Eng. 100: Music Signal Processing
DSP Lecture 9
Music synthesis techniques

Curiosity:
- Let a thousand analog oscillators sing
  https://doi.org/10.1109/MSPEC.2020.9205540
  https://www.youtube.com/watch?v=c3wk9WWTfNs
  https://www.youtube.com/watch?v=M12kjjmD02E
- http://www.image-line.com/plugins/Synths/Harmor

Announcements:
- https://web.eecs.umich.edu/~fessler/course/100/p/synthesis.pdf

Outline

- Part 1: `Julia` functions and loops
- Part 2: Additive synthesis via Fourier series
- Part 3: FM synthesis
- Part 4: Nonlinearities
- Part 5: Envelope: analysis and synthesis
- Part 6: P3 Q/A

Learning objectives

- Understand effective coding principles
  - use functions and loops (don't repeat yourself)
  - separate data from code
- Understand additive synthesis and efficient implementation
- Awareness of FM synthesis
- Awareness of nonlinear effects
- Understand envelope

Part 1: `Julia` functions and loops

Julia functions (1)

Tedious way to write a 4-note song in Julia (violates DRY principle):

```
using Sound: sound
S = 8192
sound(0.9cos.(2π*660*(0:1/S:0.5)), S); sleep(0.5)
sound(0.9cos.(2π*880*(0:1/S:0.5)), S)
sound(0.9cos.(2π*660*(0:1/S:0.5)), S)
sound(0.9cos.(2π*440*(0:1/S:1.0)), S)
```

Q0.1 What is the duration of this song (in seconds)?

??

Q0.2 Why the `sleep` call?

??

Can we streamline this code (*e.g.*, for longer songs)? ??

Julia functions (2)

Using a function can simplify and clarify:

```
using Sound: sound
S = 8192
playnote = (f,d) -> sound(0.9cos.(2π*f*(0:1/S:d)), S)
playnote(660, 0.5); sleep(0.5)
playnote(880, 0.5)
playnote(660, 0.5)
playnote(440, 1.0)
```

Caution: if we change `S`, then how `playnote` works will change.

- Functions defined dynamically and function closure s are *very* useful (in languages that support them).
- 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...

Julia loops (1)

Using a `for` loop is the most concise and elegant:

```
using Sound: sound
S = 8192 # data
fs = [660, 880, 660, 440] # frequencies
ds = [0.5, 0.5, 0.5, 1.0] # note durations
rs = [0.5, 0.0, 0.0, 0.0] # rest durations
# code (separated from data):
playnote = (f,d) -> sound(0.9cos.(2π*f*(0:1/S:d)), S)
for index in 1:length(fs)
    playnote(fs[index], ds[index])
    sleep(rs[index])
end
```

Another principle illustrated here: separate code and data.
Complete separation? ??

Julia loops (2)

Here is another loop version that sounds better: play

```
using Sound: sound
S = 8192 # "data"
fs = [660, 880, 660, 440] # frequencies
ds = [0.5, 0.5, 0.5, 1.0] # note durations
rs = [0.5, 0.0, 0.0, 0.0] # rest durations
x = Float32[] # "code"
for (f,d,r) in zip(fs, ds, rs)
    append!(x, 0.9cos.(2π*f*(0:1/S:d)), S) # note
    append!(x, zeros(round(Int, r/S))) # rest
end
sound(x, S)
```

Loops (and functions) are ubiquitous in software.

`Julia` functions

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

```
edit playsong.jl
```

Then test it:

```
playsong(220 * [3, 4, 3, 2, 3, 4, 3], [1, 1, 1, 1, 1, 1, 2]/3)
```

Part 2: Additive synthesis via Fourier series

Additive Synthesis: Mathematical formula

Simplified version of Fourier series for monophonic audio:

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

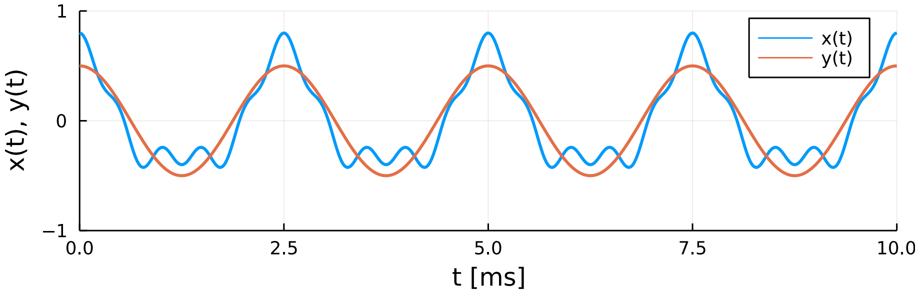
- No DC term for audio: $c_0 = 0$.
- Phase unimportant for monophonic audio, so $\theta_k = 0$.
- Which version is this? $\boxed{??}$

  (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



play play

Same pitch,
different timbre.

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

```julia
# fig_why1.jl example of additive synthesis
using Measures: mm
using Plots
default(linewidth=2, size = (600,200), left_margin = 2mm, bottom_margin = 4mm)
S = 44100
N = Int(0.5 * S) # 0.5 sec
t = (0:N-1)/S # time samples: t = n/S
y = 0.5 * cos.(2π * 400 * t)
x = y + 0.2 * cos.(2π * 800 * t) + 0.1 * cos.(2π * 2000 * t)
plot(1000t, x, label="x(t)", xlabel="t [ms]", xlims=(0,10), xticks=1000*(0:4)/400)
plot!(1000t, y, label="y(t)", ylabel="x(t), y(t)", ylims = (-1, 1), yticks=-1:1)
#savefig("fig_why1.pdf")
```

Julia 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 Julia version looks a lot like the mathematical formula:

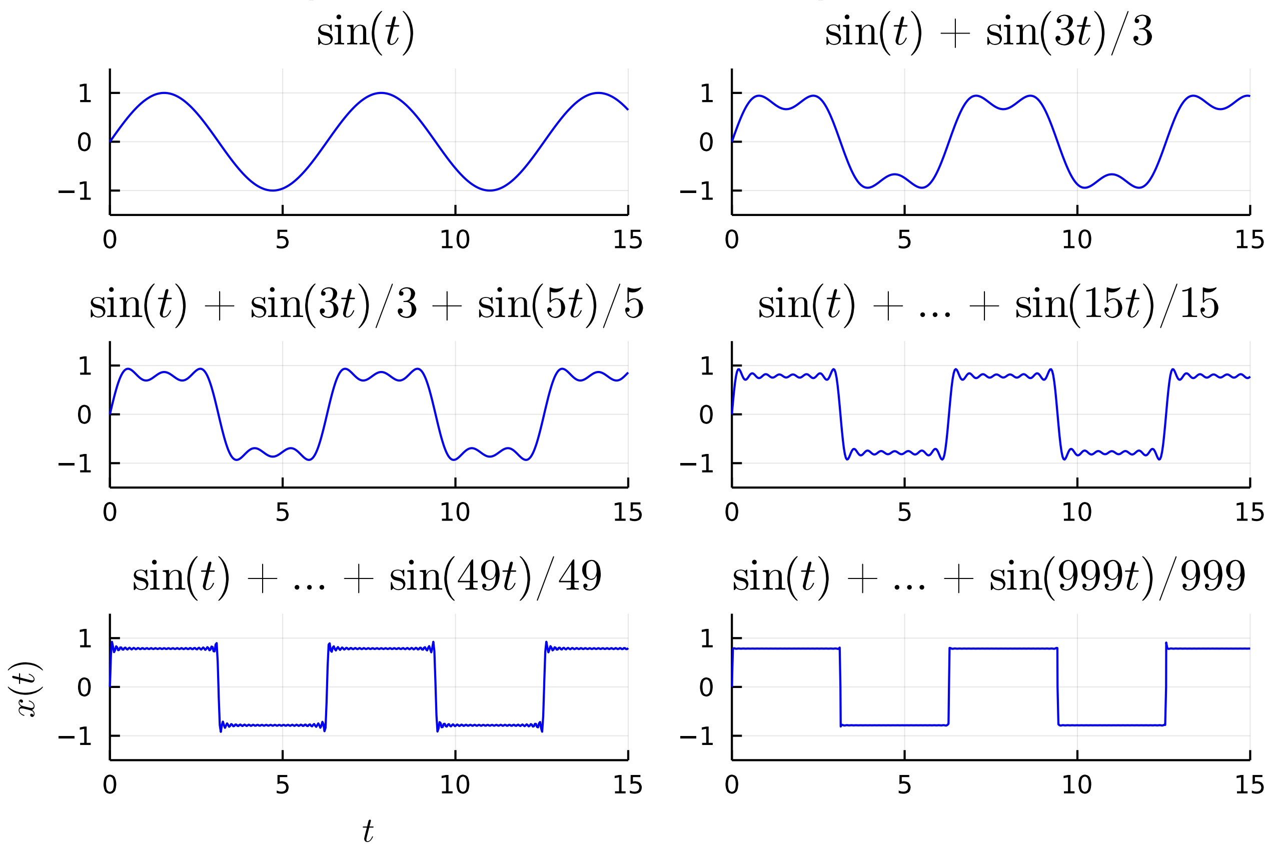
```
S = 44100
N = Int(0.5 * S) # 0.5 sec
t = (0:N-1)/S # time samples: t = n/S
x = 0.5 * cos.(2π * 400 * t) +
    0.2 * cos.(2π * 800 * t) +
    0.1 * cos.(2π * 2000 * t)
```

- There are many "hidden" `for` loops above. Where? $\boxed{??}$
- Julia 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.

C99 implementation

```
#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);
    }
}
```

Example revisited: Square wave



Many dozens of harmonics needed to get a "good" square wave approximation.

Julia implementation: Loop over harmonics

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

```
S = 44100
N = Int(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 = zeros(N)
for k in 1:length(c)
    global x += c[k] * cos.(2π * f[k] * t)
end
```

Q0.3 How many loops over N in this version?

$\boxed{??}$

- 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.

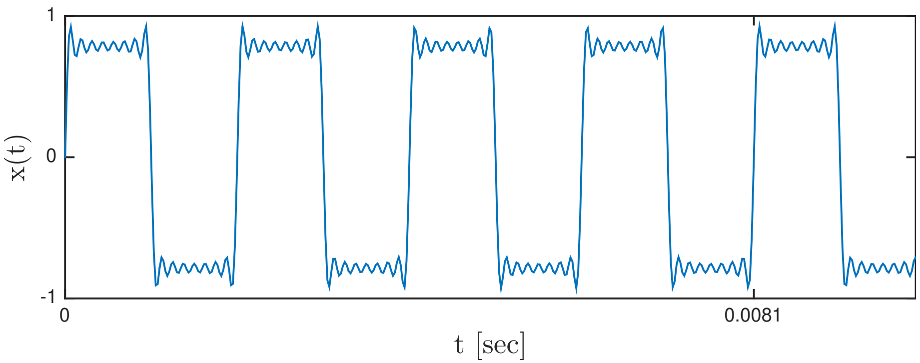
(We would not need `global` if we put this in a function.)

Example: Square wave via loop, with sin

```
S = 44100
N = Int(0.5 * S) # 0.5 sec
t = (0:N-1)/S # time samples: t = n/S
c = 1 ./ (1:2:15) # amplitudes
f = (1:2:15) * 494 # frequencies
x = zeros(N)
for k in 1:length(c)
    global x += c[k] * sin.(2π * f[k] * t)
end
```

Q0.4 How many (non-fundamental) harmonics in this example?

??



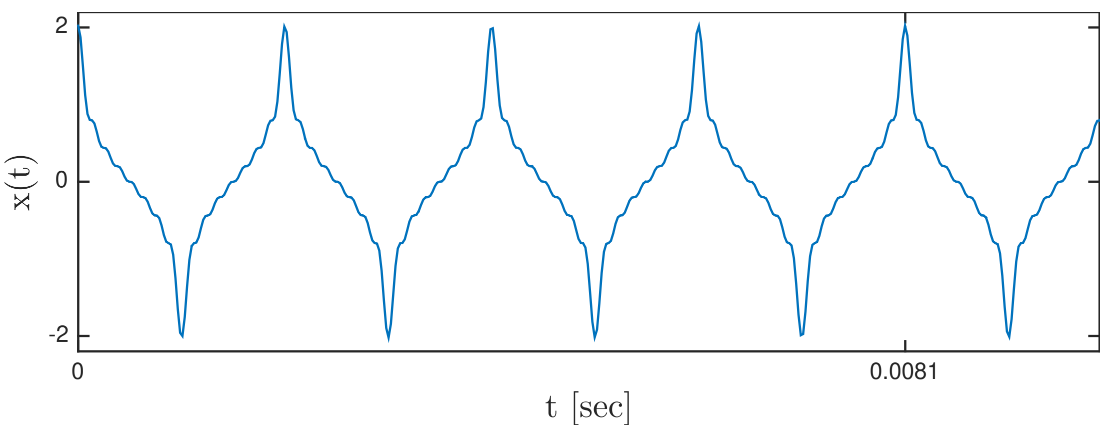
play

Music example, circa 1978: play

2011 example? Nyan cat

Example: Square wave via loop, with cos

```
S = 44100
N = Int(0.5 * S) # 0.5 sec
t = (0:N-1)/S # time samples: t = n/S
c = 1 ./ (1:2:15) # amplitudes
f = (1:2:15) * 494 # frequencies
x = zeros(N)
for k in 1:length(c)
    global x += c[k] * cos.(2π * f[k] * t)
end
```



play

Does it sound different? ??

