

Intelligent Signal Processing (CM3065)

Course Notes

Felipe Balbi

June 1, 2021

Contents

Week 1	4
1.004 Technical information and tools used in this module	4
1.101 What is sound?	4
1.102 Characteristics of sound waves	4
1.104 Human sound perception	6
1.106 Audio fundamentals	7
1.201 The digital world: sample rate and bit depth	8
1.203 Digital audio representation: the time domain	9
1.205 Digital audio	10
Week 2	11
1.401 Audio programming with p5.js: loading and playing back audio files . . .	11
1.403 Audio programming with p5: playback controls	12
1.404 Audio programming with p5: capture and record audio	13
1.501 Audio programming with Python	15
1.503 Audio in Python	15
Week 3	16
2.104 Digital audio effects	16
2.107 Processing digital audio in p5.js. The p5.Effect class	16
Week 4	18
2.201 Filtering. Implementation of an averaging low-pass filter	18
2.203 FIR (Finite Impulse Response) filters	20
2.205 LTI systems	20
Week 5	22
3.001 Review of spectral analysis	22
3.003 Spectral analysis of periodic signals	22
Week 6	23
3.101 Complex synthesis	23
Week 7	24
4.001 Introduction to audio feature extraction	24
4.003 Audio feature extraction	26
4.101 Real-time audio visualisations with p5.js. – time domain features	26
4.102 Real-time audio visualisations with p5.js – frequency domain features (1)	26

Contents

4.105 Real-time audio visualisations with Meyda	26
4.107 Meyda: an audio feature extraction library for the Web Audio API	26

Week 1

Key Concepts

- Use an audio editor and write simple programs to work with digital audio signals
- Explain the relevance of bit depth and sampling rates for digital audio signals and select appropriate parameters for different types of signals
- Describe the characteristics of sound waves and how they are perceived by humans

1.004 Technical information and tools used in this module

- Audacity
- P5.js
- Python
- ffmpeg

1.101 What is sound?

From a physics perspective, sound is a form of mechanical energy *produced* by vibrating matter. It needs a medium to propagate. In contrast to electromagnetic energy, sound cannot travel in a vacuum.

From a physiological or psychological perspective, sound is the reception of these waves and its processing by the brain. We capture sound waves with our inner ear and the brain converts it into what we call sound.

1.102 Characteristics of sound waves

In figure 1 below, we see several sine waves. Remember that the sine wave equation is given by $f(x) = A \sin(2\pi ft + \phi)$ where A is the amplitude, f is the frequency, t is the time, and ϕ is the phase.

The speed of sound waves depends on the temperature, elasticity, and density of the medium the sound is travelling through. As an example, at 0 °C the speed of sound travelling through the air is around 331 m s⁻¹. When the air is around 20 °C the speed of sound is around 343 m s⁻¹.

