Digital Signal Processing

Lecture 5 – Non-periodic signals

상명대학교
컴퓨터과학과
강상욱 교수

Non-periodic signals

- Frequency components DO change over time.
- Spectrogram : a common way to visualize non-periodic signals.

Linear chirp I

- Chirp : A sinusoid that sweeps linearly through a range of frequencies.
- A chirp that sweeps from 220 Hz (A3) to 880 Hz (A5)
  - 참조 : http://www.korea.kr/celebrity/cultureColumnView.do?newsId=148825273

```
signal = thinkdsp.Chirp(start=220, end=880)
wave = signal.make_wave()
```

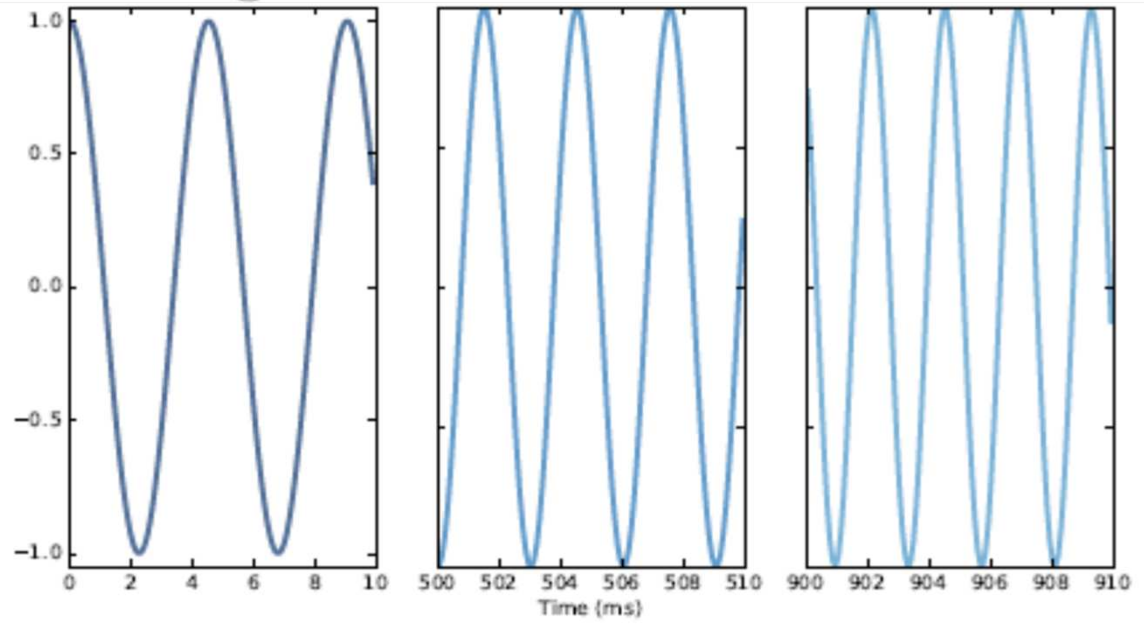


Figure 3.1: Chirp waveform near the beginning, middle, and end.

Linear chirp 2

```
class Chirp(Signal):

    def __init__(self, start=440, end=880, amp=1.0):
        self.start = start
        self.end = end
        self.amp = amp
```

start and end are the frequencies, in Hz, at the start and end of the chirp. amp is amplitude.

A private method
Beginning a method name with an underscore makes it "private"

Returns a Numpy array of evenly spaced numbers over the interval [start, stop]

```
def evaluate(self, ts):
    freqs = np.linspace(self.start, self.end, len(ts)-1)
    return self._evaluate(ts, freqs)
```

Linear chirp 3

```
    def _evaluate(self, ts, freqs):
        dts = np.diff(ts)
        dphis = PI2 * freqs * dts
```

Computes the difference between adjacent elements of ts.
If the elements of ts are equally spaced, the dts are all the same.

Figure out how much the phase changes during each interval
$\Delta\emptyset = 2\pi f(t)\Delta t$

```
        phases = np.cumsum(dphis)
        phases = np.insert(phases, 0, 0)
        ys = self.amp * np.cos(phases)
        return ys
```

Calculate the total phase at each timestep by adding up the changes.