Julia implementation: Concise

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

```
S = 44100
N = Int(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.(2π * t * f')
x = z * c
```

`c` is $3(\times 1)$ `f'` is 1×3 `t` is $N(\times 1)$

`z` is ??

`x` is ??

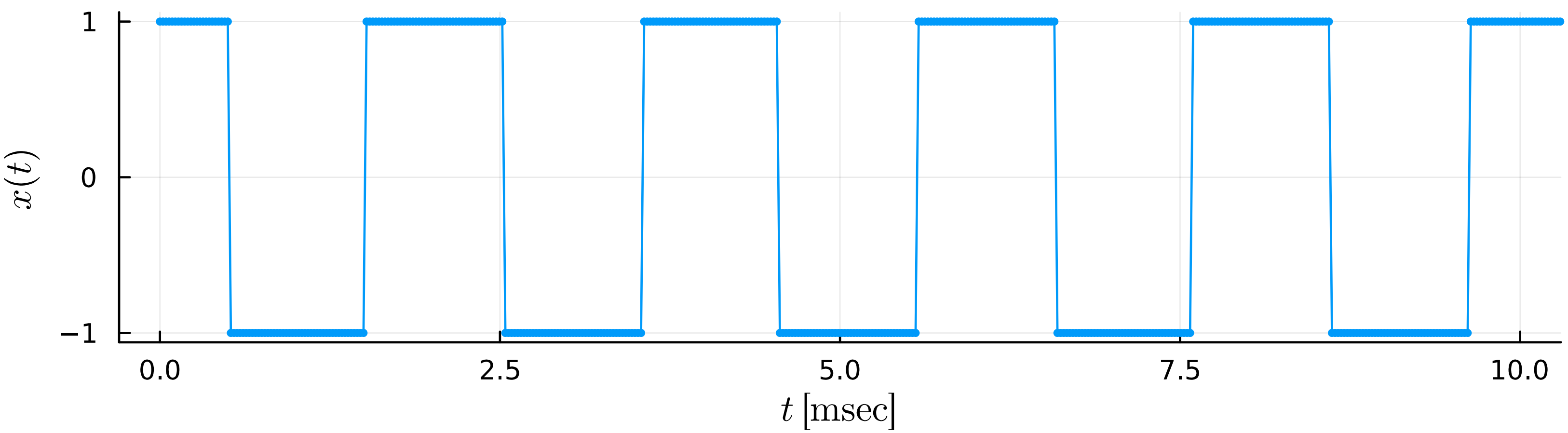
Q0.5 Where are the (hidden) loops in this version?

??

Use this approach or the previous slide for Project 3 additive synthesis.

Example: Square wave via sign

```
S = 44100
N = Int(0.5 * S) # 0.5 sec
t = (0:N-1)/S # time samples: t = n/S
f = 494 # (fundamental) frequency
x = sign.(cos.(2π * f * t))
```



play

Simplest code, but least customizable.

Wavetable synthesis

See synthesis.pdf document.

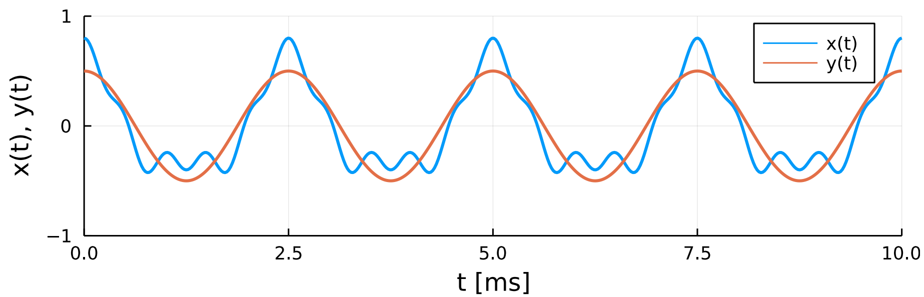
Part 3: FM synthesis

Additive synthesis review

Mathematical formula (Fourier series) for additive synthesis [wiki]:

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

- Parameters that control timbre: $c_1, \ldots, c_K$
- Parameter that controls pitch: ??



play play

Frequency Modulation (FM) synthesis

In 1973, John Chowning of Stanford invented the use of frequency modulation (FM) as a technique for musical sound synthesis [1, 2].

The mathematical formula for FM synthesis is [wiki]:

$$x(t) = A\sin(2\pi f t + I\sin(2\pi g t)),$$

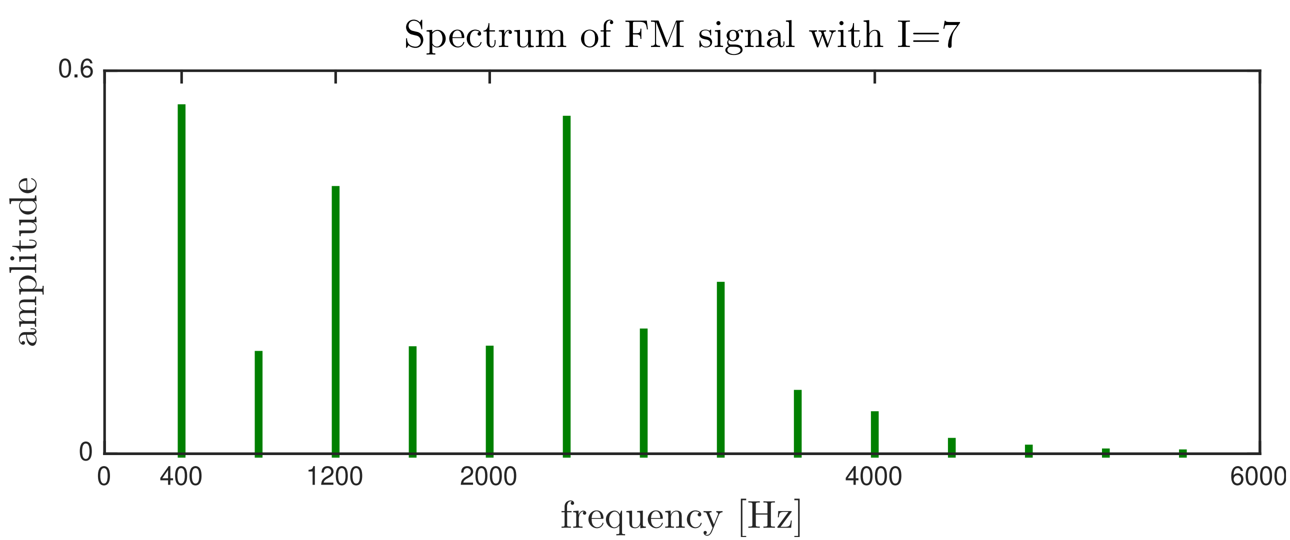
where I is the modulation index and f and g are both frequencies.
(Yamaha licensed the patent for synthesizers and Stanford made out well.)

