Eng. 100: Music Signal Processing

DSP Lecture 10

Music synthesis: Advanced methods

Announcements:

- Final Exam: Tue. Dec. 16, 1:30-3:30 PM, 1017 + 1018 Dow
- CoE Graduate Research Symposium: Friday (Nov. 14)
  - ECE posters in EECS atrium
  - Signal processing posters 10:45 AM - 12:45 PM
  - http://gradsymposium.engin.umich.edu

Outline

- Part 1. Advanced music synthesis methods
  - Amplitude variations
    - Envelope (done already last lecture)
    - Attack, Decay, Sustain, Release (ADSR)
    - Tremolo
  - Frequency / spectrum variations
    - Vibrato
    - Glissando
- Part 2. Project 3 logistics and Q/A
- Part 3. Project 3 tips
  - `reshape`, `function`
  - Transcriber hints: note durations
- Part 4. DSP application: Beats per minute

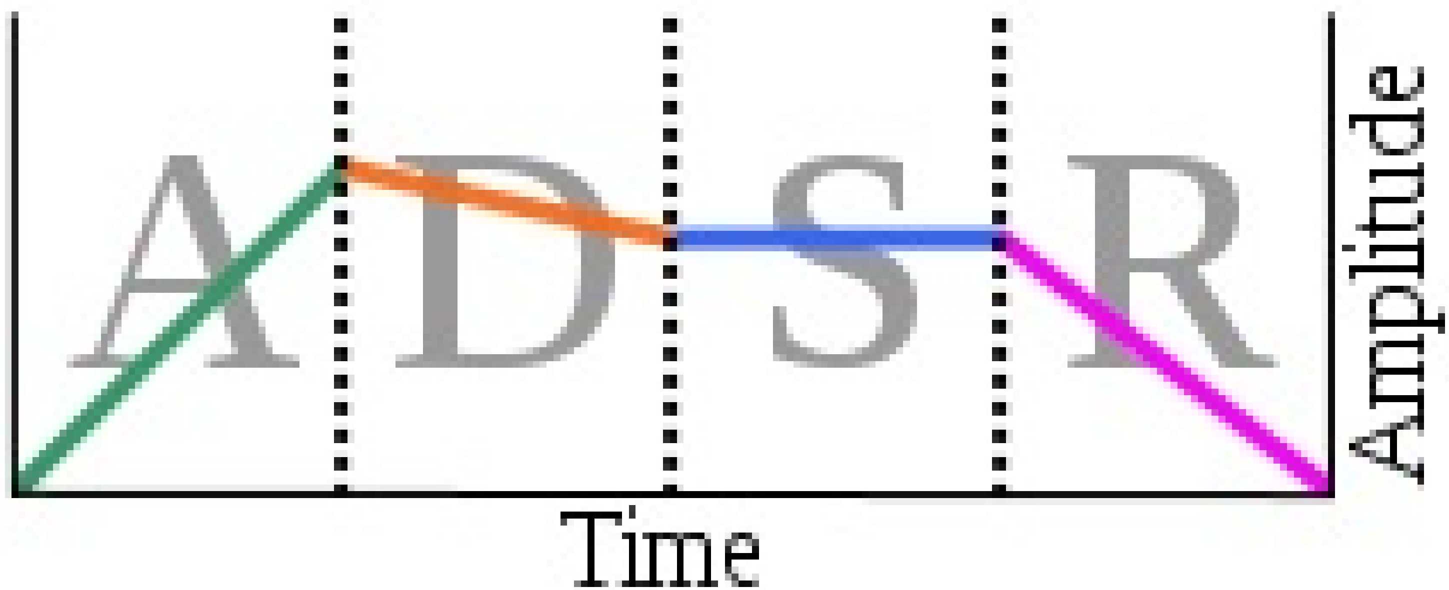
Part 1. Advanced music synthesis methods

Amplitude variations: envelope and tremolo

Attack, Decay, Sustain, Release (ADSR)

Many synthesizers let user control envelope using 4 variables:

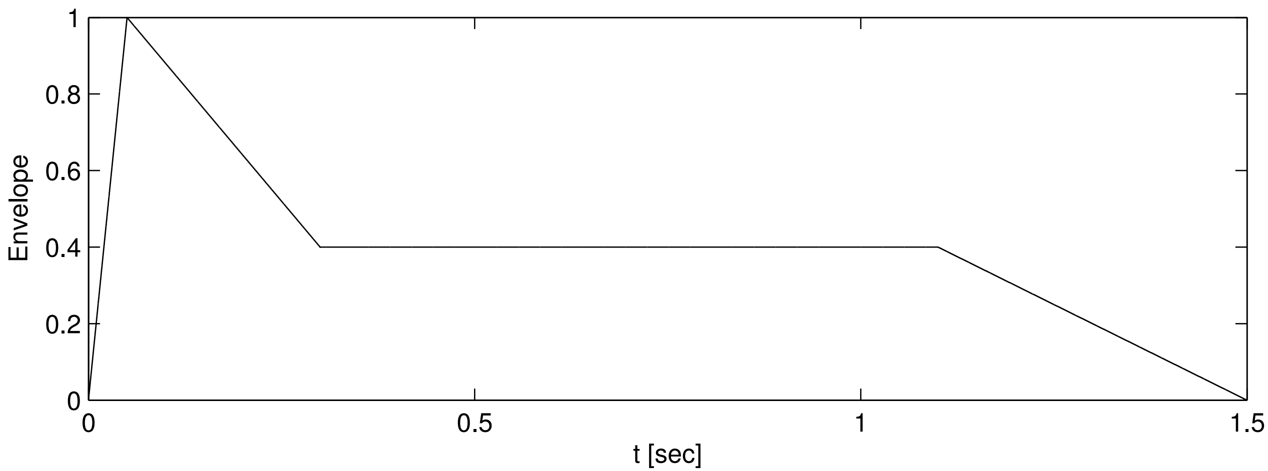
- Attack: initial rise
- Decay: initial fall
- Sustain: while key is held (or sustain pedal is pressed)
- Release: after key (or pedal) is released



http://en.wikipedia.org/wiki/ADSR_envelope

Example ADSR implementation in Matlab

```
S = 44100;
N = 1.5 * S;
t = [0:N-1]/S;
c = 1 ./ [1:2:15]; % amplitudes
f = [1:2:15] * 494; % frequencies
x = c * sin(2 * pi * f' * t); % fourier synthesis
env = interp1([0 0.05 0.3 1.1 1.5], [0 1 0.4 0.4 0], t); % !!
subplot(211), plot(t, env, '-o'), xlabel 't [sec]', ylabel 'Envelope'
y = env .* x;
sound(y, S), % wavwrite(y, S, 8, 'fig_adsr1.wav'), savefig cw fig_adsr1
```

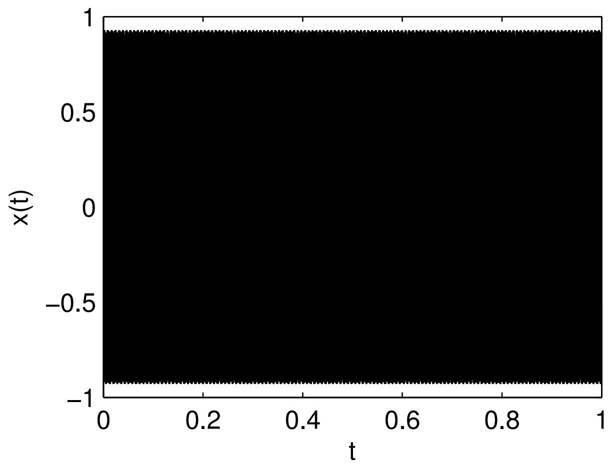


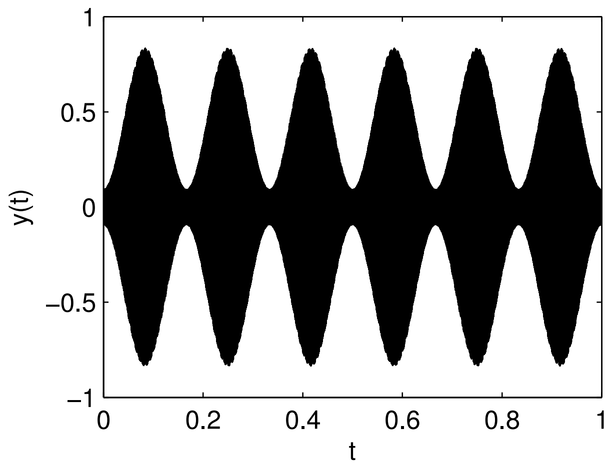
play

Tremolo

Tremolo implementation: LFO

```
S = 44100;
N = 1 * S;
t = [0:N-1]/S;
c = 1 ./ [1:2:15]; % amplitudes
f = [1:2:15] * 494; % frequencies
x = c * sin(2 * pi * f' * t); % concise way
lfo = 0.5 - 0.4 * cos(2*pi*6*t); % what frequency?
y = lfo .* x;
```