To add 0 at the beginning

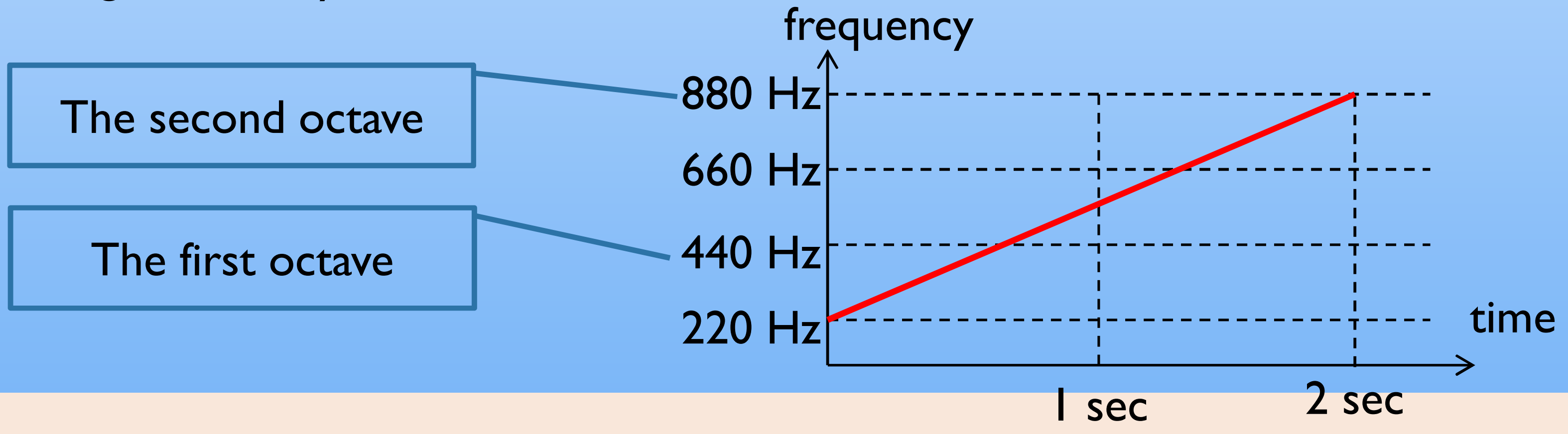
np.cos( ) computes the amplitude of the wave as a function of phase.
$d\emptyset = 2\pi f(t)dt$: phase is the integral of freq.
$d\emptyset/df = 2\pi f(t)$: freq. is the derivative of phase

Example

```
>>> freqs = np.linspace(10, 60, 5)
>>> ts = [0, 0.1, 0.2, 0.3, 0.4, 0.5]
>>> dts = np.diff(ts)
>>> dts
array([0.1, 0.1, 0.1, 0.1, 0.1])
>>> dphis = 2*3.14*freqs*dts
>>> dphis
array([ 6.28, 14.13, 21.98, 29.83, 37.68])
>>> phases = np.cumsum(dphis)
>>> phases
array([   6.28,   20.41,   42.39,   72.22, 109.9 ])
>>> ys = 1 * np.cos(phases)
>>> ys
array([ 0.99999493,  0.01035206, -
0.02149917, -0.99932916, -0.99844677])
```

Exponential chirp I

- When you listen to the previous chirp, you might notice that the pitch rises quickly at first then slows down.
  - This chirp spans two octaves, but it only takes 2/3 seconds to span the first octave, and twice as long to span the second.
  - Human perception of pitch depends on the logarithm of frequency.
  - The interval we hear between two notes depends on the ratio of their frequencies.
  - As a result, if freq. increases linearly, the perceived pitch increases logarithmically.