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 = Int(1.0 * S)
t = (0:N-1)/S # time samples: t = n/S
I = 7 # adjustable
x = sin.(2π*400*t + I * sin.(2π*400*t))
```

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



play

FM example 2: Time-varying

Time-varying modulation index: $x(t) = A \sin\left(2\pi f t + \underbrace{I(t)} \sin(2\pi g t)\right).$

Simple formula / implementation can make intriguing sounds. play

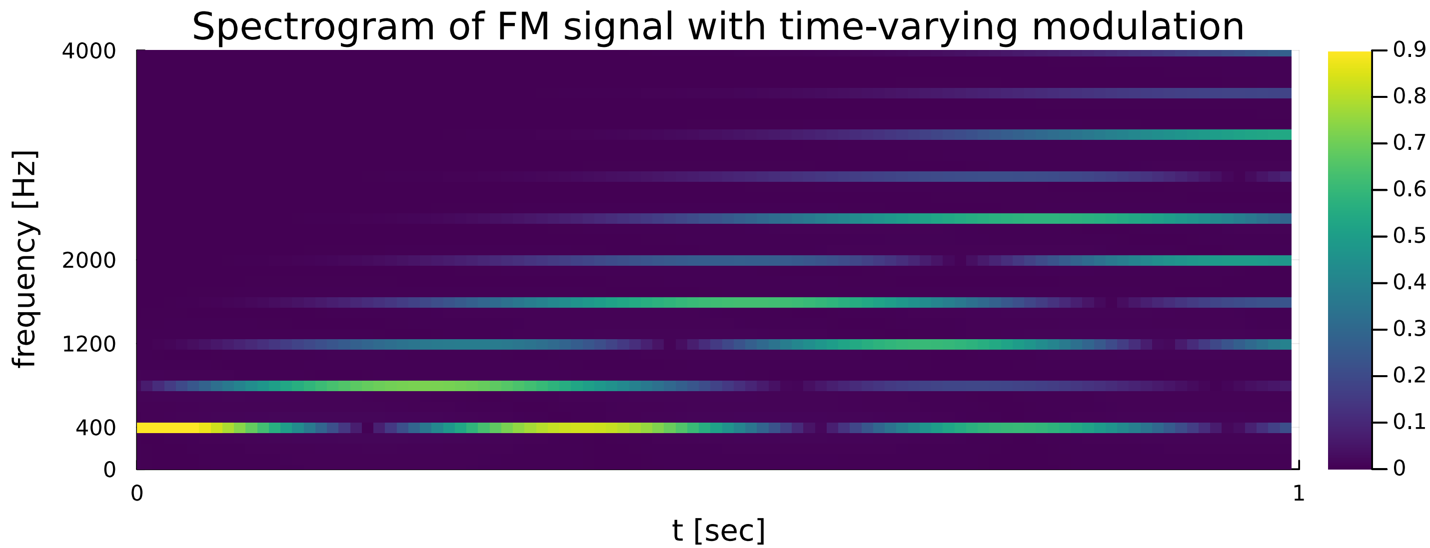
```
S = 44100
N = Int(1.0 * S)
t = (0:N-1)/S # time samples: t = n/S
I = 0 .+ 9*t/maximum(t) # slowly increase modulation index
x = sin.(2π*400*t + I .* sin.(2π*400*t))
```

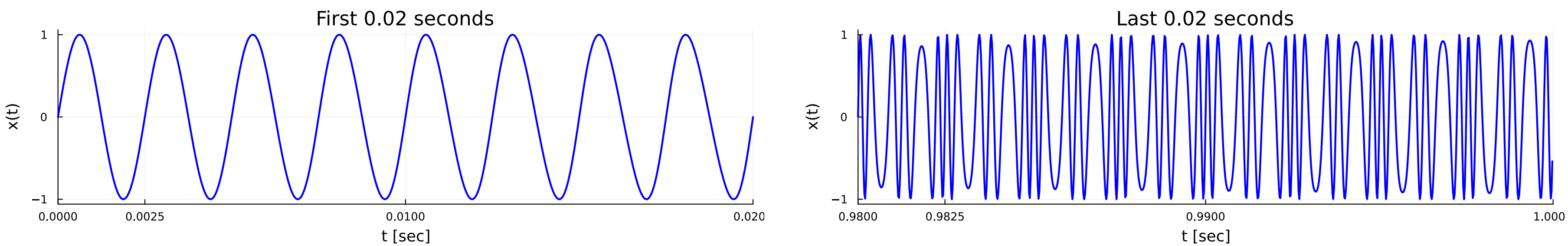
Besides making the modulation index `I` a vector, how else did the code change? ??
Previous code for reference:

```
S = 44100
N = Int(1.0 * S)
t = (0:N-1)/S # time samples: t = n/S
I = 7 # adjustable
x = sin.(2π*400*t + I * sin.(2π*400*t))
```

Q0.6 What is the most informative graphical representation?
A: Time plot B: FFT spectrum C: Line spectrum D: Spectrogram E: None of these $\boxed{??}$

Illustrations of previous FM signal





Q0.7 What is the approximate fundamental frequency of the final 0.02 seconds?

$\boxed{??}$

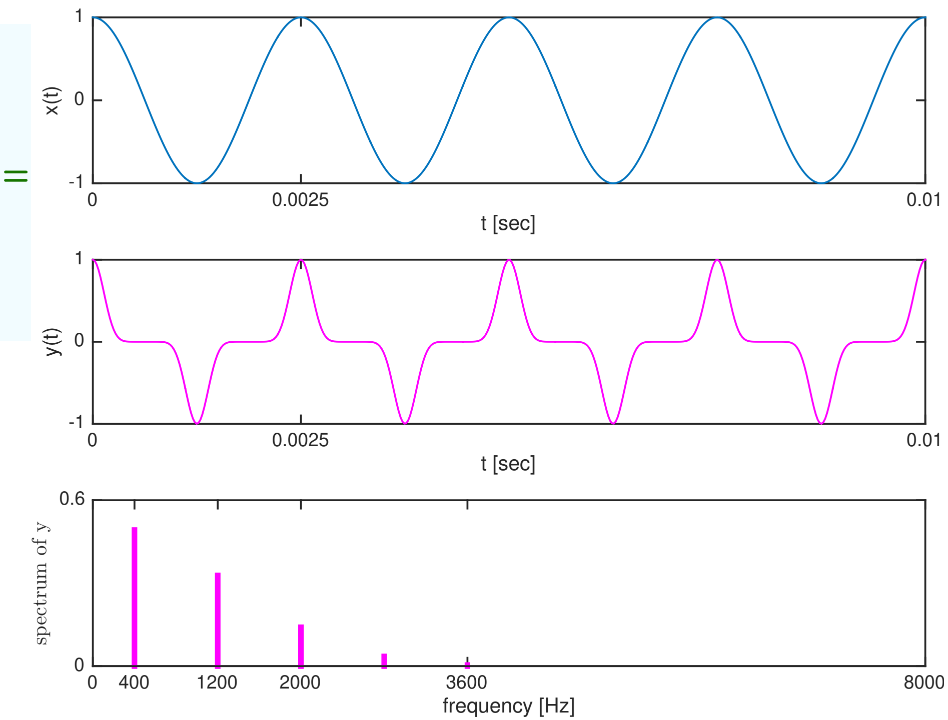
Part 4: Nonlinearities

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 = Int(1.0 * S)
t = (0:N-1)/S # time samples: t =
x = cos.(2π*400*t)
y = x.^9
```

play



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.

$$\underbrace{\cos(2\pi f t)}_{\text{input}} \to \boxed{\text{Amplifier}} \to c_1 \cos(2\pi f t) + c_2 \cos(2\pi 2 f t) + c_3 \cos(2\pi 3 f t) + \cdots$$

- A formula for THD is: [wiki]

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

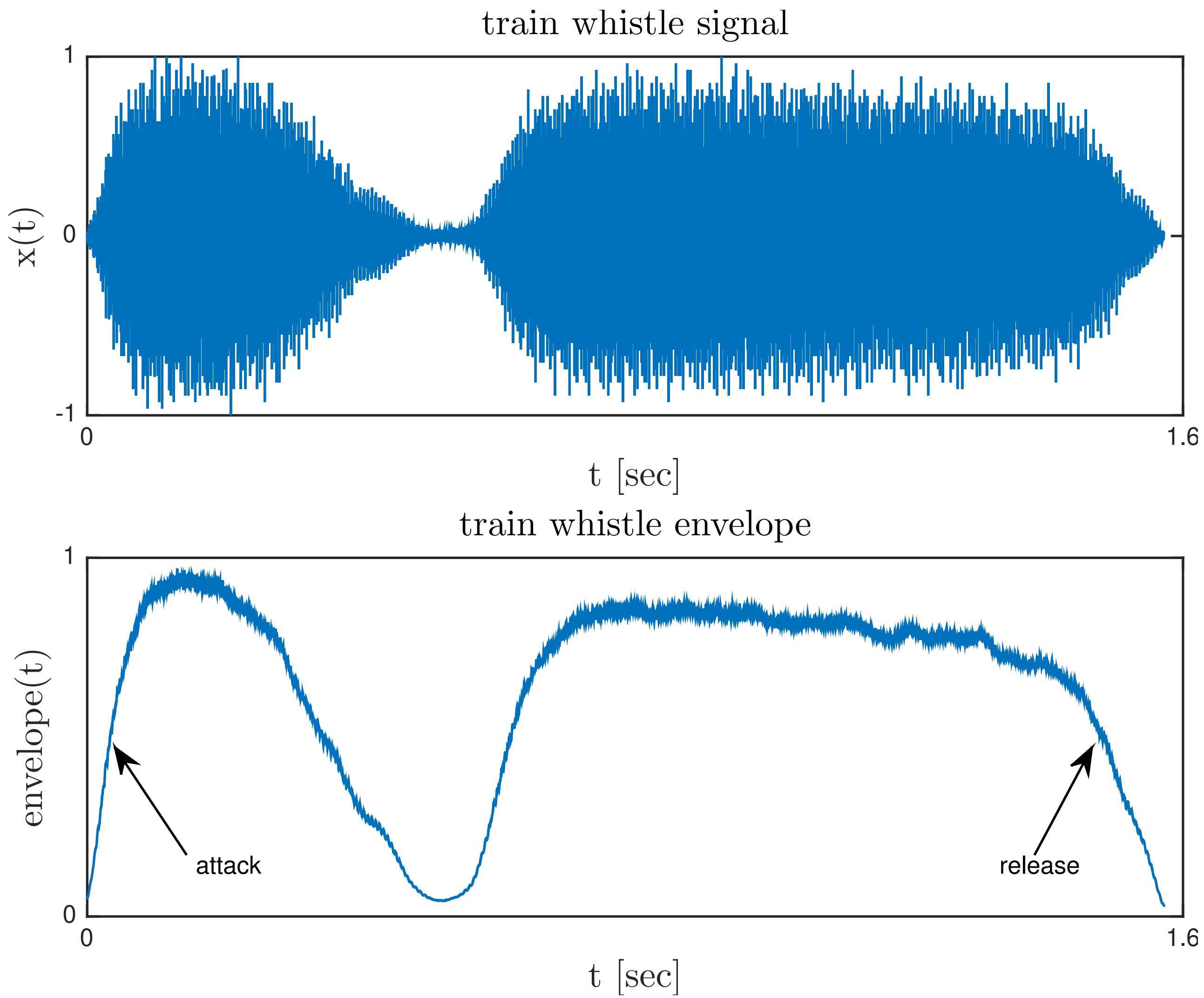
- What is the best possible value for THD? ??
- On the other hand, electric guitarists often deliberately operate their amplifiers nonlinearly to induce distortion, thereby introducing more harmonics than produced by a simple vibrating string.
  - pure: play
  - distorted: play

