Digital Signal Processing

Lecture 4 – Harmonics

상명대학교
컴퓨터과학과
강상욱 교수

Triangle waves I

- Triangle wave : a straight-line version of a sinusoid.
- To generate a triangle wave : input arguments (freq, amp, offset)

One period
=1/200 sec
= 0.005 sec

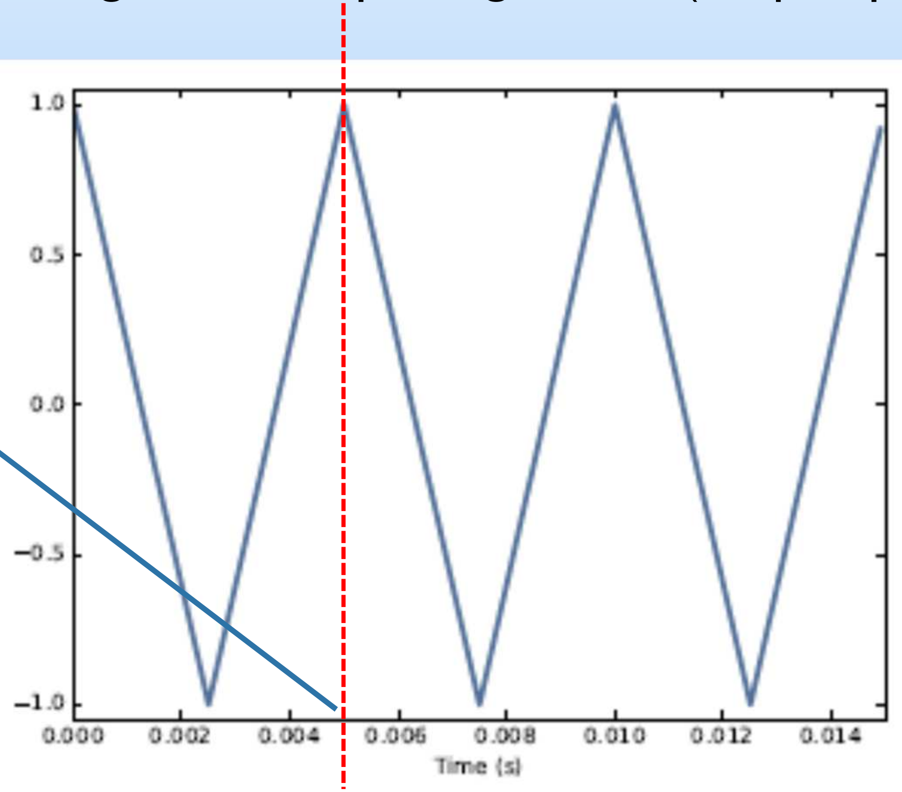


Figure 2.1: Segment of a triangle signal at 200 Hz.

Triangle waves 1

- Triangle wave : a straight-line version of a sinusoid.
- To generate a waveform, we need to evaluate the signal.

```
def make_wave(self, duration=1, start=0, framerate=11025):
    n = round(duration * framerate)
    ts = start + np.arange(n) / framerate
    ys = self.evaluate(ts)
    return Wave(ys, ts, framerate=framerate)
```

Recall make_wave

```
class TriangleSignal(Sinusoid):

  def evaluate(self, ts):
    # Evaluates the signal at the given times.
    # ts: float array of times
    # returns: float wave array

    cycles = self.freq * ts + self.offset / PI2
    frac, _ = np.modf(cycles)
    ys = np.abs(frac - 0.5)
    ys = normalize(unbias(ys), self.amp)
    return ys
```

Triangle waves 2

1. `cycles` is the number of cycles since the start time. `np.modf` splits the number of cycles into the fraction part, stored in `frac`, and the integer part, which is ignored [1].

2. `frac` is a sequence that ramps from 0 to 1 with the given frequency. Subtracting 0.5 yields values between -0.5 and 0.5. Taking the absolute value yields a waveform that zig-zags between 0.5 and 0.