Exponential chirp 2

- If you want the perceived pitch to increase linearly, the freq. has to increase exponentially. This is called an "exponential chirp".

```
class ExpoChirp(Chirp):

    def evaluate(self, ts):
        start, end = np.log10(self.start), np.log10(self.end)
        freqs = np.logspace(start, end, len(ts)-1)
        return self._evaluate(ts, freqs)
```

Creates a series of frequencies whose logarithms are equally spaced.
The freq. increases exponentially.

```
signal = thinkdsp.ExpoChirp(start=220, end=880)
wave = signal.make_wave(duration=1)
```

Example

```
>>> s, e = np.log10(100), np.log10(1600)
>>> s,e
(2.0, 3.2041199826559246)
>>> freq = np.logspace(s, e, 5)
>>> freq
array([  100.,   200.,   400.,   800.,  1600.])
```

Spectrum of a chirp

- The spectrum of a one-second and one-octave chirp.

```
signal = thinkdsp.Chirp(start=220, end=440)
wave = signal.make_wave(duration=1)
spectrum = wave.make_spectrum()
```

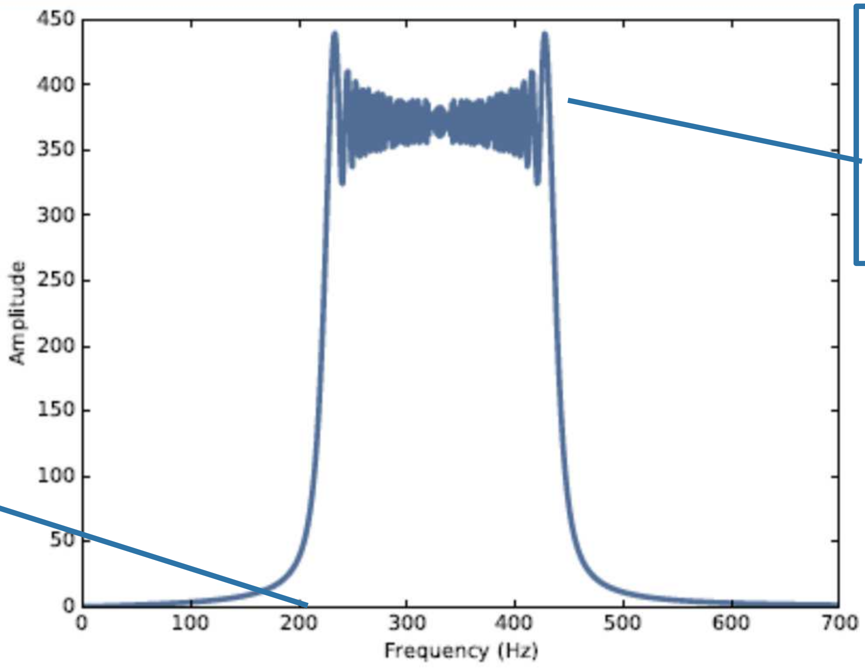


Figure 3.2: Spectrum of a one-second one-octave chirp.

The spectrum has components at every freq. from 220 to 440 Hz

The spectrum is approximately flat between 220 and 440 Hz, which indicates that the signal spends equal time at each freq.

Based on this observation, guess what the spectrum of an exponential chirp looks like?

The spectrum gives hints about the structure of the signal but it obscures the relationship between freq. and time.

Spectrogram I

- Short-time Fourier Transform (STFT)
  - To recover the relationship between freq. and time, the chirp is broken into segments and transformed to the spectrum.
- Spectrogram shows time on the x-axis and freq. on the y-axis.

Each column in the spectrogram shows the spectrum of a short segment, using color or grayscale to represent amplitude.

The freq. increases linearly over time. However, notice that the peak in each column is blurred across 2-3 cells.
The blurring reflects the limited resolution