Part 5: Envelope of musical signals

Envelope has two applications

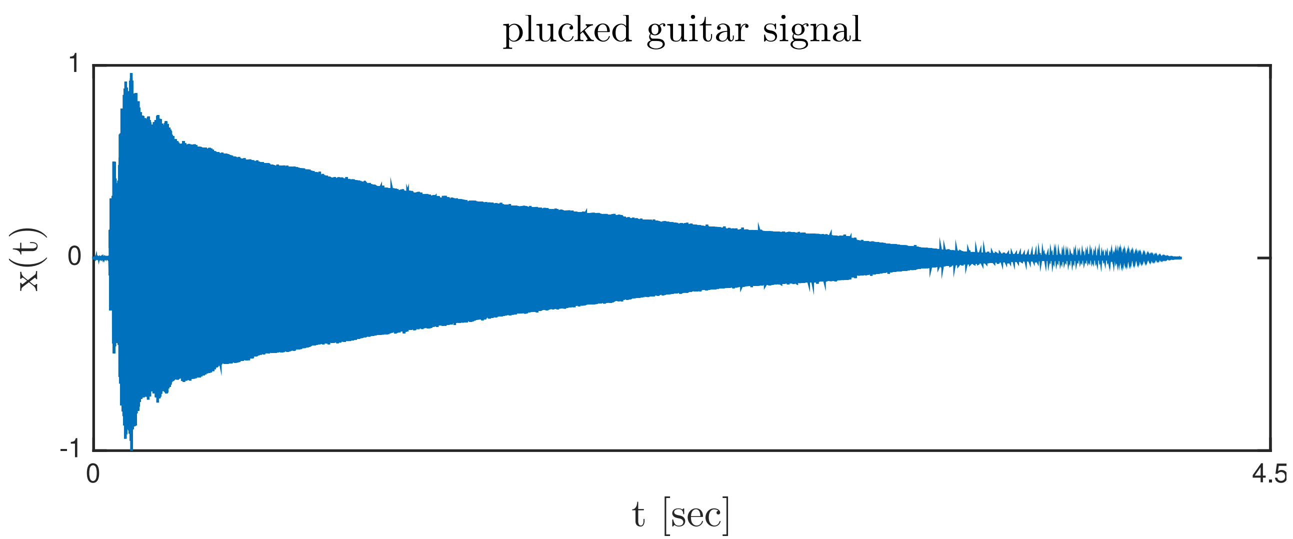
- Analyzing (processing) musical signals
  (*e.g.*, to find note start/stop/duration, rhythms)
- Synthesizing (interesting or more realistic) musical signals

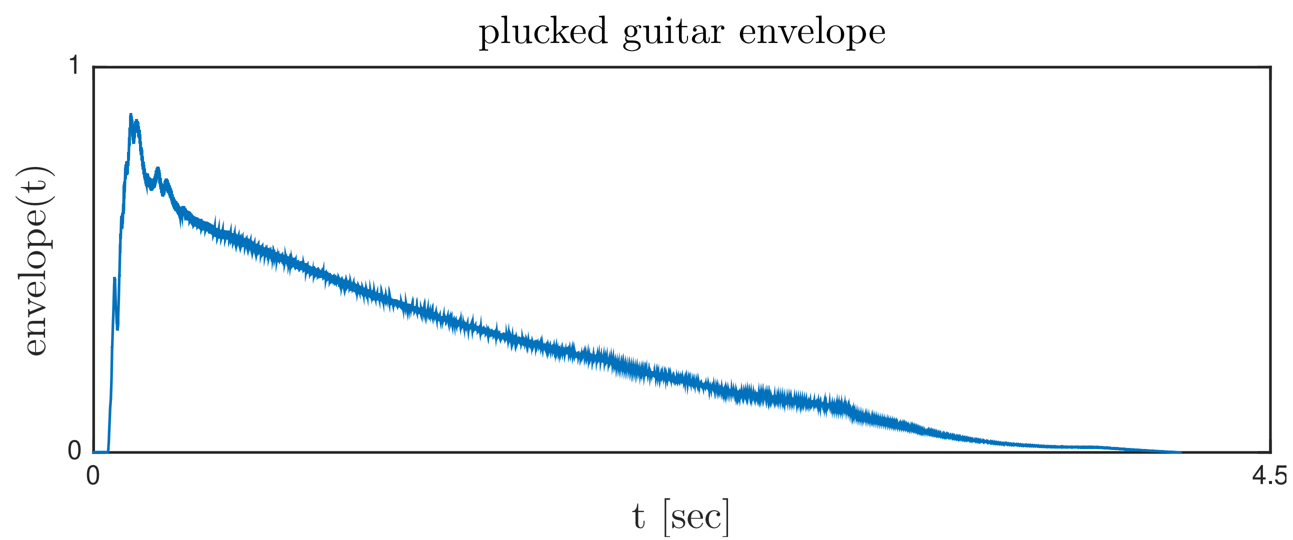
Envelope example: Train whistle



play

Envelope example: Plucked guitar





play

Compute envelope with moving average

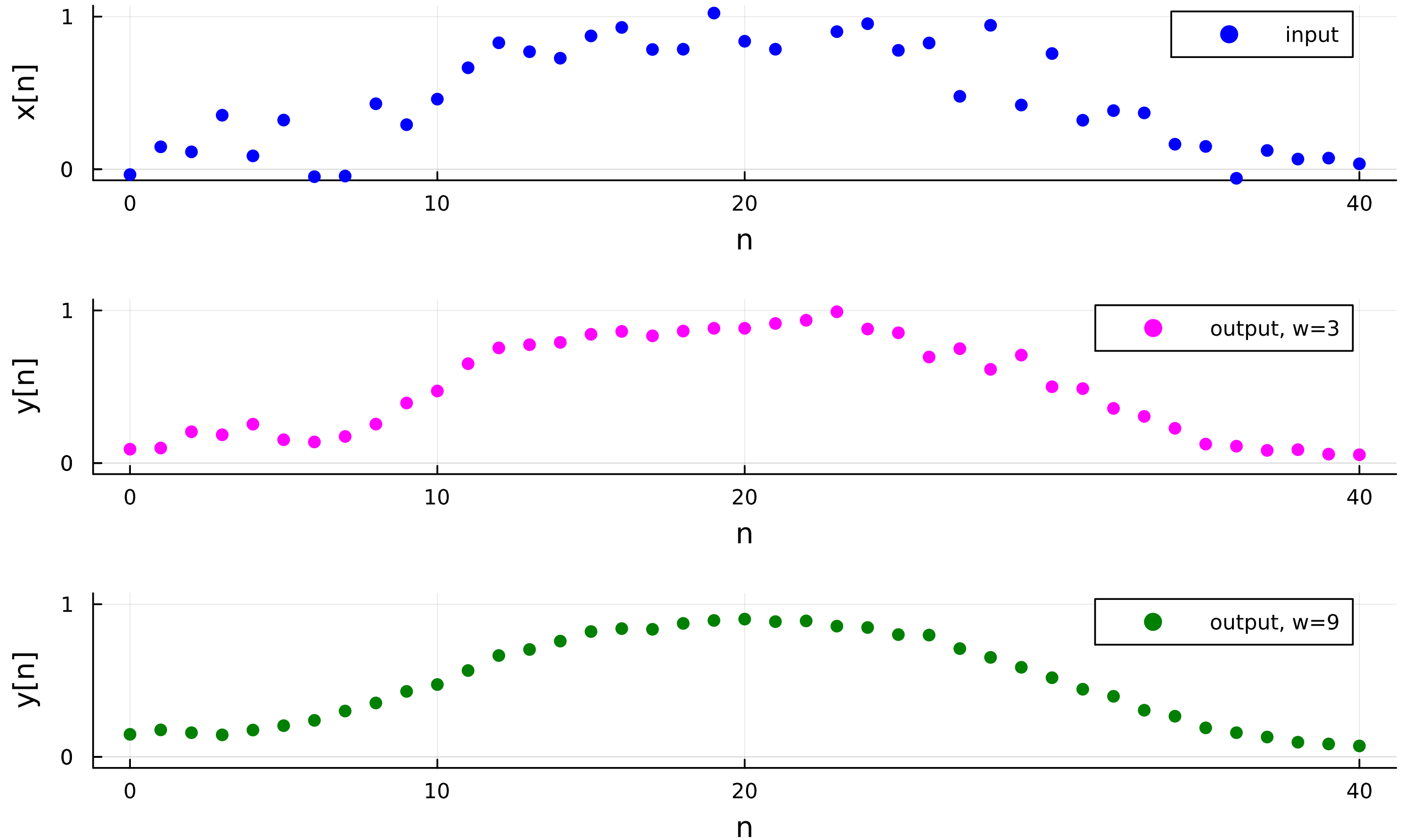
Find envelope using moving average, aka "sliding window"

```
function envelope(x; w::Int = 201) # uses moving average
    h = (w-1) ÷ 2 # sliding window half-width (default 100)
    x = abs.(x) # absolute value is crucial!
    avg(v) = sum(v) / length(v) # function for (moving) average
    return [avg(x[max(n-h,1):min(n+h,end)]) for n in 1:length(x)]
end
```

```
using WAV: wavread
using Plots
x, S = wavread("train-whistle.wav")
env = envelope(x) # call moving-average function
plot((1:length(x))/S, env, label="envelope", xlabel="t [sec]")
```