3. `unbias` shifts the waveform down so it is centered at 0; then `normalize` scales it to the given amplitude, `amp`.

```
signal = thinkdsp.TriangleSignal(200)
signal.plot()
wave = signal.make_wave(duration=0.5, framerate=10000)
spectrum = wave.make_spectrum()
spectrum.plot()
```

Spectrum of triangle signal

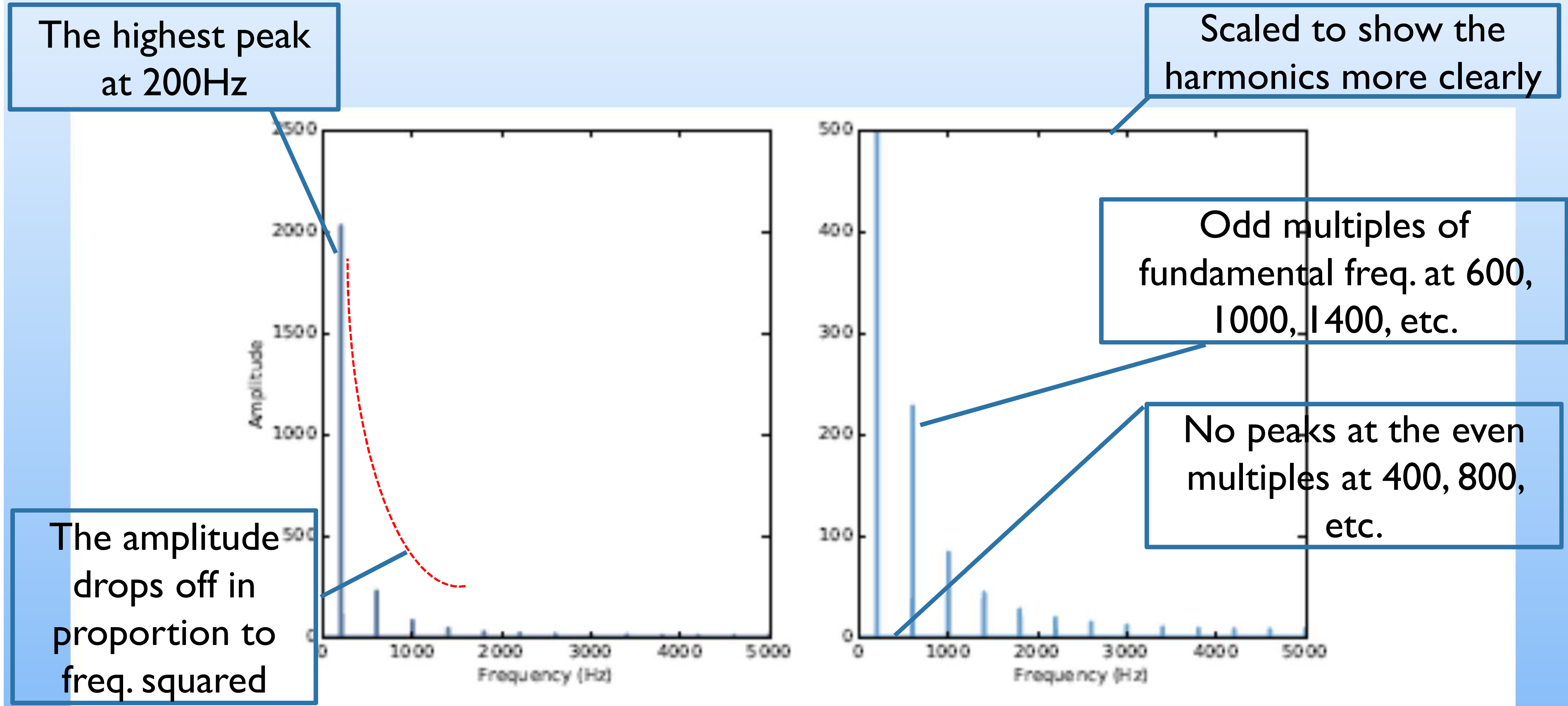


Figure 2.2: Spectrum of a triangle signal at 200 Hz, shown on two vertical scales. The version on the right cuts off the fundamental to show the harmonics more clearly.