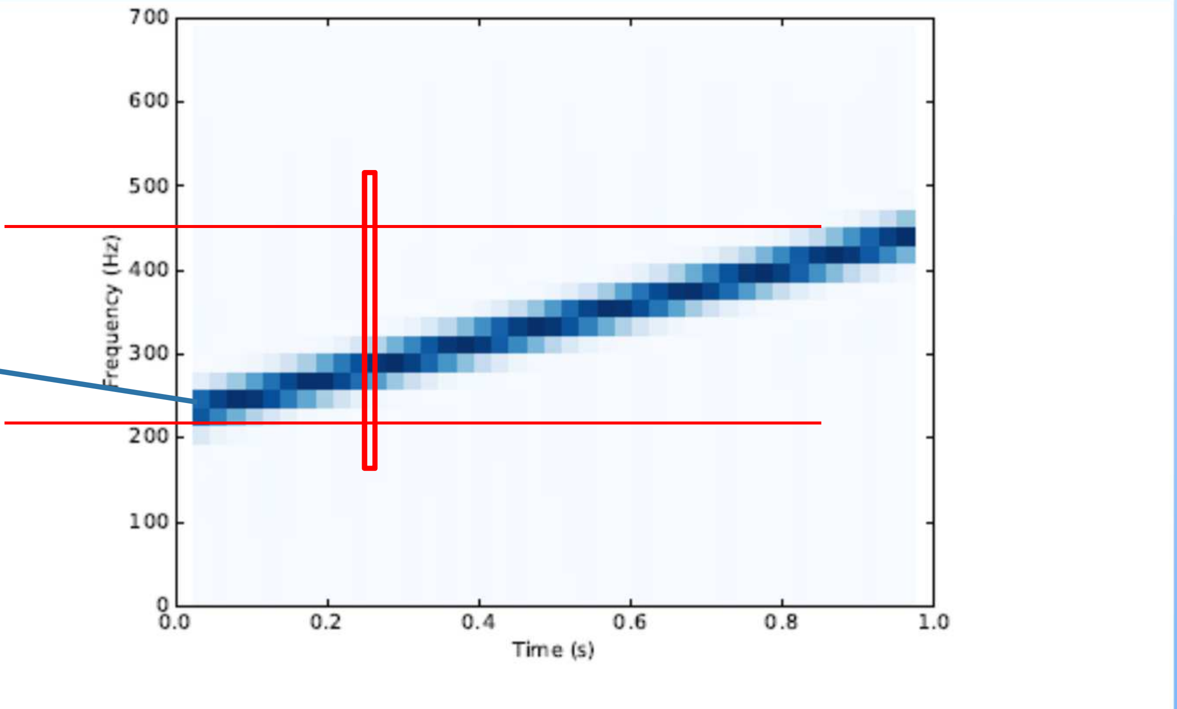
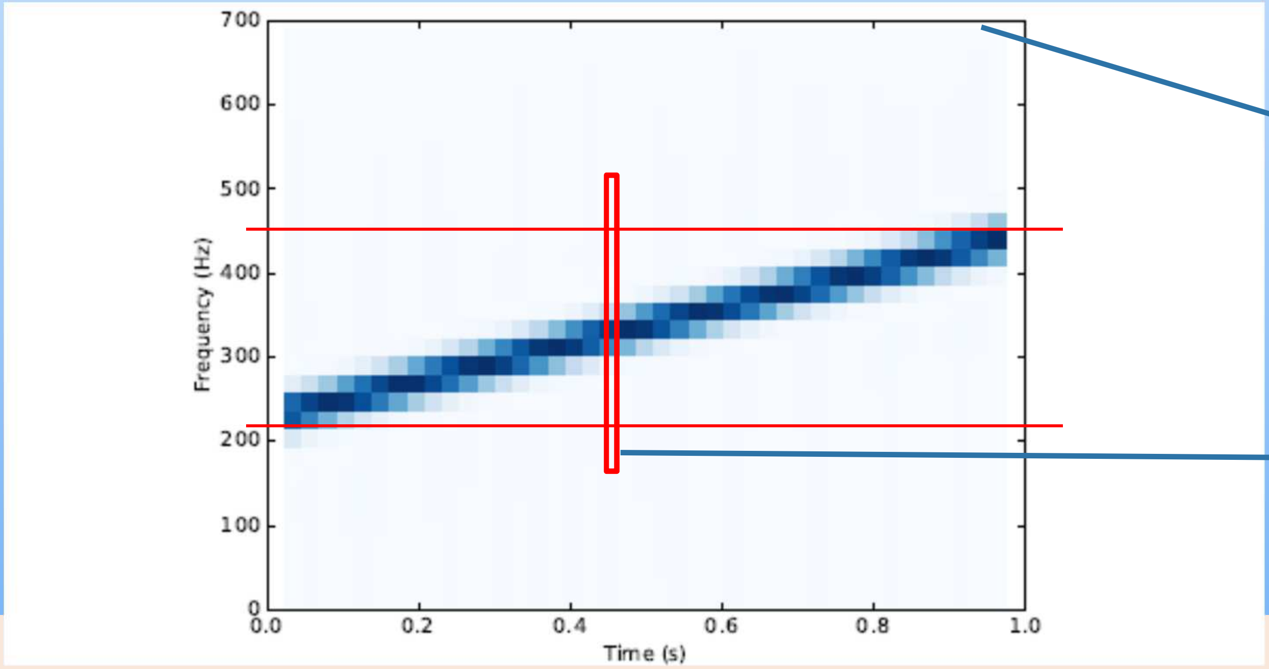