The preceding two figures used this code.

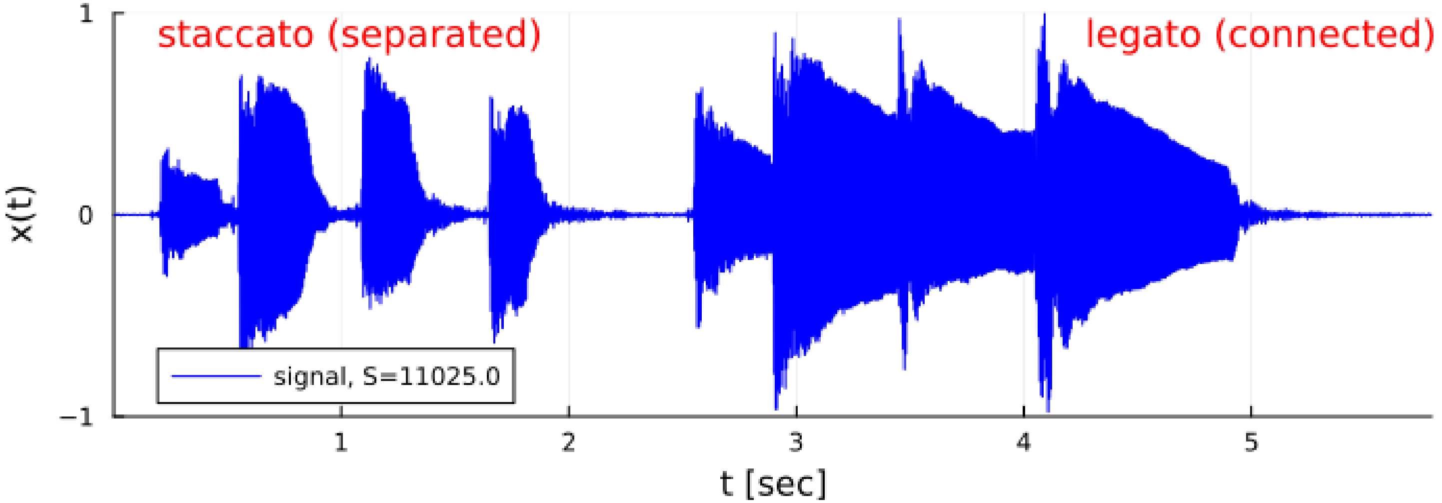
Moving average



$$y[10] = \frac{1}{3}(y[9] + y[10] + y[11])$$

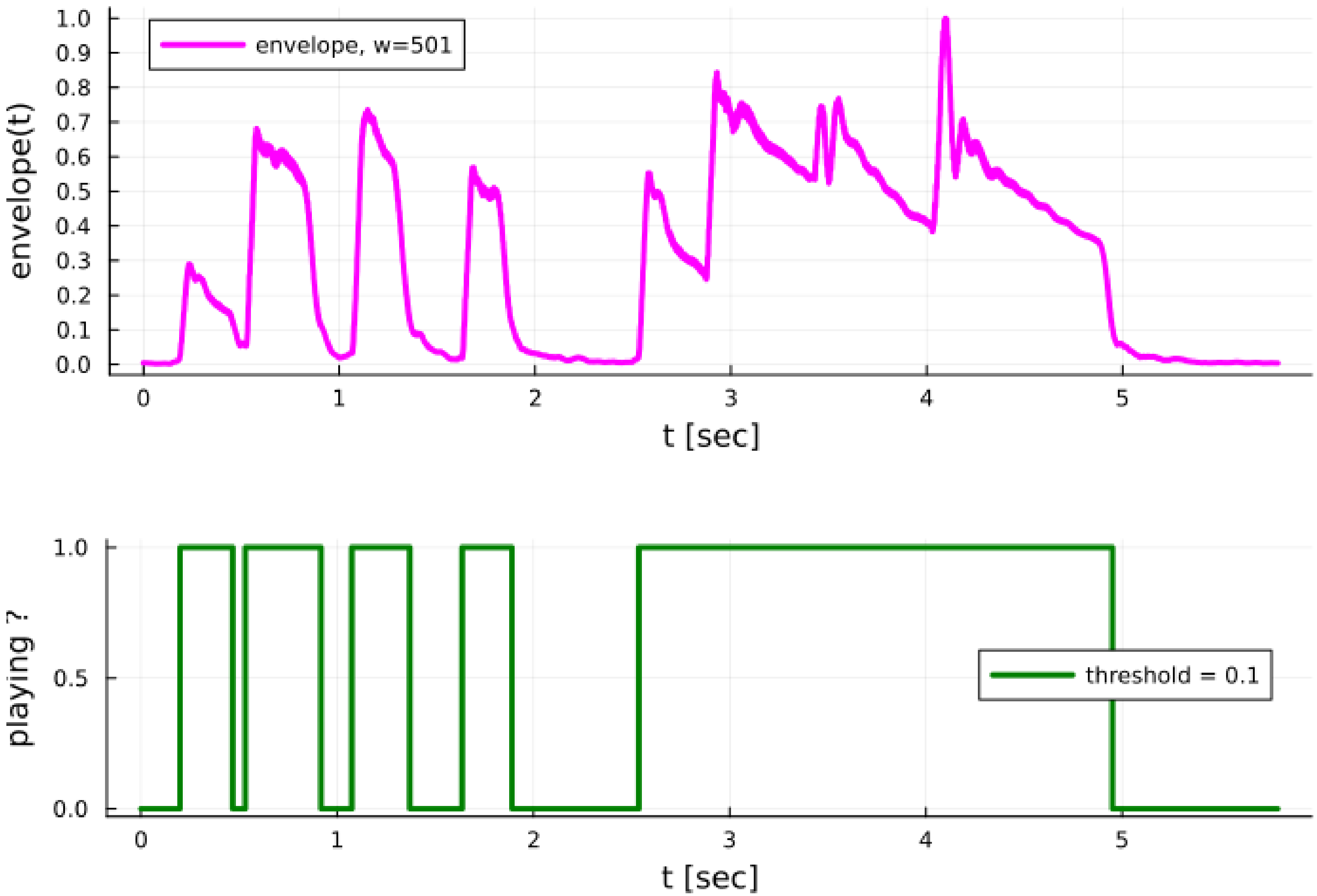
$$z[n] = \frac{1}{9}\sum_{k=-4}^{4} x[n+k], \text{ for } 5 < n < N-5$$