The harmonic structure

- The amplitude drops off in proportion to freq. squared
  - The first two harmonics
    - freq. ratio=3 (600Hz/200Hz), amp. ratio=9
    - $3^2 = 9$
  - The next two harmonics
    - freq. ratio=1.7 (1000Hz/600Hz), amp. ratio = 2.9
    - $1.7^2 = 2.9$

Square waves 1

- To generate a triangle wave : input arguments (freq, amp, offset)

```
class SquareSignal(Sinusoid):

    def evaluate(self, ts):
        cycles = self.freq * ts + self.offset / PI2
        frac, _ = np.modf(cycles)
        ys = self.amp * np.sign(unbias(frac))
        return ys
```

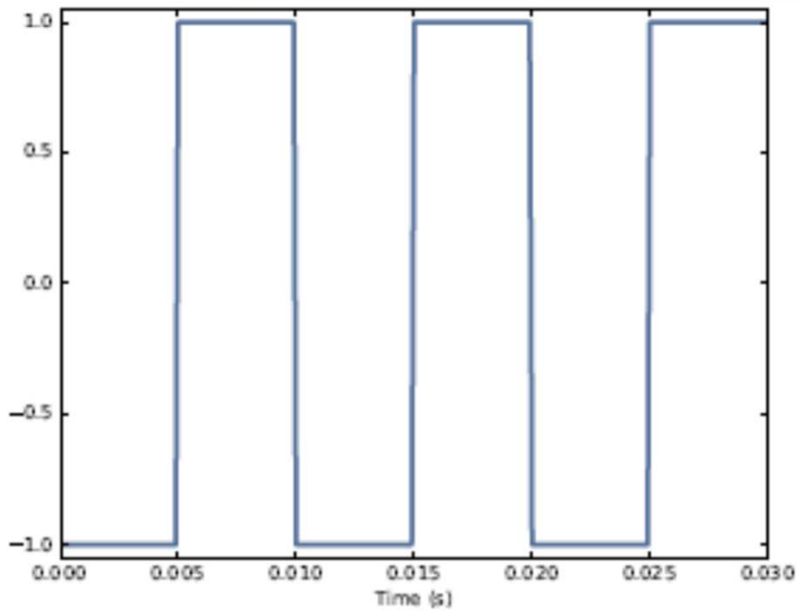


Figure 2.3: Segment of a square signal at 100 Hz.

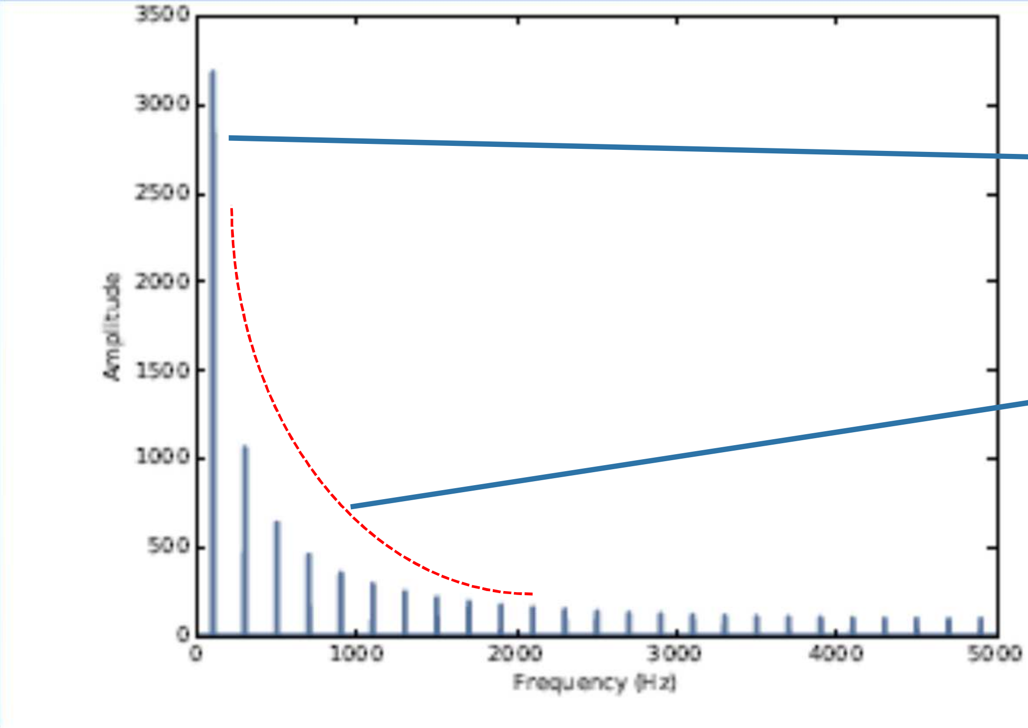
Square waves 2

And the `evaluate` method is similar. Again, `cycles` is the number of cycles since the start time, and `frac` is the fractional part, which ramps from 0 to 1 each period.