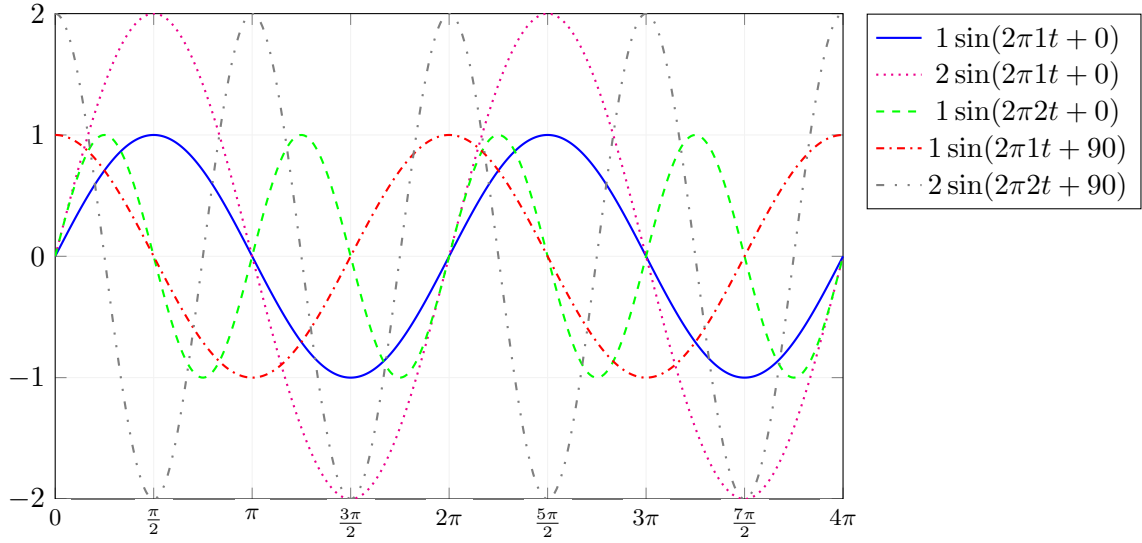


Figure 1: Sine waves

Waves can be represented in the domain of space. In this scenario we can characterize the wave by its wavelength and amplitude. The wavelength (λ) is given by the distance between two subsequent crests or troughs on the wave. The amplitude is given by the distance from the x axis to a peak or trough of the wave. All of these can be seen in figure 2 below. The pressure is measured in pascals (Pa) although we use dB for measuring sound amplitude.

Waves can also be represented in the domain of time. In this scenario we can characterize the wave by its amplitude, frequency, and period as shown in figure 3 below. Note that frequency, measured in hertz (Hz), is given by the inverse of the period, i.e. $f = \frac{1}{T}$ where T is the period in seconds.

There is a direct relation between sound speed, wavelength, and frequency given by:

$$v = f \cdot \lambda$$

For example, if the wavelength is $\lambda = 2 \text{ m}$, $v = 4 \text{ m s}^{-1}$, then:

$$\begin{aligned} 4 \text{ m s}^{-1} &= f \cdot 2 \text{ m} \\ f &= \frac{4 \text{ m s}^{-1}}{2 \text{ m}} \\ f &= 2 \text{ Hz} \end{aligned}$$

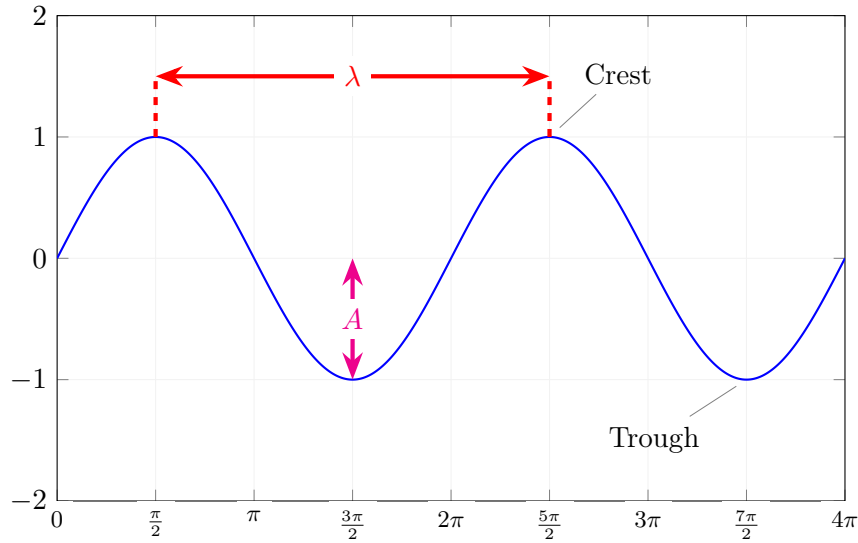


Figure 2: Sine wave in space domain

1.104 Human sound perception

Humans perceive sound through *pitch*, *loudness*, and *timbre*. In table 1 we summarize some of the properties of sound, both physical and psychological.

Table 1: Physical vs Psychological Properties of Sound

Physical	Psychological
Frequency	Pitch
Amplitude	Loudness
Waveform	Timbre
Wavelength	
Period	
Duration	

We can say that there is a relation between the matching pairs in table 1. That is, *Frequency* is related to *Pitch*, *Amplitude* is related to *Loudness*, and *Waveform* is related to *Timbre*. The relation, however, is not linear. That is, a certain change in frequency does not correspond to a similar change in pitch.

What is said above is a simplification of reality. A change in frequency also has an impact in loudness and timbre.

Pure Tone is a sound wave composed of a single sine wave. In figure 4 we have a representation of a pure tone while in figure 5 we have a representation of a composition of several sine waves.