Find note durations using envelope



play

```
w = 501
env = envelope(x; w)
env /= maximum(env)
threshold = 0.1
playing = env .> threshold
```

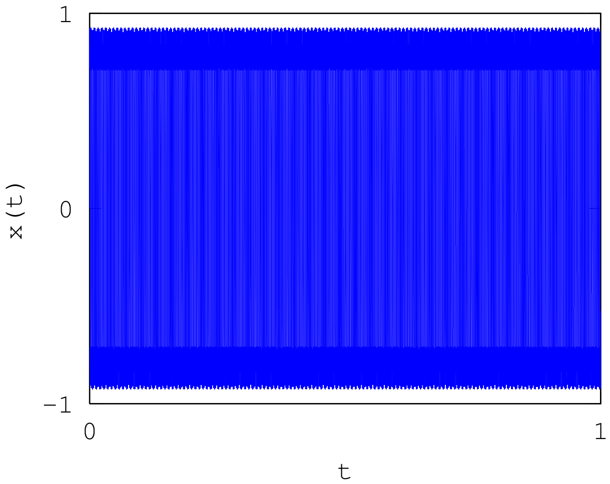


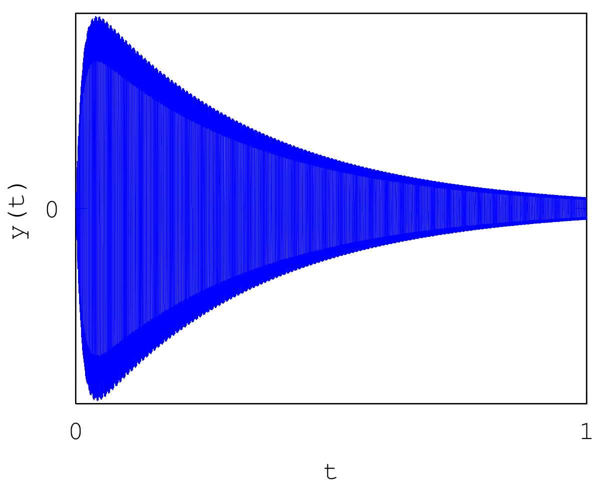
play

Simple threshold sufficed for separated notes;
legato notes need more effort

Envelope synthesis in Julia

```
S = 44100
N = Int(1 * S)
t = (0:N-1)/S
c = 1 ./ (1:2:15) # amplitudes
f = (1:2:15) * 494 # frequencies
x = +([c[k] * sin.(2π * f[k] * t) for k in 1:length(c)]...) # !!
env = (1 .- exp.(-80*t)) .* exp.(-3*t) # fast attack; slow decay
y = env .* x
```

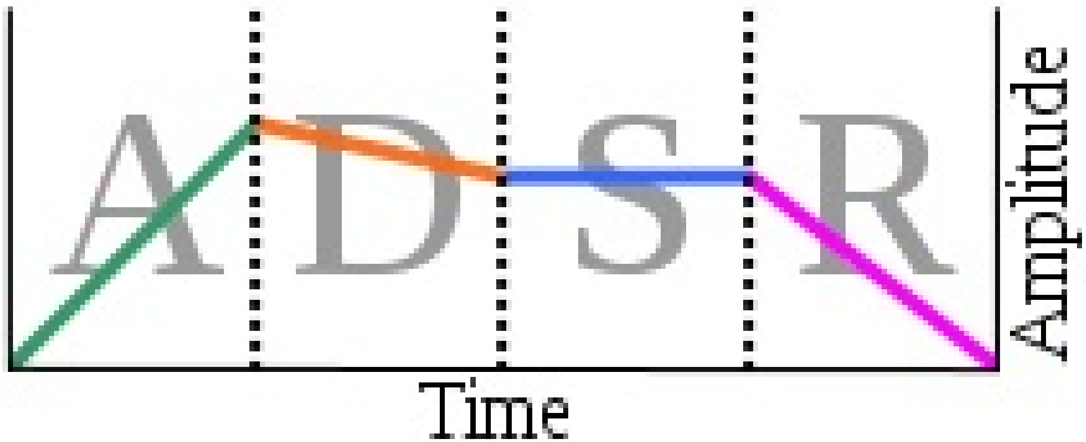




x: play y: play

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.



[wiki]

- 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.
- Does duration of release portion depend on how quickly one releases the key? ??
- For a pipe organ, how long is the attack and decay?
  ??

Music synthesis 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
- Other advanced synthesis methods:
  - sound reversal
  - physical modeling
  - sampling
  - ...
- Ample room for creativity and originality!

Part 6: P3 Q/A?

References

[1] J. M. Chowning. The synthesis of complex audio spectra by means of frequency modulation. *J. of the Audio Engineering Soc.*, 21(7):526–34, September 1973.

[2] J. M. Chowning. The synthesis of complex audio spectra by means of frequency modulation. *Computer Music J.*, 1(2):46–54, April 1977.