Figure 3.3: Spectrogram of a one-second one-octave chirp.

Spectrogram 2

```
signal = thinkdsp.Chirp(start=220, end=440)
wave = signal.make_wave(duration=1, framerate=11025)
spectrogram = wave.make_spectrogram(seg_length=512)
spectrogram.plot(high=700)
```

seg_length is the number of samples in each segment. I chose 512 because FFT is most efficient when the number of samples is a power of 2.



The full range goes to 5512.5 Hz, which is half of framerate.

The duration is 512/11025 seconds.

The Gabor limit I

- Time resolution : the duration of the segments, which corresponds to the width of the cells in the spectrogram.
  - At figure 3.3 : time resolution is $\frac{512\ frames}{11025\ frames/sec} = 0.046\ sec.$
- Frequency resolution : the freq. range between elements in the spectrum, which corresponds to the height of the cells.
  - At figure 3.3 : With 512 frames, we get 256 freq. components over a range from 0 to 5512.5Hz.
  - So the range between components is $\frac{5512.5\ Hz}{256} = 21.53Hz$.
- Generally, the freq. resolution is $\frac{r/2}{n/2} = r/n.$
  - n : the segment length
  - r : the frame rate
  - Time resolution is n/r

The Gabor limit 2

- Ideally, we'd like time resolution to be small, so we can see rapid changes in freq.
  - And we'd like freq. resolution to be small so we can see small changes in freq.
- In practical, time resolution is the inverse of freq. resolution.
  - If you double the segment length, you cut freq. resolution in half.
  - Even increasing the frame rate doesn't help. You get more samples, but the range of frequencies at the same time.
- This tradeoff is called the Gabor limit.

Leakage I

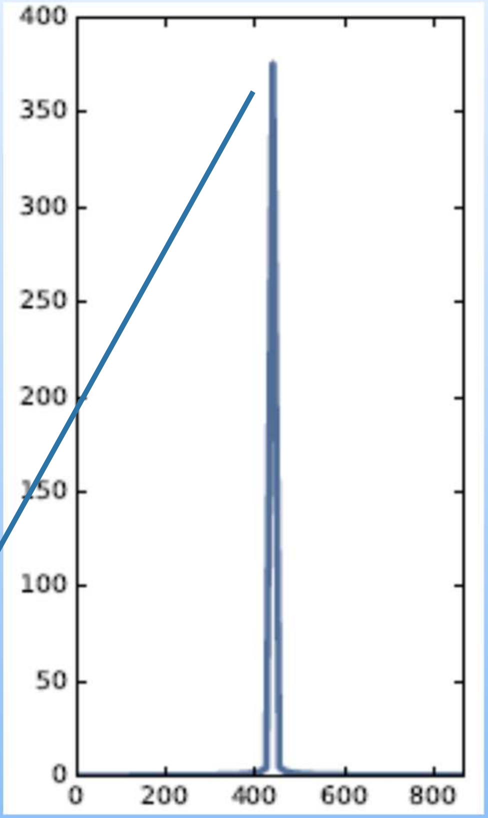
- The DFT treats waves as if they are periodic.
  - DFT assumes that the finite segment it operates on is a complete period from an infinite signal that repeats over all time.
  - Of course, this assumption is often false, which creates problems.
- Discontinuity at the beginning and end of the segment.
  - The end of the segment does not connect smoothly to the beginning.
  - The discontinuity creates additional freq. components in the segment that are not in the signal.