The relationship between Pitch and Frequency is non-linear. In practice this means that as the pitch increases, a greater change in frequency is required. Another impor-

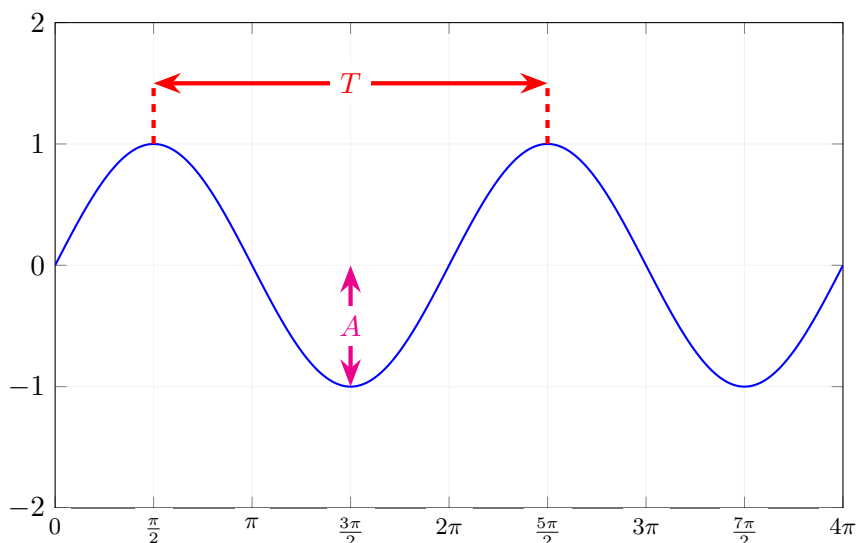


Figure 3: Sine wave in time domain

tant details is that humans can hear sound within the range from 20 Hz to 20 000 Hz. Frequencies below 20 Hz are known as infra-sound and frequencies above 20 000 Hz are known as ultrasound.

Loudness is a sensation to the perception of amplitude in sound waves. The pressure is measured in pascals (Pa). A quiet bedroom at night would measure sound pressure on the order of 630 μPa .

It turns out that measuring sound pressure in pascals is not very convenient, because of that we generally use a logarithmic scale called *dB SPL*. The conversion for pascal to dB is given by:

$$SPL = 20 \log_{10} \left(\frac{p}{p_{ref}} \right) \text{ dB}$$

Where $p_{ref} = 20 \mu\text{Pa}$, for sound pressure in air.

Timbre, or sound quality, helps us differentiate between two sounds with the same frequency and amplitude. Timbre is related to the waveform and the sound spectrum. In summary, we can hear the different component sound waves of a sound and interpret it as timbre.

1.106 Audio fundamentals

- Hosken, D. W. An introduction to music technology. (New York: Routledge, 2011).
– Chapter 1: What is sound (pp.7–9) – Chapter 2: Sound properties and the waveform view (pp.17–26)

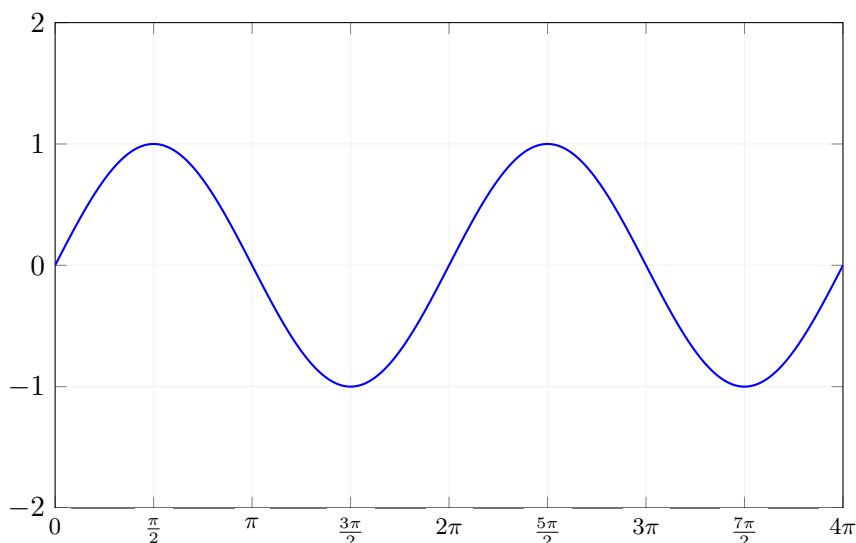


Figure 4: Pure Tone

1.201 The digital world: sample rate and bit depth

A sound level meter uses a microphone to convert air pressure changes into an electrical signal measured in volts. This is the first step to digitize a sound wave.