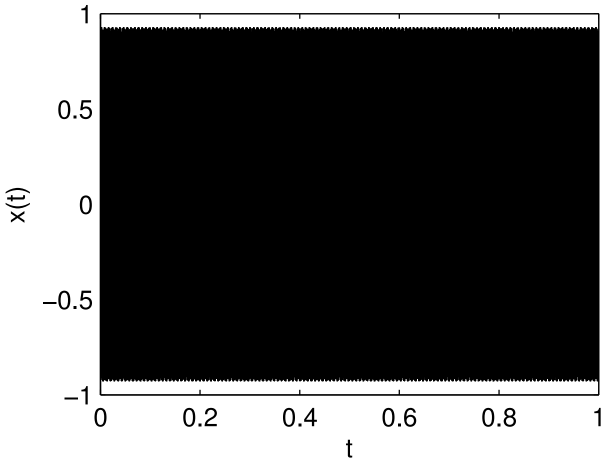


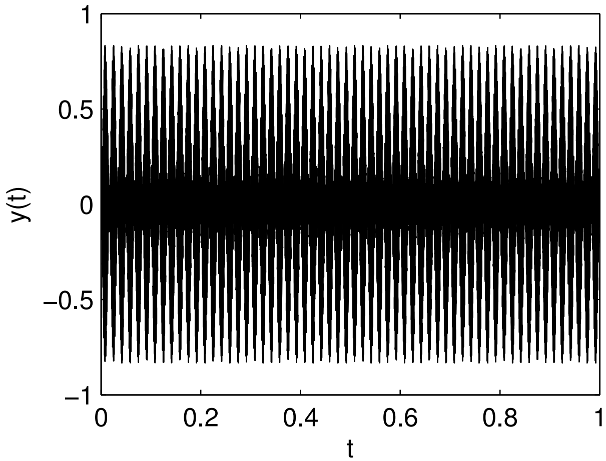
play

play

Tremolo: Why LFO?

```
S = 44100;
N = 1 * S;
t = [0:N-1]/S;
c = 1 ./ [1:2:15]; % amplitudes
f = [1:2:15] * 494; % frequencies
x = c * sin(2 * pi * f' * t); % concise way
lfo = 0.5 - 0.4 * cos(2*pi*60*t); % what frequency now?
y = lfo .* x;
```