`unbias` shifts `frac` so it ramps from -0.5 to 0.5, then `np.sign` maps the negative values to -1 and the positive values to 1. Multiplying by `amp` yields a square wave that jumps between `-amp` and `amp`.

Spectrum of square signal

- A signal : a Python representation of a mathematical function.
  - It is defined for all values of t, from negative infinity to infinity.



The square wave contains only odd harmonics at 300, 500, 700Hz, etc.

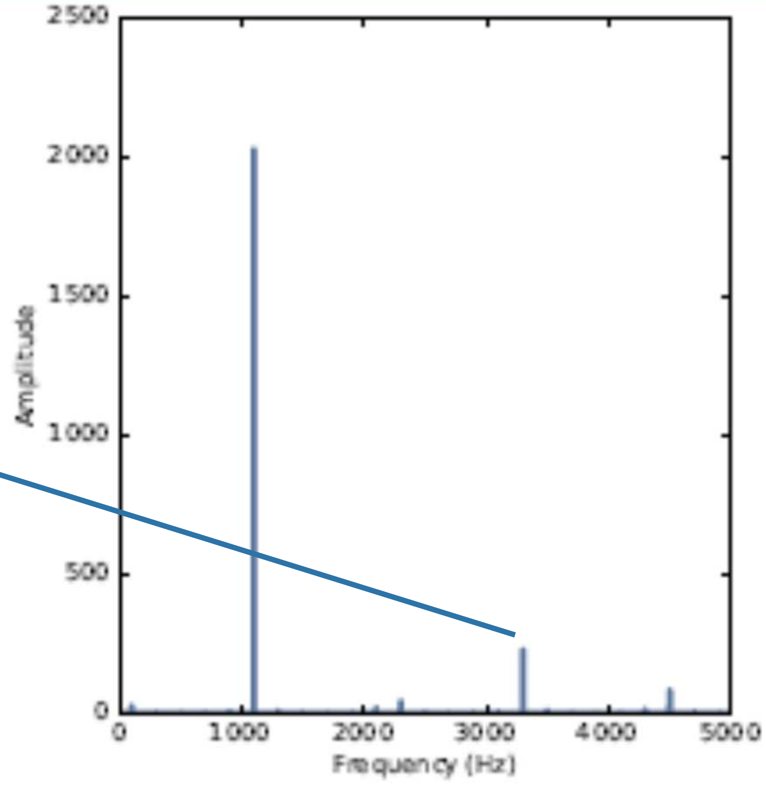
The amplitude drops in proportion to freq. (slower than triangle waves)

Figure 2.4: Spectrum of a square signal at 100 Hz.

Aliasing I

- The previous examples chose parameters carefully.
  - Now, we meets somewhat different spectrum structure.
- The spectrum of a triangle wave at 1100Hz, sampled at 10,000 frames per second.

The harmonics should be at 3300, 5500, 7700 and 9900 Hz. But there are no harmonics at 5500, 7700 and 9900 Hz.