Leakage 2

```
signal = thinkdsp.SinSignal(freq=440)
duration = signal.period * 30
wave = signal.make_wave(duration)
spectrum = wave.make_spectrum()
```

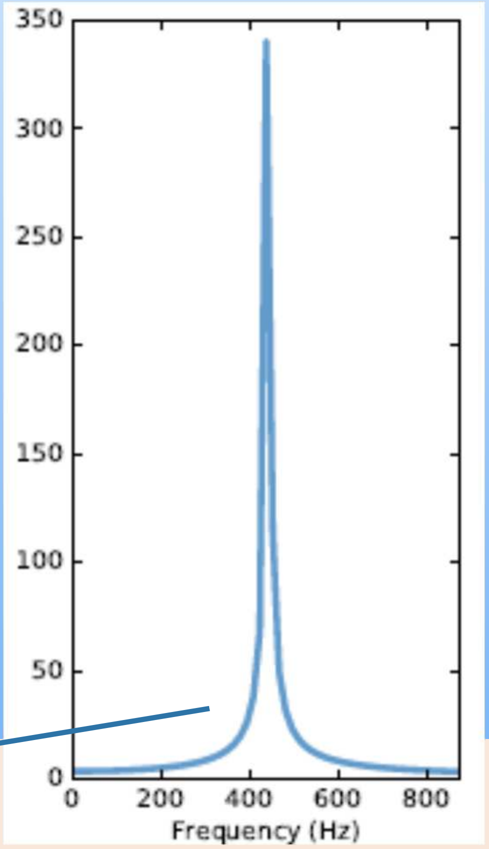
A segment that is an integer multiple of the period.

The spectrum has a single peak at 440 Hz.



```
duration = signal.period * 30.25
```

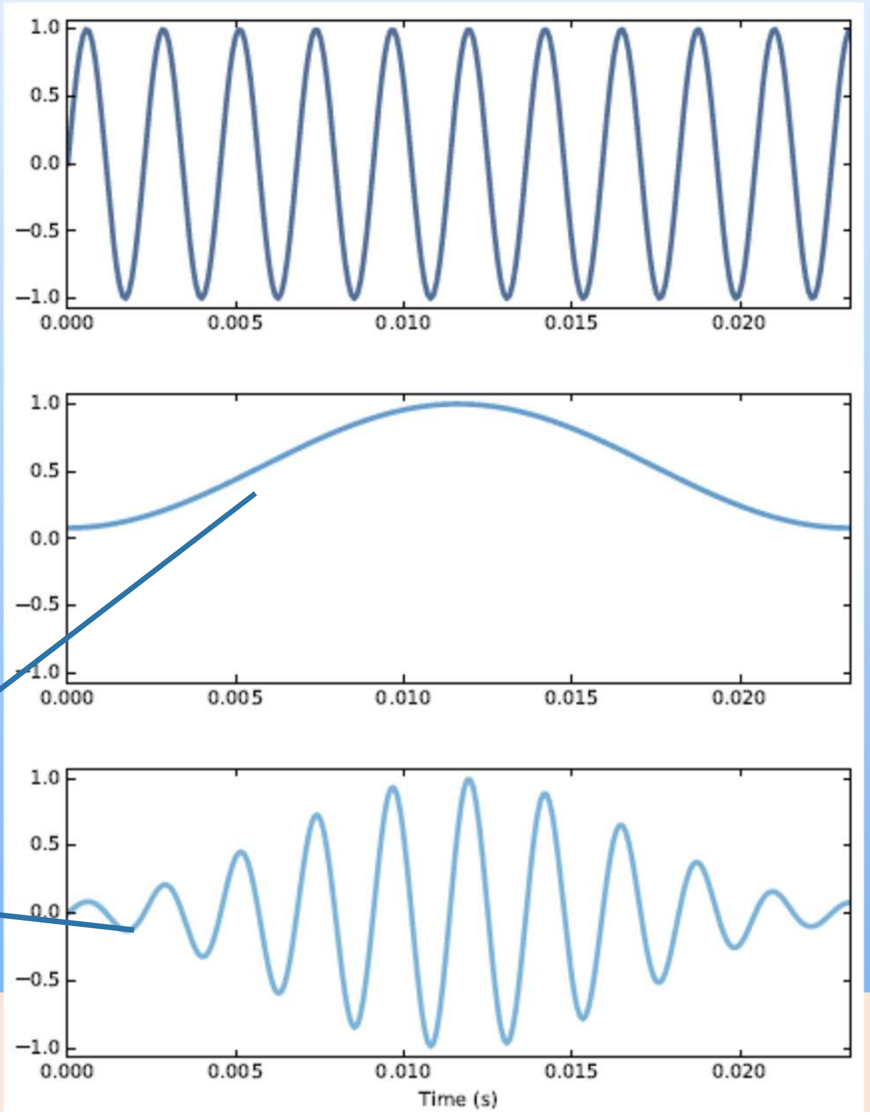
A segment that is NOT an integer multiple of the period.