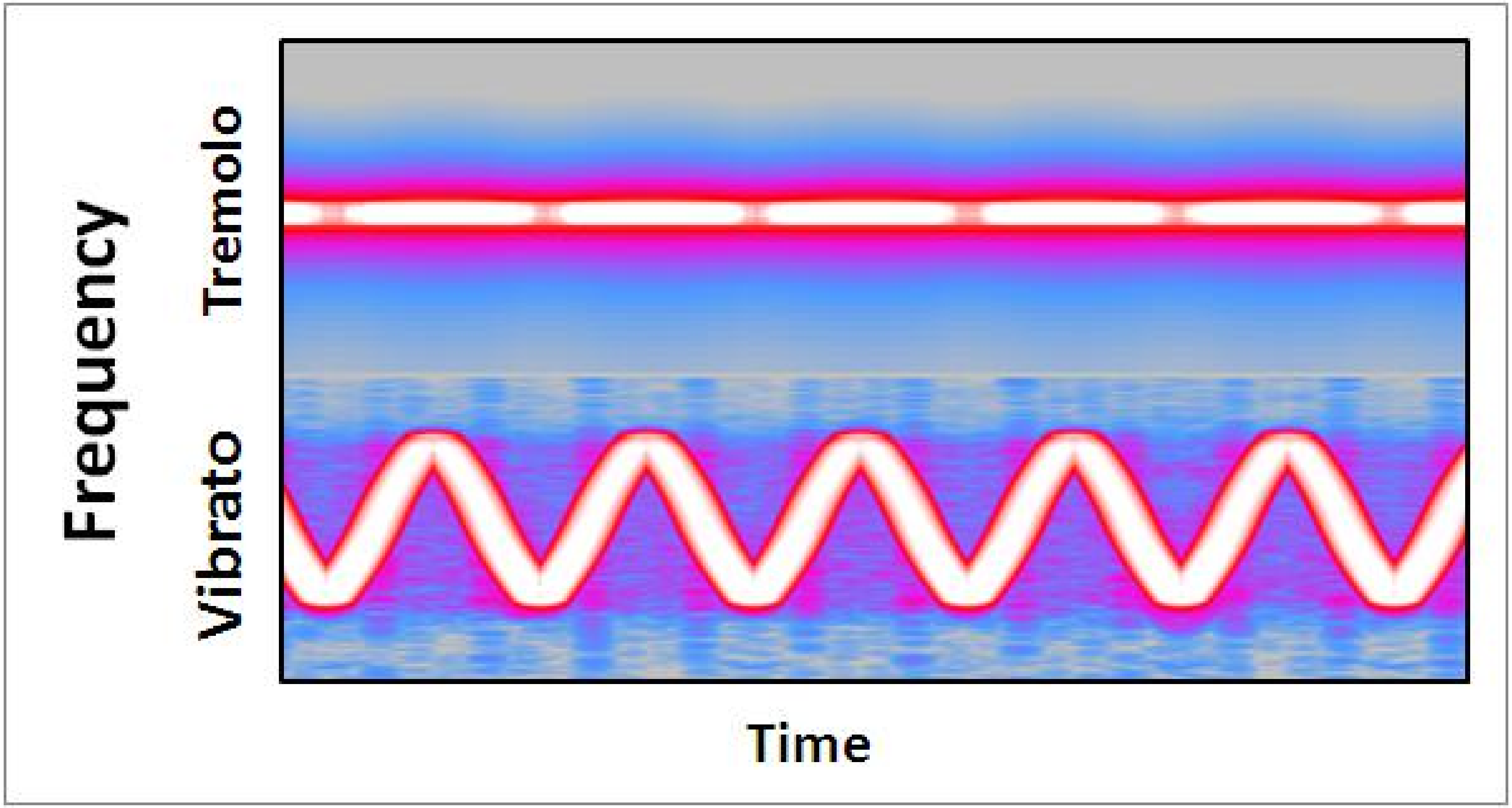
play

play

Frequency variations: vibrato and glissando

Vibrato

Vibrato vs Tremolo: Spectrograms



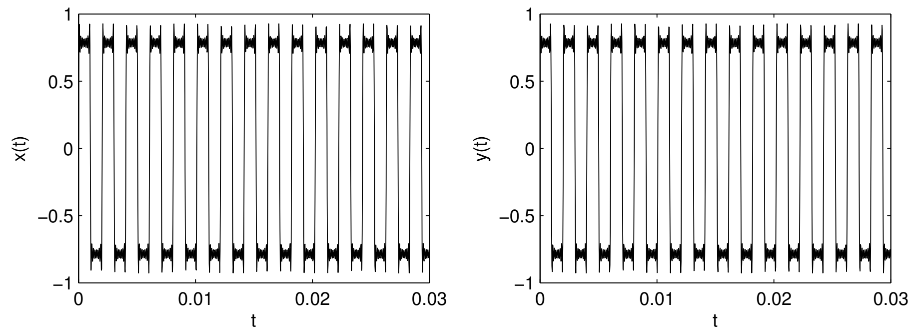
http://en.wikipedia.org/wiki/File:Vibrato_and_tremolo_graph.PNG

A Leslie speaker in a Hammond organ has both:

http://en.wikipedia.org/wiki/Leslie_speaker

Vibrato implementation: LFO

```
S = 44100;
N = 2 * S;
t = [0:N-1]/S;
c = 1 ./ [1:2:15]; % amplitudes
f = [1:2:15] * 494; % frequencies
x = 0; y = 0;
lfo = 0.001 * cos(2*pi*4*t) / 4; % about 0.1% pitch variation
for k=1:length(c)
        fnew = f(k) * lfo;
        x = x + c(k) * sin(2 * pi * f(k) * t);
        y = y + c(k) * sin(2 * pi * f(k) * t + f(k) * lfo);
end
```



play

play

Vibrato: Why LFO?