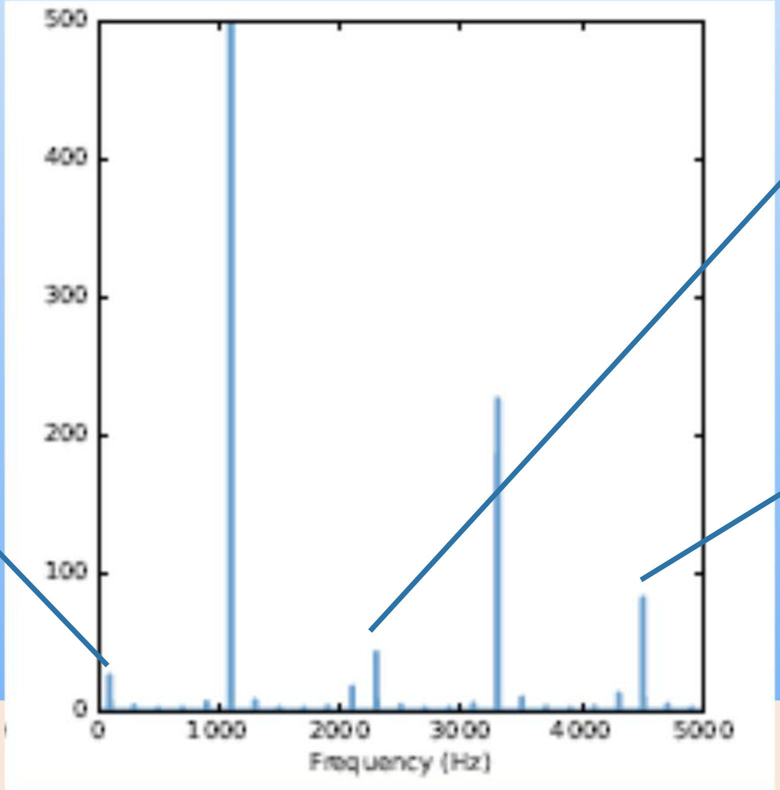


scaled

Aliasing 2

- The spectrum of a triangle wave at 1100Hz, sampled at 10,000 frames per second.

The peak that should be at 9900 is actually at 100Hz.



The fourth is at 2300, not 7700.

The third peak is at 4500, not 5500.

Aliasing 3

- The problem is that when you evaluate the signal at discrete points in time, you lose information about what happened between samples.
  - For low freq. components, that's not a problem because you have lots of samples per period.
- A signal at 5000Hz is sampled 10,000 frame/sec.
  - Only two samples per period : just barely enough
  - Higher frequency will lose information.

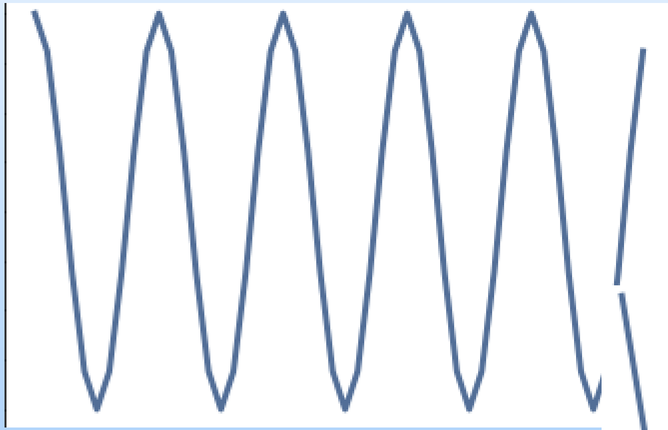
Aliasing 4

- Let's generate cosine signals at 4500 and 5500 Hz, and sample them at 10,000 frame/sec.

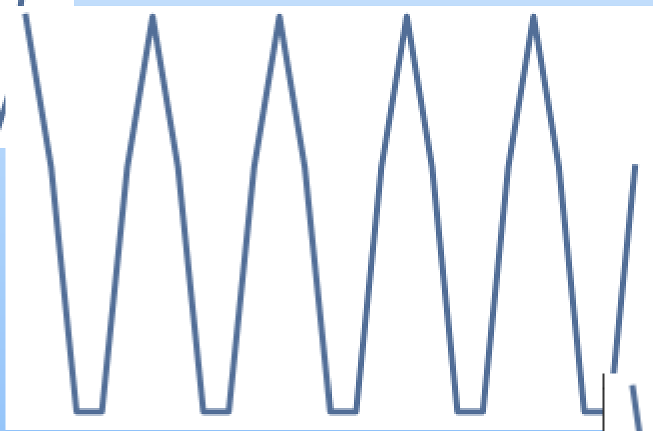
```
signal = thinkdsp.CosSignal(4500)
duration = signal.period*5
segment = signal.make_wave(duration, framerate=framerate)
segment.plot()

signal = thinkdsp.CosSignal(5500)
segment = signal.make_wave(duration, framerate=framerate)
segment.plot()
```