The peak is still at 440 Hz, but now there are additional components spread out from 240 to 640.

Windowing I

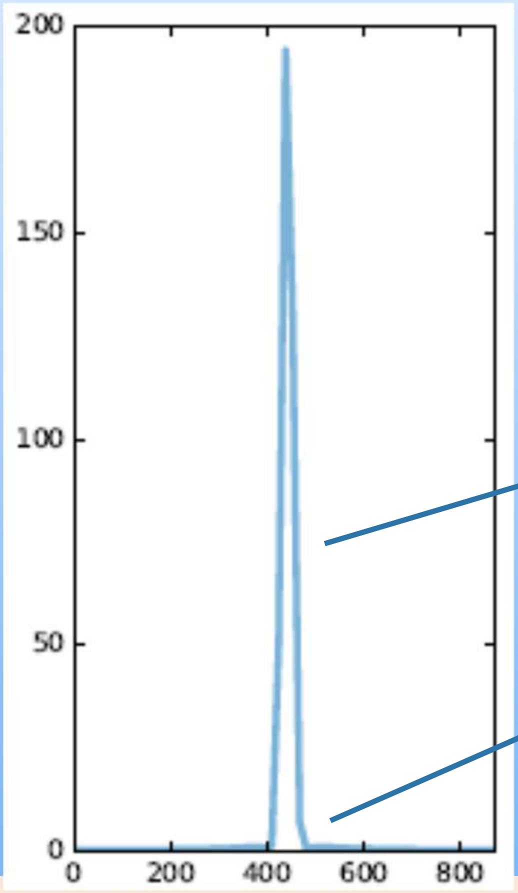
- We can reduce leakage by smoothing out the discontinuity between the beginning and end of the segment using windowing.
- Window : a function designed to transform a non-periodic segment into something that can pass for periodic.



Hamming window

The result of multiplying the window by the original signal. The end of the segment connects smoothly to the beginning.

Windowing 2



```
#class Wave:
    def window(self, window):
        self.ys *= window
```

```
window = np.hamming(len(wave))
wave.window(window)
```

The spectrum of Hamming windowed signal.

Windowing has reduced leakage substantially, but not completely.

NumPy provides other window function, including bartlett, blackman, hanning and kaiser.

Implementing spectrogram I

```
#class Wave:
    def make_spectrogram(self, seg_length):
        window = np.hamming(seg_length)
        i, j = 0, seg_length
        step = seg_length / 2

        spec_map = {}

        while j < len(self.ys):
            segment = self.slice(i, j)
            segment.window(window)

            t = (segment.start + segment.end) / 2
            spec_map[t] = segment.make_spectrum()

            i += step
            j += step
```