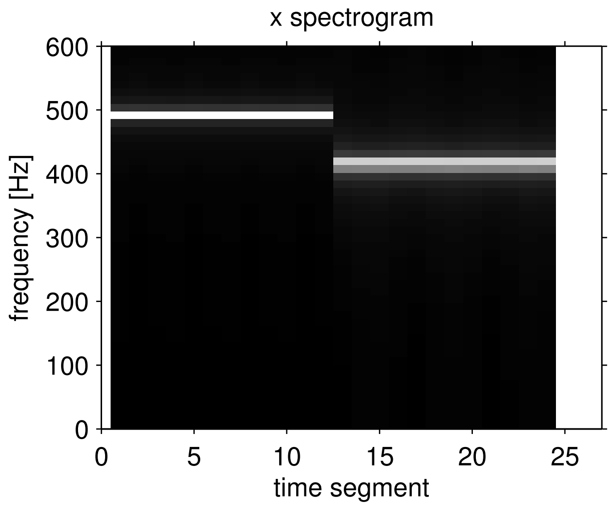
(try more; you may not like it)

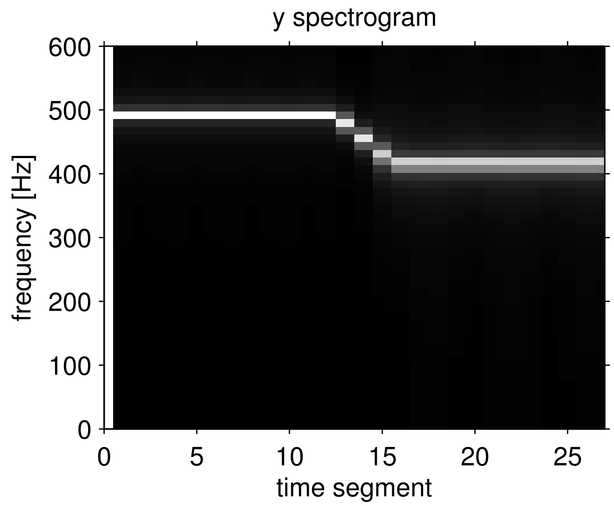
Note: FM synthesis is like an extreme form of vibrato

Glissando

Glissando implementation

```
S = 44100;
N = 1 * S;
t = [0:N-1]/S;
f = [1 2^(-3/12)] * 494; % frequencies: B G#
x = 0.9 * [cos(2*pi * f(1) * t), cos(2*pi * f(2) * t)];
tau = 0.25; % length of glissando
t2 = [0:(tau*S)-1]/S;
gliss = cos(2*pi* (f(1) * t2 + t2.^2/tau/2*(f(2) - f(1))));
y = 0.9 * [cos(2*pi * f(1) * t), gliss, cos(2*pi * f(2) * t)];
```





play

play

Frequency variations: Theory

(This page requires calculus and is entirely optional.)

For a (standard) sinusoid: $x(t) = \cos(2\pi f t) \Longrightarrow$

$$\frac{\mathrm{d}}{\mathrm{d}t} x(t) = -\sin(2\pi f t)\, 2\pi \underbrace{f}_{\text{frequency}}$$

For a sinusoid with time-varying phase: $x(t) = \cos(\phi(t)) \Longrightarrow$

$$\frac{\mathrm{d}}{\mathrm{d}t} x(t) = -\sin(\phi(t))\, 2\pi \underbrace{\frac{1}{2\pi} \frac{\mathrm{d}}{\mathrm{d}t} \phi(t)}_{\substack{\text{instantaneous} \\ \text{frequency}}}$$

Example: Glissando

Example. If we want a signal with instantaneous frequency

$$f(t) = f_1 + \frac{t}{\tau}(f_2 - f_1),$$

then we need

$$\begin{aligned}\phi(t) &= 2\pi \int_0^t f(t')\,\mathrm{d}t' = 2\pi \int_0^t \left[f_1 + \frac{t'}{\tau}(f_2 - f_1)\right]\mathrm{d}t' \\ &= 2\pi f_1 t + 2\pi \frac{t^2}{2\tau}(f_2 - f_1),\end{aligned}$$

so that

$$\frac{1}{2\pi}\frac{\mathrm{d}}{\mathrm{d}t}\phi(t) = f_1 + \frac{t}{\tau}(f_2 - f_1).$$

So the desired signal is

$$x(t) = \cos(\phi(t)) = \cos\left(2\pi f_1 t + 2\pi \frac{t^2}{2\tau}(f_2 - f_1)\right).$$