The signal output of a microphone is very small, therefore we need to amplify the signal before passing it along to an Analog to Digital Converter, or ADC. The ADC measures the incoming voltage periodically and converts each sample to a numerical value within its dynamic range.

The sampling rate is the rate at which the ADC is *sampling* the input voltage. The sampling rate is given in Hertz (Hz). Each measurement of the input signal's amplitude is called a *sample*, the faster we sample, the better the quality, but the resulting amount of data per second will be larger. Table 2 below gives a set of common sampling rates and where they're used.

Table 2: Common Sampling Rates

Application	Sample Rate
CD	44.1 kHz
DVD	48 kHz
Professional Audio (?)	88.2 kHz
Professional Audio (?)	96 kHz

The Nyquist-Shannon Sampling Theorem states that the sampling rate must be at twice the highest frequency to be sampled. In other words, we can say that the Nyquist Frequency is $\frac{\text{samplerate}}{2}$ and any frequency above the Nyquist Frequency will not be recorded properly by the ADC, introducing artificial frequencies through aliasing.

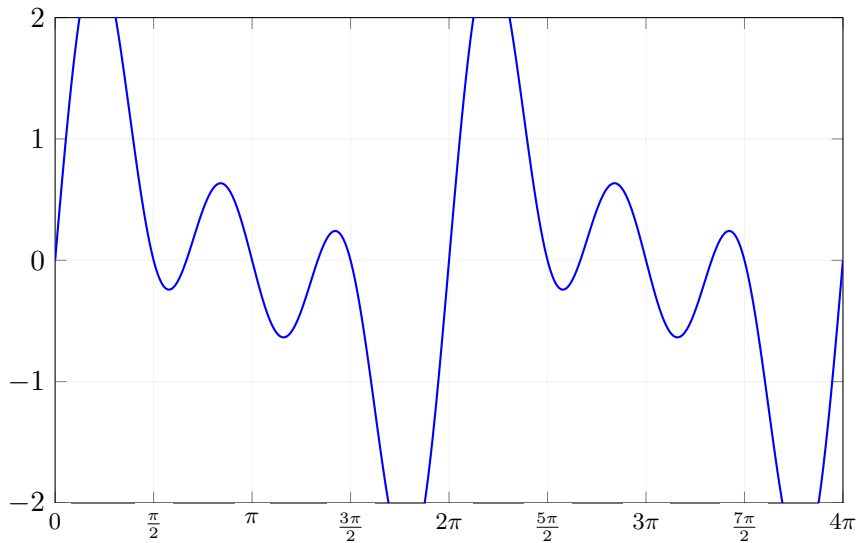


Figure 5: Composite Tone

To counter the problem of aliasing, most audio systems including an anti-aliasing frequencies which are basically a low-pass filter that cuts off frequencies above the Nyquist Frequency of the system.

Bit Depth is the number of bits used to record the samples. It's also referred to as Dynamic Range, sample width, or quantization level. The more bits we use, the more accurately we can measure the analogue waveform, but it results in more memory usage per sample. Common bit widths for digital audio are 8, 16, 24, and 32 bits.

Clipping occurs when the sampled amplitude is outside the ADC's dynamic range, then the value will either be clipped to 0 or maximum allowed value. Clipping generates perceivable distortions in the audio and should be avoided.

1.203 Digital audio representation: the time domain

There are two ways to represent audio:

Time Domain plotted as amplitude vs time

Frequency Domain plotted as amplitude vs frequency

Amplitude is usually represented in *dBFS* or db Full Scale. The relation is given by:

$$dBFS = 20 \log_{10}(|value|)$$

1.205 Digital audio

- Hosken, D. W. An introduction to music technology. (New York: Routledge, 2011).
– Chapter 4: Analog and digital (p.51) – Chapter 4: The audio recording path (pp.51–53) – Chapter 5: Digital audio data (pp.72–85) – Chapter 6: Recording (pp.91–94)
- Smith, S. Digital signal processing: a practical guide for engineers and scientists. (Burlington, MA: Elsevier, 2013).