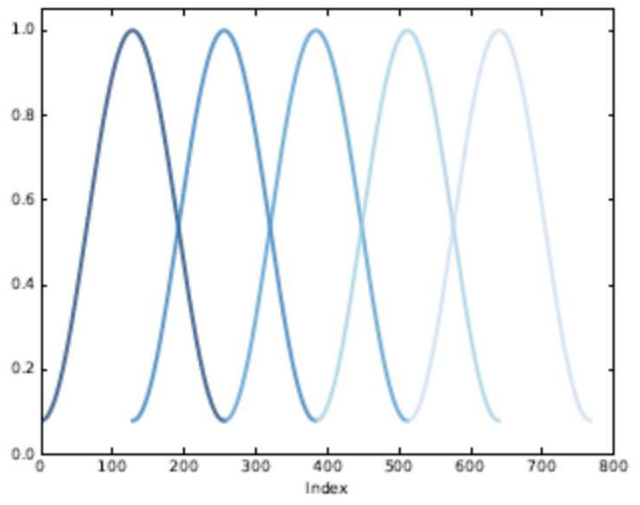
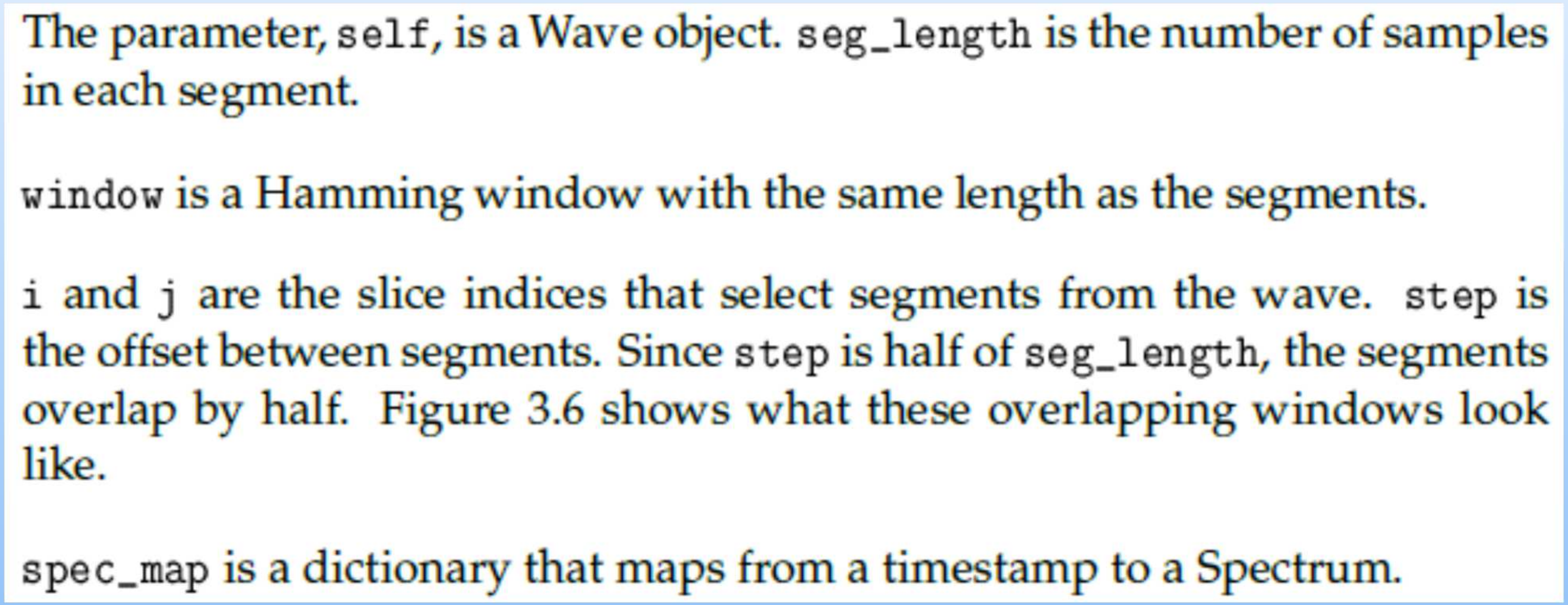


Figure 3.6: Overlapping Hamming windows.

Implementing spectrogram 2



The parameter, `self`, is a Wave object. `seg_length` is the number of samples in each segment.

`window` is a Hamming window with the same length as the segments.

`i` and `j` are the slice indices that select segments from the wave. `step` is the offset between segments. Since `step` is half of `seg_length`, the segments overlap by half. Figure 3.6 shows what these overlapping windows look like.

`spec_map` is a dictionary that maps from a timestamp to a Spectrum.