Announcements:
- Files > projects > synthesis.pdf on Canvas
- Include citations to source material (e.g., web sites) in proposal

Curiosity: http://www.image-line.com/plugins/Synths/Harmor
Outline

- Part 1: Matlab functions and loops
- Part 2: Music synthesis via Fourier series
- Part 3: Other music synthesis techniques
- Part 4: Project 3 Q/A
Part 1: Matlab functions and loops
Matlab anonymous functions (1)

Tedious way to write a 4-note song in Matlab:

```
S = 8192;
sound(0.9*cos(2*pi*660*[0:1/S:0.5]), S); pause(0.5)
sound(0.9*cos(2*pi*880*[0:1/S:0.5]), S); pause(0.5)
sound(0.9*cos(2*pi*660*[0:1/S:0.5]), S); pause(0.5)
sound(0.9*cos(2*pi*440*[0:1/S:1.0]), S); pause(1.0)
```

fig/loop/s_loop1a.m

What is the duration of this song? ??
Why the pause commands? ??
Can we streamline this? ??
Matlab anonymous functions (2)

Using an *anonymous function* can simplify and clarify:

S = 8192;
playnote = @(f,d) sound(0.9*cos(2*pi*f*[0:1/S:d]), S);
playnote(660, 0.5); pause(0.5)
playnote(880, 0.5); pause(0.5)
playnote(660, 0.5); pause(0.5)
playnote(440, 1.0); pause(1.0)

- Anonymous functions are very useful.
- General form: `function_handle = @(arg1, arg2, ... argN)`.
- Simpler (less typing for coder)
- Easier to see the key elements of the song (notes and duration).
- Easier to make global changes (such as amplitude 0.9).
- But still tedious if the song is longer than 4 notes...
Matlab loops (1)

Using a for loop is the most concise and elegant: (fig/loop/s_loop1c.m)

S = 8192;
playnote = @(f,d) sound(0.9*cos(2*pi*f*[0:1:S:d]), S);
fs = [660 880 660 440]; % list of frequencies
ds = [0.5 0.5 0.5 1.0]; % list of durations
for index=1:numel(fs)
    playnote(fs(index), ds(index))
    pause(ds(index))
end
Matlab loops (2)

Here is another loop version that sounds better:

```matlab
S = 8192;
note = @(f,d) 0.9*cos(2*pi*f*[0:1/S:d]);
fs = [660 880 660 440];
ds = [0.5 0.5 0.5 1.0];
x = [];
for index=1:numel(fs)
    x = [x note(fs(index), ds(index))];
end
sound(x, S)
```

Loops (and functions) are ubiquitous in software.
Matlab functions

Two types of functions in Matlab:

- Anonymous functions
defined “in line” using syntax like: triple = @(x) 3 * x;

- Functions defined in separate files.

Example. If the file triple.m is in your Matlab path and has the lines

```matlab
function y = triple(x)
y = 3 * x;
```

Then you can use it like any other Matlab function, e.g.,
a = [10 15 20];
b = triple(a)

Exercise. Create a function playsong that has two inputs, a list of frequencies and a list of durations, and plays the corresponding song.

```matlab
edit playsong.m
```

Then test it: playsong(220 * [3 4 3 2 3 4 3], [1 1 1 1 1 1 2]/3)
Part 2: Music synthesis via Fourier series
Additive Synthesis: Mathematical formula

Simplified version of Fourier series for monophonic audio:

\[ x(t) = \sum_{k=1}^{K} c_k \cos \left(2\pi \frac{k}{T} t \right) \]

- No DC term for audio: \( c_0 = 0 \).
- Phase unimportant for monophonic audio, so \( \theta_k = 0 \).
- Which version is this? ??
  (Sinusoidal form? trigonometric form? complex exponential form?)

Example:

\[ x(t) = 0.5 \cos(2\pi 400t) + 0.2 \cos(2\pi 800t) + 0.1 \cos(2\pi 2000t) \]
Example: Why we might want harmonics

\[ y(t) = 0.5 \cos(2\pi 400t) \]
\[ x(t) = 0.5 \cos(2\pi 400t) + 0.2 \cos(2\pi 800t) + 0.1 \cos(2\pi 2000t) \]

% fig_why1: example of additive synthesis
S = 44100;
N = 0.5 * S; % 0.5 sec
t = [0:N-1]/S; % time samples: t = n/S
y = 0.5 * cos(2*pi * 400 * t);
x = y + 0.2 * cos(2*pi * 800 * t) + 0.1 * cos(2*pi * 2000 * t);
plot(t, x, ':-', t, y, '--'), legend('x', 'y')
xlabel 't', ylabel 'x(t), y(t)', axis([0 0.01 -1 1])