Week 2

Key Concepts

- Use an audio editor and write simple programs to work with digital audio signals
- Explain the relevance of bit depth and sampling rates for digital audio signals and select appropriate parameters for different types of signals
- Describe the characteristics of sound waves and how they are perceived by humans

1.401 Audio programming with p5.js: loading and playing back audio files

A simple example to load an audio file into p5.js is shown below:

```
1  let mySound;
2  let playStopButton;
3
4  function preload() {
5    soundFormats('mp3', 'wav')
6    mySound = loadSound('sound')
7  }
8
9  function playStopSound() {
10   if (mySound.isPlaying()) {
11     mySound.stop();
12     /* mySound.pause() can be used */
13     playStopButton.html('play')
14   } else {
15     mySound.play()
16     playStopButton.html('stop')
17   }
18 }
19
20 function setup() {
21   createCanvas(400, 200)
22   background(180)
23 }
```

```

24   playStopButton = createButton('play')
25   playStopButton.position(20, 20)
26   playStopButton.mousePressed(playStopSound)
27 }
28
29 function draw() {
30 }

```

1.403 Audio programming with p5: playback controls

Augmenting the previous application with some controls, we get:

```

1  let mySound;
2  let playStopButton;
3  let jumpButton;
4  let sliderVolume;
5  let sliderRate;
6  let sliderPan;
7
8  function preload() {
9    soundFormats('mp3', 'wav')
10    mySound = loadSound('sound')
11  }
12
13  function playStopSound() {
14    if (mySound.isPlaying()) {
15      mySound.stop();
16      /* mySound.pause() can be used */
17      playStopButton.html('play')
18    } else {
19      mySound.play()
20      playStopButton.html('stop')
21    }
22  }
23
24  function jumpSound() {
25    let dur = mySound.duration()
26    let t = random(dur)
27    mySound.jump(t)
28  }
29
30  function setup() {
31    createCanvas(400, 200)
32    backgroun(180)

```

```

33
34   playStopButton = createButton('play')
35   playStopButton.position(200, 20)
36   playStopButton.mousePressed(playStopSound)
37
38   jumpButton = createButton('jump')
39   jumpButton.position(250, 20)
40   jumpButton.mousePressed(jumpSound)
41
42   sliderVolume = createSlider(0, 2, 1, 0.01)
43   sliderPosition(20, 25)
44   text('volume', 80, 20)
45
46   sliderRate = createSlider(-2, 2, 1, 0.01)
47   sliderRate.position(20, 70)
48   text('rate', 80, 65)
49
50   sliderPan = createSlider(-1, 1, 0, 0.01)
51   sliderPan.position(20, 115)
52   text('pan', 80, 110)
53 }
54
55 function draw() {
56   mySound.setVolume(sliderVolume.value())
57   mySound.rate(sliderRate.value())
58 }

```

1.404 Audio programming with p5: capture and record audio

We can also use p5.js to capture audio, an example is shown below.

```

1  let mic
2  let recorder
3  let soundFile
4  let state = 0
5
6  function setup() {
7    createCanvas(400, 400)
8
9    mic = new p5.AudioIn()
10   mic.start()
11
12   recorder = new p5.SoundRecorder()
13   recorder.setInput(mic)

```

```

14
15   soundFile = new p5.SoundFile()
16 }
17
18 function draw() {
19   background(200);
20
21   let vol = mic.getLevel()
22
23 }
24
25 function mouseClicked() {
26   if (getAudioContext().state !== 'running')
27     getAudioContext().resume()
28
29   switch (state) {
30     case 0:
31       background(255, 0, 0)
32       text('Recording', 40, 40)
33
34       recorder.record(soundFile)
35
36       state++
37       break
38     case 1:
39       background(0, 255, 0)
40       text('Click to play and download!', 40, 40)
41
42       recorder.stop()
43
44       state++
45       break
46     case 2:
47       background(200)
48       text('Click to record', 40, 40)
49
50       soundFile.play()
51       save(soundFile, 'output.wav')
52
53       state = 0
54       break
55   }
56 }

```

1.501 Audio programming with Python

Python has useful libraries for sound processing. The Python Audio Wiki has interesting information about these libraries. The Python In Music page also contains important information related to audio processing with Python.

The libraries we will use are:

- NumPy
- SciPy
- matplotlib
- ThinkDSP

1.503 Audio in Python

- Downey, A.B. Think DSP - Digital Signal Processing in Python. (Needham: Green Tea Press, 2014).