Sampling effect

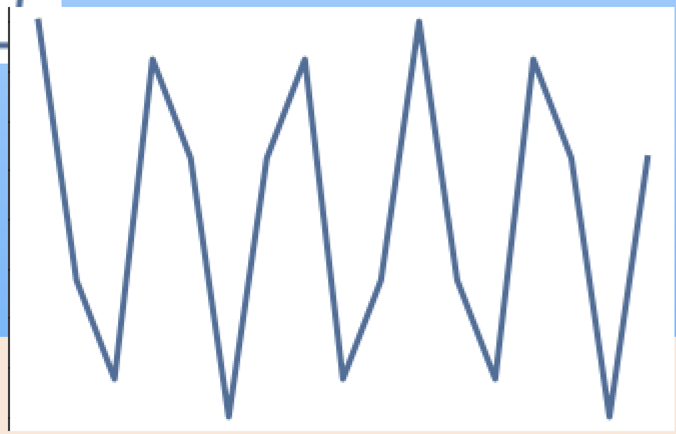


Cosine signal at 1000Hz
and 10,000 framerate



Cosine signal at 2000Hz
and 10,000 framerate

Cosine signal at 3000Hz
and 10,000 framerate



Why aliasing occurs?

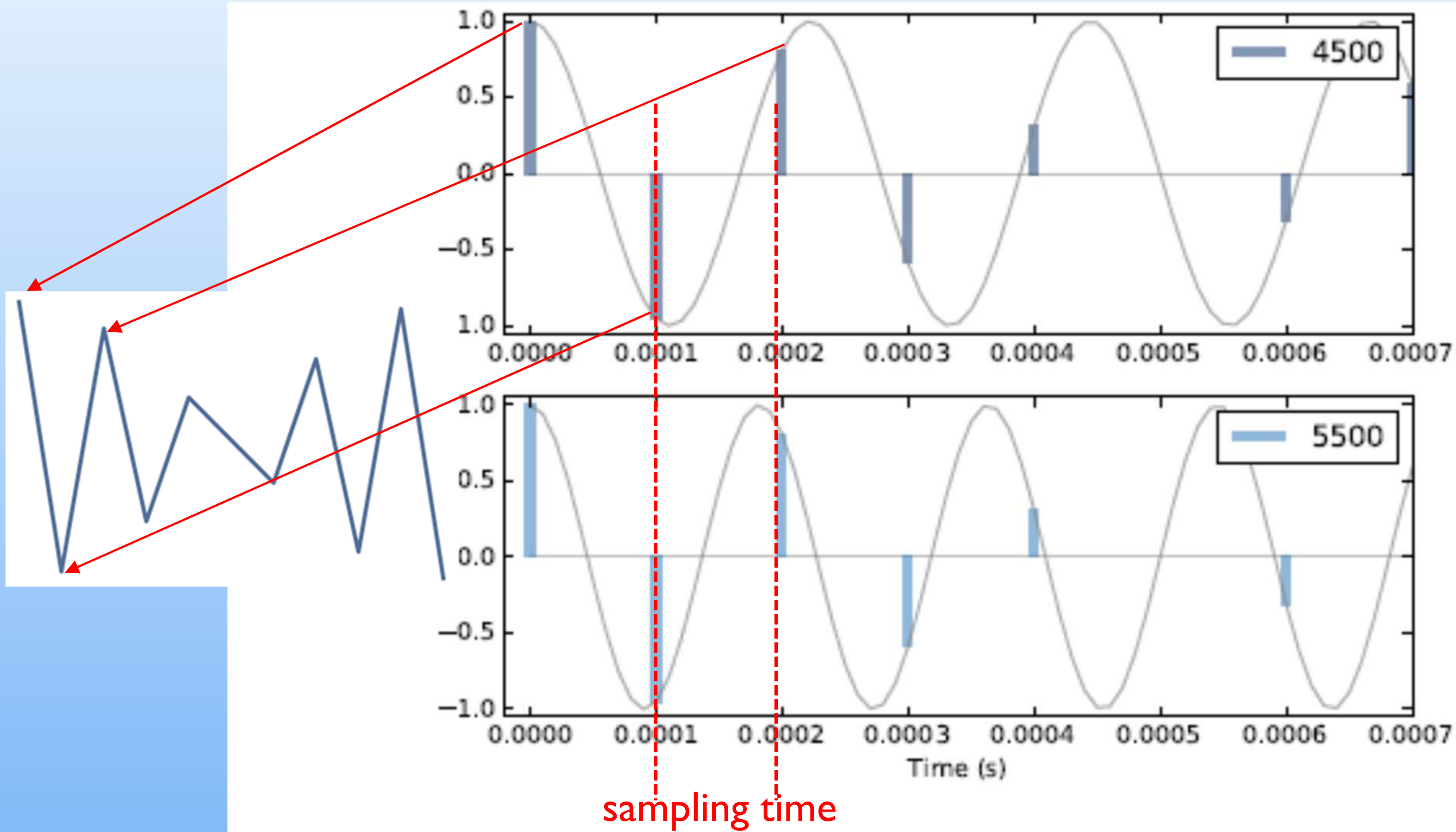


Figure 2.6: Cosine signals at 4500 and 5500 Hz, sampled at 10,000 frames per second. The signals are different, but the samples are identical.

Nyquist frequency

- The effect that a signal appears to be a low frequency signal when the high frequency signal is sampled.
  - 4500 Hz = 5500 Hz
  - 7700 Hz = 2300 Hz
  - 9900 Hz = 100 Hz
- In this example, the highest freq. we can measure is 5000Hz, which is half the sampling rate (Nyquist frequency).
  - Frequencies above 5000Hz are folded back below 5000Hz.
  - The pattern continues if the aliased freq. goes below zero
    - Example>
    - 5th harmonics is at 12,100Hz.
    - Folded at 5,000Hz : -2,100Hz
    - Folded at 0Hz : 2,100Hz
    - So, the 5th harmonics appears at 2,100Hz
    - Similarly, 6th is at 4,300Hz.

Computing the spectrum

- Implementation of make_spectrum( ) method.

```
from np.fft import rfft, rfftfreq

# class Wave:
    def make_spectrum(self):
        n = len(self.ys)
        d = 1 / self.framerate

        hs = rfft(self.ys)
        fs = rfftfreq(n, d)

        return Spectrum(hs, fs, self.framerate)
```

The parameter `self` is a Wave object. `n` is the number of samples in the wave, and `d` is the inverse of the frame rate, which is the time between samples.