Same fundamental period, same pitch, but different timbre.
Matlab implementation: “Simple”

Example: $x(t) = 0.5 \cos(2\pi 400t) + 0.2 \cos(2\pi 800t) + 0.1 \cos(2\pi 2000t)$

- A simple Matlab version looks a lot like the mathematical formula:

  ```matlab
  S = 44100;
  N = 0.5 * S; % 0.5 sec
  t = [0:N-1]/S; % time samples: t = n/S
  x = 0.5 * cos(2 * pi * 400 * t) ... 
  + 0.2 * cos(2 * pi * 800 * t) ... 
  + 0.1 * cos(2 * pi * 2000 * t);
  ```

- There are many “hidden” for loops above. Where? ??

- Matlab saves us from the tedium of writing out those loops, thereby making the syntax look more like the math.

- In “traditional” programming languages like C, one would have to code all those loops.

- This “simple” implementation is still somewhat tedious, particularly for signals having many harmonics.
#include <math.h>

void my_signal(void)
{
    float S = 44100;
    int N = 0.5 * S; // 0.5 sec
    float x[N]; // signal samples
    for (int n=0; n < N; ++n)
    {
        float t = n / S;
        x[n] = 0.5 * cos(2 * M_PI * 400 * t)
            + 0.2 * cos(2 * M_PI * 800 * t)
            + 0.1 * cos(2 * M_PI * 2000 * t);
    }
}
Many dozens of harmonics needed to get a “good” square wave approximation.
Matlab implementation: Loop over harmonics

Example: \( x(t) = 0.5 \cos(2\pi 400t) + 0.2 \cos(2\pi 800t) + 0.1 \cos(2\pi 2000t) \)

\[
S = 44100; \\
N = 0.5 * S; \% 0.5 sec \\
t = [0:N-1]/S; \% time samples: t = n/S \\
c = [0.5 0.2 0.1]; \% amplitudes \\
f = [1 2 5] * 400; \% frequencies \\
x = 0; \\
for k=1:length(c) \\
    x = x + c(k) * \cos(2 * \pi * f(k) * t); \\
end
\]

- I think this version is the easiest to read and debug.
- It looks the most like the Fourier series formula: \( x(t) = \sum_{k=1}^{K} c_k \cos \left( 2\pi \frac{k}{T} t \right) \).
- In fact it is a slight generalization.
  - In Fourier series, the frequencies are multiples: \( k/T \).
  - In this code, the frequencies can be any values we put in the \( f \) array.
Example: Square wave via loop, with sin

\[ S = 44100; \]
\[ N = 0.5 \times S; \quad \% \quad 0.5 \text{ sec} \]
\[ t = [0:N-1]/S; \quad \% \text{ time samples: } t = n/S \]
\[ c = 1 ./ [1:2:15]; \quad \% \text{ amplitudes} \]
\[ f = [1:2:15] \times 494; \quad \% \text{ frequencies} \]
\[ x = 0; \]
\[ \text{for } k=1:\text{length}(c) \]
\[ \quad x = x + c(k) \times \sin(2 \times \pi \times f(k) \times t); \]
\[ \text{end} \]

How many harmonics in this example? ??

Music example, circa 1978:
Example: Square wave via loop, with cos

\[ S = 44100; \]
\[ N = 0.5 * S; \% \text{ 0.5 sec} \]
\[ t = [0:N-1]/S; \% \text{ time samples: } t = n/S \]
\[ c = 1 ./ [1:2:15]; \% \text{ amplitudes} \]
\[ f = [1:2:15] * 494; \% \text{ frequencies} \]
\[ x = 0; \]
\[ \text{for } k=1:\text{length}(c) \]
\[ \quad x = x + c(k) * \cos(2 * \pi * f(k) * t); \]
\[ \text{end} \]
Matlab implementation: Concise

We can avoid writing any explicit for loops (and reduce typing) by using the following more concise (i.e., tricky) Matlab version:

```
S = 44100;
N = 0.5 * S; % 0.5 sec
t = [0:N-1]/S; % time samples: t = n/S
c = [0.5 0.2 0.1]; % amplitudes
f = [1 2 5] * 400; % frequencies
z = cos(2 * pi * f' * t);
x = c * z;
```

c is $1 \times 3$
f' is $3 \times 1$
t is $1 \times N$

z is ??
x is ??

Where are the (hidden) loops in this version? ??

Use this approach or the previous slide in Project 3 synthesizers.
Part 3: Other music synthesis techniques
Frequency Modulation (FM) synthesis

In 1973, John Chowning of Stanford invented the use of frequency modulation (FM) as a technique for musical sound synthesis:

\[ x(t) = A \sin(2\pi ft + I \sin(2\pi gt)), \]

where \( I \) is the modulation index and \( f \) and \( g \) are both frequencies. (Yamaha licensed the patent for synthesizers and Stanford made out well.)