Week 3

Key Concepts

- Define homogeneity, additivity and shift invariance as they relate to linear time invariant systems
- Carry out convolution by hand and in code and analyse the process in terms of impulse responses and the properties of linear time invariant systems
- Use and write software to edit and to apply a range of effects processors to digital audio signals

2.104 Digital audio effects

- Audacity 'Index of effects, generators, analyzers and tools' (2021).
- Collins, N. Introduction to computer music. (Chichester: John Wiley & Sons, 2010).
 - Chapter 4: A compendium of marvellous digital audio effects (pp.143–155)

2.107 Processing digital audio in p5.js. The p5.Effect class

The simplest p5js to play audio is shown below.

```
1  let player;
2
3  function preload() {
4    player = loadSound("sound.wav");
5  }
6
7  function setup() {
8    let canvas = createCanvas(200, 200);
9    background(0, 255, 0);
10   canvas.mousePressed(mousePressed);
11 }
12
13 function mousePressed() {
14   player.loop();
15 }
```


It builds the simple audio pipeline depicted in figure 6.

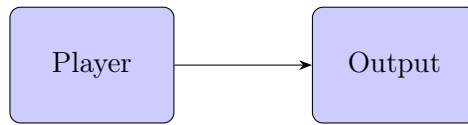


Figure 6: Simple Pipeline

When we want to add effects to the audio, we simply insert boxes between *Player* and *Output*. For example, for a reverb effect, our modified pipeline would look like the one shown in figure 7.

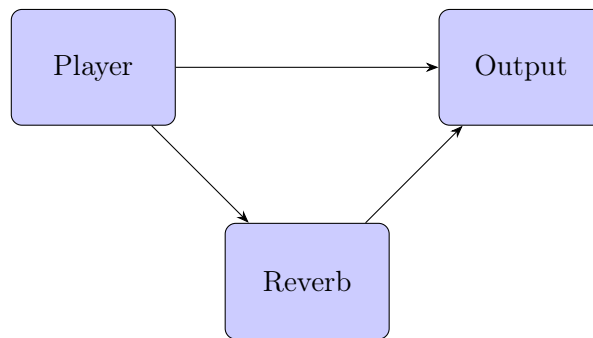


Figure 7: Pipeline with Reverb

Now we just modify our original code to include a reverb object. As shown below.

```

1  let player;
2  let reverb;
3
4  function preload() {
5    player = loadSound("sound.wav");
6    reverb = new p5.Reverb();
7  }
8
9  function setup() {
10   let canvas = createCanvas(200, 200);
11   background(0, 255, 0);
12   canvas.mousePressed(mousePressed);
13   reverb.process(player, 2, 3);
14 }
15
16 function mousePressed() {
17   player.loop();
18 }
  
```

Week 4

Key Concepts

- Define homogeneity, additivity and shift invariance as they relate to linear time invariant systems
- Carry out convolution by hand and in code and analyse the process in terms of impulse responses and the properties of linear time invariant systems
- Use and write software to edit and to apply a range of effects processors to digital audio signals

2.201 Filtering. Implementation of an averaging low-pass filter

Signal Averaging is a technique that tries to remove random noise from a signal through the process of averaging. For the averaging, we generally use an N-point moving average, as shown in the equation below:

$$y(n) = \frac{1}{N} \sum_{k=0}^{N-1} x(n-k)$$

A 3-point moving average filter would be implemented in Python as shown below:

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 values = np.array([3., 9., 3., 4.,
5                    5., 2., 1., 7.,
6                    9., 1., 3., 5.,
7                    4., 9., 0., 4.,
8                    2., 8., 9., 7.])
9
10 N = 3
11
12 averages = np.empty(20)
13
14 for i in range(1, 19):
```

```

15     temp = (values[i-1] + values[i] + values[i+1])/N
16     averages[i] = temp
17
18     # Preserve the edge values
19     averages[0] = values[0]
20     averages[19] = values[19]
21
22     plt.plot(values, 'b-', label='values')
23     plt.plot(averages, 'r==', label='averages')
24     plt.legend(loc="upper left")

```

We can also use `numpy.convolve` or `scipy.ndimage.uniform_filter1d` to implement the moving average:

```

1  window = np.ones(3)
2  window /= sum(window)
3  print(window)
4  plt.plot(window, 'yo-')
5  plt.xlabel('Index')
6
7  averages = np.convolve(values, window, mode='same')
8  plt.plot(values, 'b-', label='values')
9  plt.plot(averages, 'r--', label='averages')
10 plt.legend(loc="upper left")

```

A moving average filter can be used as a simple low-pass filter as it removes high frequency components from the original signal.

