIR filter coefficients in the previous lab exercises. Consider an appropriate size for your sine wave table in order to get good interpolation quality with minimal aliasing.

After the sine wave generator is working, make a different wave table signal that contains a more complicated—but still harmonic—waveform. You can use Matlab or some other program to sum up a Fourier series of harmonic components, then normalize the result to fit within the +/-1 range of the DSP. How does it sound?

## ⇨Exercise C: Make a wavetable music player

**Part 1:**

Program a monophonic (one note at a time) wavetable synthesizer that can read the contents of an encoded musical 'score' and generate a musical 'tune'. Apologies in advance to all serious musicians. Your program should “play” the score and then stop automatically. The score is a simple “pitch, duration” encoding like the following:

```
; format of score is [note# ,duration], ...,[note#,duration],[-1]
;
;
fs      equ     48000
eighth  equ     fs/6
quarter equ     2*eighth
half    equ     2*quarter
whole   equ     2*half

c_      equ     0
c#_     equ     1
d_      equ     2
```

```
d#_     equ     3
        …etc…
;

        your definition of musical pitches (SI values) goes here!!
SI_int
  dc  …etc…
SI_fract
  dc  …etc…
;
scoretab
  dc      12+c_,half,b_,eighth,a_,eighth,g_,eighth,f_,eighth
  dc      e_,quarter,d_,quarter,c_,whole,-1
```

Your musical pitch definitions must use the 1939 international standard frequency values in Hz for notes in the middle octave of the piano as given in the following table. (see **Science and Music**, by Sir James Jeans, Dover reprint, pp. 22-25)

| C | C# | D | D# | E | F | F# | G | G# | A | A# | B |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 261.6 | 277.2 | 293.6 | 311.1 | 329.6 | 349.2 | 370.0 | 392.0 | 415.3 | 440.0 | 466.2 | 493.9 |

Study the way that the tones are encoded carefully so that you can determine the integer (`SI_int`) and fractional (`SI_fract`) parts of the wavetable step size for your synthesizer. If possible, have the pre-processor do the calculation at assemble time. Note that the SI values will depend on the sample rate, table length, and the desired output frequency. When your program is "playing" the score, it should read the musical pitch index and look up the proper SI value from your `SI_int` and `SI_fract` tables, then read the note duration and set up a counter to produce the desired number of output samples for that note. This process should continue until the end of the score is reached.

Find a short musical piece or compose one of your own, then encode the notes into the `scoretab` format shown above. You must be able to demonstrate that your song player can read your given score and play the music. *Optional*: How would you also handle musical "rests": silences between notes?

**Part 2:**
Now modify your wavetable program to include a simple linear amplitude envelope. At first, just have the envelope start at its maximum value and then decay linearly to zero over the specified duration of the note. *Optional*: If you have time, modify the envelope to have a linear attack segment going from zero to maximum in 100ms, then a linear decay to zero over the remainder of the note.

**Additional exercise for Graduate Students:**
Alter your wavetable music player to allow duets: at least two wavetable notes playing and mixed (added) together at the same time. You can make two different `scoretab` tables, one for each of the two musical voices. Find or compose a simple tune to demonstrate your duet player.

*Optional*: If you are very ambitious, perhaps even consider a trio (three-part) system, such as a "Row, Row, Row Your Boat" round player with looping?

## *Report and Grading Checklist*

**A: Random number generator**

Code listing for Linear Congruential routine, with comments.
Examples of "good" and "bad" values for *a* and the seed.
Histogram results from "good" output and comments on the empirical statistics.
Comments on results of filtered noise, if any.
Grad student exercise, if applicable

**B: Wavetable signal generator**

Code listing with comments for wavetable sine wave synthesizer with linear interpolation.
Results (discussion) of signal generator with more complicated harmonic waveform.

**C: Wavetable music player**

**Part 1:** Code listing with comments for wavetable music player. Fully explain your design and any choices you needed to make.
**Part 2:** Code listing with comments for wavetable synthesizer with simple amplitude envelope.
Grad student exercise, if applicable.

*Grading Guidelines (for each grade, you must also satisfy the requirements of all lower grades):*

| | |
|---|---|
| F | Anything less than what is necessary for a D. |
| D | Exercise A code listing with example *a* values you determined. |
| D+ | Exercise A empirical results. |
| C | Exercise B with clear and concise comments. |
| B- | Exercise C music player, functional but no envelope. |
| B | Exercise C music player with envelope. |
| A | Demo of wavetable music player for Prof. Maher or lab TA. |

*Grad student grades also require the additional exercises (parts A and C).*