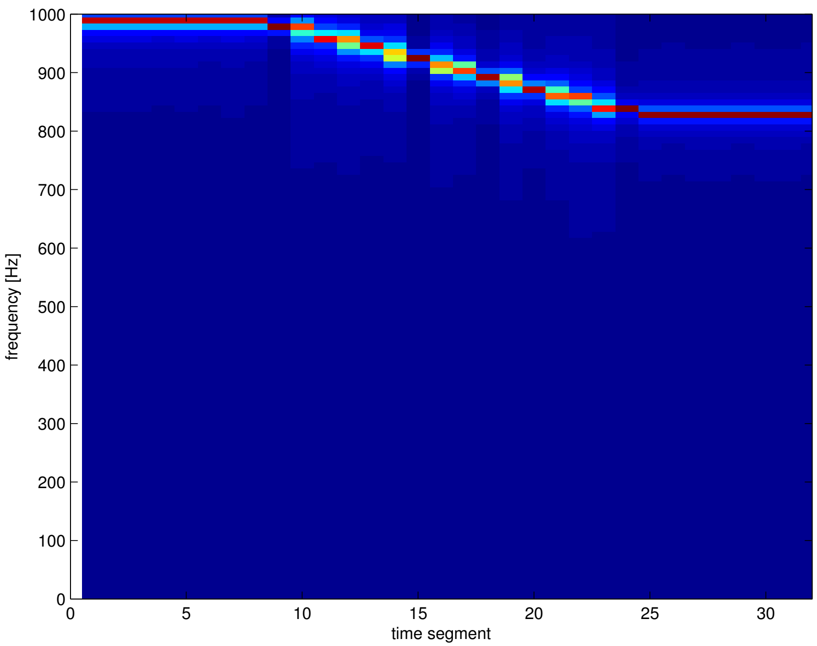
See `Matlab` example on earlier slide.

Vibrato combined with glissando

```
% fig_gliss2.m glissando + vibrato

S = 44100;
N = 2^15;
t1 = [0:N-1]/S;
f = 2 * [1 2^(-3/12)] * 494; % frequencies: B G#
x = 0.9 * [cos(2*pi * f(1) * t1), cos(2*pi * f(2) * t1)];
Ngliss = 2^16; tau = Ngliss / S; % length of glissando
t2 = [0:Ngliss-1]/S;
nvibe = 9;
phi = 2 * pi * (f(1) * t2 + t2.^2/tau/2*(f(2) - f(1)) ...
        + 5 * tau/nvibe/2/pi*cos(2*pi*nvibe/tau*t2));
x = 0.9 * [cos(2*pi * f(1) * t1), cos(phi), cos(2*pi * f(2) * t1)];
M = 2^12;
y = reshape(x, M, []);
imagesc(1:size(y,2), [0:M-1]/M*S, 2/M*abs(fft(y)))
axis xy, axis([0 size(y,2) 0 2*500])
xlabel 'time segment', ylabel 'frequency [Hz]'
sound(x, S), % wavwrite(x, S, 8, 'fig_gliss2.wav'), savefig cw fig
```

play



Other advanced synthesis methods: sound reversal, modeling, sampling, ...

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 2. Project 3 logistics and Q/A

P3 logistics

- Each presentation will use *two* computers (*e.g.*, laptops): one for the slides and one for the live demonstration.
- Teams where a member has `Matlab` on their laptop should use that laptop for the demo and another laptop for the slides.
- It may be possible to copy `Matlab` from a CAEN machine to your laptop; such a copy will run only when you are on the UM network.
- Teams without `Matlab` should arrange to meet with Prof. Fessler to upload their code onto his laptop (at latest) the day before their presentation.
- Do not use `Matlab` remote desktop for live demo: too risky!

Questions?

P3 presentation schedule

Lecture period is 80 min; $80 = 5 \times 16$

Each team should prepare 10 minutes of "talking"
plus 3 minutes of integrated "live demonstration"
Total presentation should not exceed 13 minutes!

3 minutes of Q/A and transition
(next group sets up demo while previous group answers questions)

Tue Dec 2 lecture: 5 teams
Thu Dec 4 lecture: 5 teams
Thu Dec 4 lab: 2 teams from Thu Lab

First presentation will begin exactly at 10:40AM;
come at 10:30AM to set up.

All students must attend all presentations during Tue/Thu lectures.

Part 3. Project 3 tips

Using reshape to simplify indexing

Given a vector `x` with 3 instruments each playing 12 notes, with `N = 1000` samples per note.

How do we access the 4th note of the 3rd instrument?

One way: `y = x(27001:28000);`

Slightly better way: `y = x(27000+(1:N));`

Still better way: `y = x((2*12+3)*N+(1:N));`

Elegant way using 3D array slicing:

```
z = reshape(x, N, 12, 3);
y = z(:,4,3);
```

`Matlab` functions introduction

`Matlab` does not have a `sqr` function for squaring values.