This is a simple way to generate periodic signals that are rich in harmonics. However, finding the value of \( I \) that gives a desired effect requires experimentation.
FM example 1: Traditional

\[ S = 44100; \]
\[ N = 1.0 \times S; \]
\[ t = [0:N-1]/S; \quad \text{% time samples: } t = n/S \]
\[ I = 7; \quad \text{% adjustable} \]
\[ x = 0.9 \times \sin(2\pi\times400\times t + I \times \sin(2\pi\times400\times t)); \]

Very simple implementation (both in analog and digital hardware), yet can produce harmonically very rich spectra:

![Spectrum of FM signal with I=7](image-url)
**FM example 2: Time-varying**

Time-varying modulation index: \( x(t) = A \sin \left( 2\pi ft + I(t) \sin(2\pi gt) \right) \).

Simple formula / implementation can make remarkably intriguing sounds.

```plaintext
S = 44100;
N = 1.0 * S;
t = [0:N-1]/S; % time samples: t = n/S
I = 0 + 9*t/max(t); % slowly increase modulation index
x = 0.9 * sin(2*pi*400*t + I .* sin(2*pi*400*t));
```

Besides making the modulation index \( I \) a vector, how else did the code change? ??

Previous code for reference:

```plaintext
S = 44100;
N = 1.0 * S;
t = [0:N-1]/S; % time samples: t = n/S
I = 7; % adjustable
x = 0.9 * sin(2*pi*400*t + I * sin(2*pi*400*t));
```

What is the most informative graphical representation? ??
Illustrations of previous FM signal

Spectrogram of FM signal with time-varying modulation

First 0.05 seconds

Last 0.02 seconds
Nonlinearities

Another way to make signals that are rich in harmonics is to use a nonlinear function such as $y(t) = x^9(t)$.

S = 44100;
N = 1.0 * S;
t = [0:N-1]/S; % time samples: t = n/S
x = cos(2*pi*400*t);
y = x.^9;
Nonlinearities in amplifiers

- High quality audio amplifiers are designed to be very close to linear because any nonlinearity will introduce undesired harmonics (see previous slide).
- Quality amplifiers have a specified maximum *total harmonic distortion* (THD) that quantifies the relative power in the output harmonics for a pure sinusoidal input.

\[
\cos(2\pi ft) \rightarrow \text{Amplifier} \rightarrow c_1 \cos(2\pi ft) + c_2 \cos(2\pi 2ft) + c_3 \cos(2\pi 3ft) + \cdots
\]

- A formula for THD is: [wiki]

\[
\text{THD} = \frac{c_2^2 + c_3^2 + c_4^2 + \cdots}{c_1^2} \cdot 100\%
\]

- What is the best possible value for THD? ?

- On the other hand, electric guitarists often deliberately operate their amplifiers nonlinearly to induce distortion, thereby introducing more harmonics than produced by a simple vibrating string.
  - pure: play
  - distorted: play

25
Envelope of a musical sound
Envelope example: Train whistle

![Train whistle signal](image1)

![Train whistle envelope](image2)

**Attack**

**Release**
Envelope example: Plucked guitar

plucked guitar signal

\[ x(t) \]

plucked guitar envelope

\[ \text{envelope}(t) \]
Envelope implementation

\[
S = 44100; \\
N = 1 * S; \\
t = [0:N-1]/S; \\
c = 1 ./ [1:2:15]; \% amplitudes \\
f = [1:2:15] * 494; \% frequencies \\
x = 0; \\
for k=1:length(c) \\
    x = x + c(k) * \sin(2 * \pi * f(k) * t); \\
end \\
env = (1 - \exp(-80*t)) .* \exp(-3*t); \% fast attack; slow decay \\
y = env .* x;
\]

![Graph of x(t) and y(t)]
Attack, Decay, Sustain, Release (ADSR)

Programmable music synthesizers usually allow the user to control separately the time durations of these 4 components of the envelope.

- For synthesizers with a keyboard, the “sustain” portion lasts as long as the key is pressed.
- The “release” portion occurs after the key is released.
- In synthesizers with “touch control” the properties of the “attack” and “decay” portions may depend on how hard/fast one presses the key.
Summary

● There are numerous methods for musical sound synthesis
● Additive synthesis provides complete control of spectrum
● Other synthesis methods provide rich spectra with simple operations (FM, nonlinearities)
● Time-varying spectra can be particularly intriguing
● Signal envelope (time varying amplitude) also affects sound characteristics
● Ample room for creativity and originality!
Part 4: Project 3 Q/A?