```

1  import numpy as np
2  import matplotlib.pyplot as plt
3  from thinkdsp import SquareSignal, Wave
4
5  # Suppress scientific notation for small numbers
6  np.set_printoptions(precision=3, suppress=True)
7
8  signal = SquareSignal(freq=440)
9  wave = signal.make_wave(duration=1.0, framerate=48000)
10 wave.make_audio()
11
12 window = np.ones(15)
13 window /= sum(window)
14 print(window)
15 plt.plot(window, 'yo-')
16 plt.xlabel('Index')
17

```

```

18 segment = wave.segment(duration=0.01)
19 plt.plot(segment.ys, 'b-')
20 plt.xlabel('Samples')
21
22 averages = np.convolve(wave.ys, window, mode='same')
23 smooth = Wave(averages, framerate=wave.framerate)
24 smooth.make_audio()
25
26 plt.plot(wave.ys[:500], 'b-', label='original')
27 plt.plot(smooth.ys[:500], 'r--', label='filtered')
28 plt.xlabel('Samples')
29 plt.legend(loc="upper left")

```

2.203 FIR (Finite Impulse Response) filters

An Averaging Low-Pass Filter is a kind of filter known as Finite Impulse Response Filter. It's *Finite* because its impulse response settles to zero in finite time.

The filter's response is given by the finite series:

$$y(n) = \sum_{k=0}^{N-1} h(k)x(n-k)$$

If we let $h(n) = \frac{1}{N}$, then we have our Averaging Low-Pass Filter as before.

To produce different impulse responses, all we have to do is modify the `window` coefficients from the previous code.

```

1 window = np.array([0.1, 0.2, 0.2, 0.2, 0.1])
2 print(window)
3 plt.plot(window, 'yo-', linewidth=2)
4 decorate(xlabel='Index')
5
6 filtered = np.convolve(wave.ys, window, mode='same')
7 filtered_violin = Wave(filtered, framerate=wave.framerate)
8 filtered_violin.make_audio()

```

2.205 LTI systems

A Linear Time-Invariant System is a system that produces an output signal from any input signal subject to the constraints of linearity and time-invariance¹.

A (digital) signal is just a list of numbers. For audio signal, each number represents the amplitude of the signal sampled over time.

¹https://en.wikipedia.org/wiki/Linear_time-invariant_system

In this context, a System is something we use to process signals, filters for example. If the system happens to be **linear, time-invariant** (LTI) system, we can represent its behavior as a list of numbers known as **Impulse Response**. An Impulse Response is the response produced by an LTI system to the impulse signal.

Considering figure 8, what an LTI system would do is move the circles up and down depending on the system's parameters.

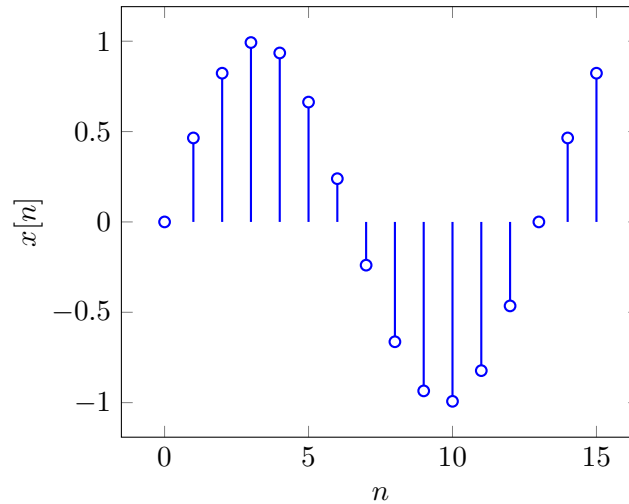


Figure 8: Discrete Sine Wave

LTI Systems use Convolution to process the input signal. Convolution is a mathematical term for a method of combining two functions f and g in a way that g expresses how to modify f . We denote the convolution of f and g and $f * g$.

LTI Systems have three properties:

Homogeneity Linear with respect to scale. Scaling input and applying filter is the same as applying filter and scaling output by the same factor: $0.5f * 0.5g = 0.5(f * g)$

Additivity Can separately process simple signals and add results together. Adding two signals f_1 and f_2 then applying filter is same as applying filter to each signal and adding results: $(f_1 * g) + (f_2 * g) = (f_1 + f_2) * g$

Shift Invariance Later signal results in a later response. Shifting a signal's phase then applying the filter is the same as applying the filter then shifting the output.

Week 5

Key Concepts

- Describe signal analysis with the discrete cosine transform (DCT)
- Review spectral analysis and describe spectral analysis with the discrete Fourier transform (DFT)
- Use the discrete Fourier transform (DFT) to complex analysis and fast convolution

3.001 Review of spectral analysis

When looking at the time domain, we don't get a feeling for how the audio is going to sound because we're oblivious to the component frequencies in the sound.

A Frequency Analyzer will convert the signal to the frequency domain and let us *see* all the component frequencies in the audio sample.

3.003 Spectral analysis of periodic signals

"Fourier Decomposition is a formalisation of the process of figuring out how to make a given signal out of a set of sine waves."

$$S_N(x) = \frac{a_0}{2} + \sum_{n=1}^N \left(a_n \cos\left(\frac{2\pi}{P}nx\right) + b_n \sin\left(\frac{2\pi}{P}nx\right) \right)$$

The Discrete Cosine Transform is given by

$$X_k = \sum_{n=0}^{N-1} x_n \cos \left[\frac{\pi}{N} \left(n + \frac{1}{2} \right) \left(k + \frac{1}{2} \right) \right]$$

where $k = 0, 1, \dots, N - 1$.

Week 6

Key Concepts

- Describe signal analysis with the discrete cosine transform (DCT)
- Review spectral analysis and describe spectral analysis with the discrete Fourier transform (DFT)
- Use the discrete Fourier transform (DFT) to complex analysis and fast convolution

3.101 Complex synthesis

The Discrete Cosine Transform ignores phase. In order to consider phases we will need Complex Numbers. Complex numbers are of the form

$$a + ib$$

where a is a *real* number and b is the *imaginary* part of the number.

In Python the character i is replaced with a j .