One way to create one is to use `edit sqr.m` and make a file called `sqr.m` containing the following two lines:

```
function y = sqr(x)
y = x.^2;
```

Now as long as the file `sqr.m` is in your `Matlab` `path` then you can use this function just like other `Matlab` functions.

Typing `sqr(5:7)`
returns `[25 36 49]`

Typing `a = [1 3 5]; b = sqr(a)`
results in variable `b` being the array `[1 9 25]`

Type `doc function` to learn more about creating functions.

Implicit function revisited

For tiny functions like the above `sqr` example,
using a separate file with only two lines is overkill.

`Matlab` provides *implicit functions* as a simpler alternative.

The following `Matlab` command defines `sqr`
to be a function of one variable: `sqr = @(x) x.^2;`

`sqr(5)`
returns 25

`sqr(5:7)`
returns `[25 36 49]`

The next page gives an bigger example that illustrates how functions can save work and facilitate customization.

Implicit function example

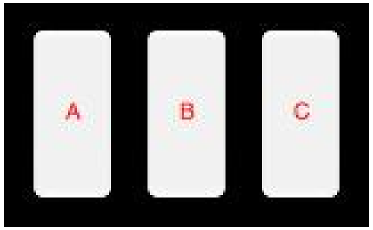
Old way:

```
uicontrol('style', 'pushbutton', 'position', [100 250 40 80], ...
      'string', 'A', 'callback', 'disp(1)', 'foregroundcolor', 'red');
uicontrol('style', 'pushbutton', 'position', [150 250 40 80], ...
      'string', 'B', 'callback', 'disp(2)', 'foregroundcolor', 'red');
uicontrol('style', 'pushbutton', 'position', [200 250 40 80], ...
      'string', 'C', 'callback', 'disp(3)', 'foregroundcolor', 'red');
```

New way:

```
fun = @(pos,str,com) uicontrol('style', 'pushbutton', ...
                    'position', pos, 'string', str, ...
                    'callback', com, 'foregroundcolor', 'red');

fun([100 250 40 80], 'A', 'disp(1)')
fun([150 250 40 80], 'B', 'disp(2)')
fun([200 250 40 80], 'C', 'disp(3)')
```



If we wanted to change the color from `red` to `blue`, which way is easier? ??

Transcriber hints: note durations

Project 3 classic synthesizer includes 100 zeros at end of each note to facilitate finding note duration.

Note	Whole	Half	Quarter	1 second
Length	$32668+100$	$16284+100$	$8092+100$	$S=44100$
	$32768=4\times 8192$	$16384=2\times 8192$	8192	

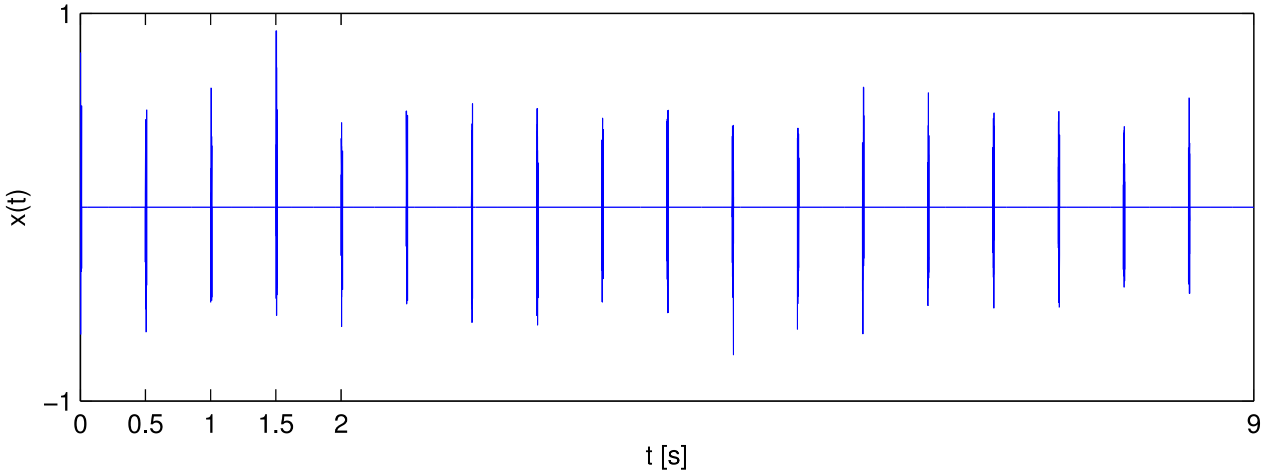
Transcriber must located those zeros. How?

```
a = reshape(x, 8192, []);
b = a(end-99:end,:);
c = sum(abs(b));
d = find(c == 0);
e = [0 d(1:end-1)]
```

Sizes of each variable? ?? ?? ?? ?? ??