`np.fft` is the NumPy module that provides functions related to the **Fast Fourier Transform** (FFT), which is an efficient algorithm that computes the Discrete Fourier Transform (DFT).

Fast Fourier Transform

- rfft (real FFT) : can be used when the wave contains only real values, not complex.
- hs : a NumPy array of complex numbers that represents the amplitude and phase offset of each frequency component in the wave.
- fs : an array that contains frequencies corresponding to the hs.
- Complex number
  - The sum of a real part and an imaginary part
  - $x + iy, \mathrm{i} = \sqrt{-1}$ (cartesian coordinates)
  - $Ae^{i\emptyset}$, A : magnitude and $\emptyset$: angle in radian (polar coordinates)
- Example
  - >>> signal = np.array([-2, 8, 6, 4, 1, 0, 3, 5, -3, 4], dtype=float)
  - >>> fourier = np.fft.rfft(signal)
  - >>> n = signal.size
  - >>> sample_rate = 100
  - >>> freq = np.fft.fftfreq(n, d=1./sample_rate)
  - >>> freq
  - array([0., 10., 20., 30., 40., -50., -40., -30., -20., -10.])
  - >>> freq = np.fft.rfftfreq(n, d=1./sample_rate)
  - >>> freq
  - array([0., 10., 20., 30., 40., 50.])

Spectrum manipulation

- To modify s spectrum, you can access the hs directly.
  - To double the amplitude
    - spectrum.hs *=2
    - spectrum.scale(2)
  - To remove frequencies that exceeds some cutoff freq.
    - spectrum.hs[spectrum.fs > cutoff] = 0
    - spectrum.low_pass(cutoff)
- Don't worry about FFT up to now. We will cover it later on.