```
1 c = 1j # Creates a new complex number
2 print(c.real, c.imag)
```

To compute the waveforms from complex numbers we rely in the exponential function e^x (`np.exp(x)` in Python).

Week 7

Key Concepts

- Describe audio signal feature extraction and the characteristics of time domain features and frequency domain features
- Use p5.sound and the JavaScript library Meyda to write software to extract audio signal features
- Use Python and the library Librosa to write software to extract audio signal features

4.001 Introduction to audio feature extraction

An audio feature is a characteristic of an audio signal that gives insight into what the signal contains.

Some examples of audio features are listed below:

- Energy
- Spectrum
- Fundamental Frequency
- Loudness
- Brightness
- Pitch
- Rhythm
- ...

These features can be programmatically extracted from audio signals by using certain algorithms. There are algorithms for real-time and non-real-time feature extraction.

Librosa is an audio and music processing library for Python which implements feature extraction. Meyda is a similar library for JavaScript.

Audio features have several applications, including:

- Real-time audio visualizations
- Visual score generation

- Speech recognition
- Music recommendation
- Music genre classification
- Feature-based synthesis
- Feature extraction linked to audio effects
- ...

From the time-domain representation, we can extract the following features:

- Energy
- RMS energy
- Zero crossing rate
- Amplitude envelope
- ...

From the frequency-domain representation, we can extract the following features:

- Spectral centroid
- Spectral rolloff
- Spectral flatness
- Spectral flux
- Spectral slope
- Spectral spread
- Mel-frequency cepstral coefficients
- Chroma
- ...

The Root-mean-square energy (RMSE) is formalised as shown below

$$RMSE = \sqrt{\frac{1}{N} \sum_n |x(n)|^2}$$

4.003 Audio feature extraction

- Collins, N. Introduction to computer music. (Chichester: John Wiley & Sons, 2010). Chapter 3: Feature extraction (pp.99–111)

4.101 Real-time audio visualisations with p5.js. – time domain features

To extract the amplitude from audio file using p5.js, we can use the Amplitude object.

4.102 Real-time audio visualisations with p5.js – frequency domain features (1)

To extract features from the frequency domain, we need an FFT object.

4.105 Real-time audio visualisations with Meyda

Meyda is an audio feature extraction library for JavaScript. It's written to use the Web Audio API. It supports both real-time and offline feature extraction.

4.107 Meyda: an audio feature extraction library for the Web Audio API

- Fiala, J., N. Segal and H.A. Rawlinson 'Meyda: an audio feature extraction library for the Web Audio API' in Web Audio Conference (WAC 2015), 2015, pp.1-6.