Part 4. Beats per minute

Metronome signal



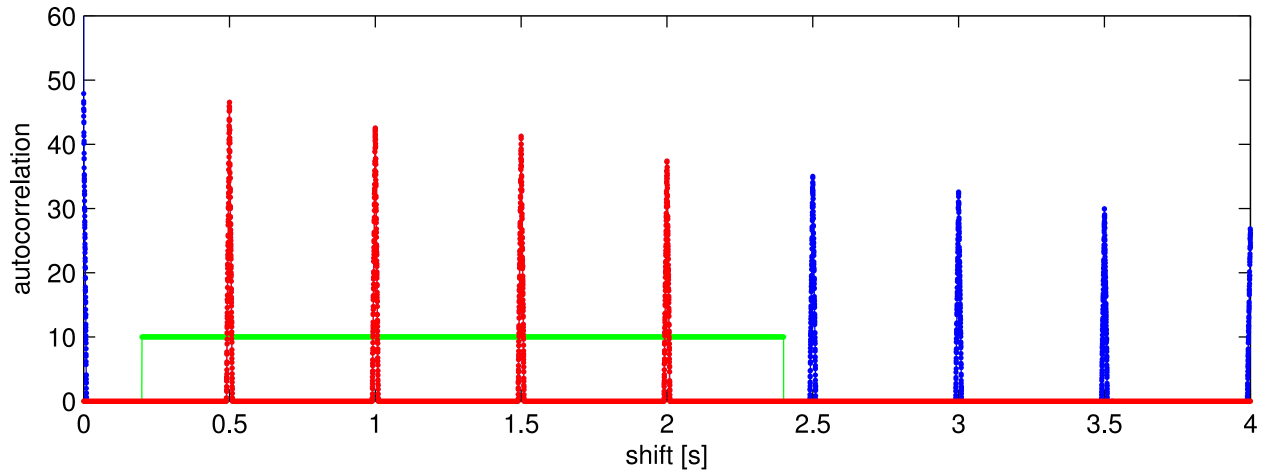
play

```
% bpm1_gen
% generate metronome tick signal to test bpm estimator

S = 8192;
bpm = 120;
bps = bpm / 60; % beats per second
spb = 60 / bpm; % seconds per beat
t0 = 0.01; % each "tick" is this long
tt = 0:1/S:9; % 9 seconds of ticking

f = 440;
%x = 0.9 * cos(2*pi*440*tt) .* (mod(tt, spb) < t0); % tone
clf, subplot(211), rng(0)
x = randn(1,numel(tt)) .* (mod(tt, spb) < t0) / 4.5; % click via "envelope"
% sound(x, S)
% wavwrite(x, S, 8, 'bpm1a.wav')
```

Metronome BPM




```
% bpm1_find
% first try at beats-per-minute (bpm) estimator

[x S] = wavread('bpm1a.wav');
x = x'; % row vector
N = numel(x);

%a = real(ifft(abs(fft(x,2*N)).^2)); % autocorrelation
a = real(ifft(abs(fft(abs(x),2*N)).^2)); % why abs?

spacing = [0:(2*N-1)]/S; % why?
good = (spacing > 60/300 & spacing < 60/25); % min and max reasonable bpm
clf, subplot(211)
plot(spacing, a, 'b.-', spacing, 10*good, 'g.-', spacing, a .* good, 'r.-')
xlabel 'shift [s]', ylabel 'autocorrelation', axis([0 4 0 60])
[~, index] = max(a .* good); % highest correlation for reasonable bpm range
disp(sprintf('estimated spb = %g, so bpm = %g', index/S, 60*S/index))
% ir_savefig -tight cw fig_bpm1b
```

BPM Summary

This small example illustrates several useful ideas.

- Noise blips
- Using modulo `mod` for repeating patterns
- Using logical operations like $<$ to make binary signals.
- Constraining `max` to reasonable search range
- Looking for correlation between bursts of noisy signals using `abs`
- A few lines of `Matlab` code can do sophisticated DSP operations

Summary: (auto)correlation is quite widely useful