**WHAT IS SOUND?**

Sound which is transmitted through the air as air pressure oscillations. In essence, sound is simply air vibrating. Sound vibrates through the air as longitudinal waves, i.e. the oscillations are parallel to the direction of propagation.

**WHAT IS AUDIO?**

Audio refers to the production, transmission, or reception of sounds that are audible by humans. An audio signal is a representation of sound that represents the fluctuation in air pressure caused by the vibration as a function of time. Unlike sheet music or symbolic representations, audio representations encode everything that is necessary to reproduce an acoustic realization of a piece of music. However, note parameters such as onsets, durations, and pitches are not encoded explicitly. This makes converting from an audio representation to a symbolic representation a difficult and ill-defined task.

1. The basic representation of an audio signal is in the time domain.
2. **Waveform:** The change in air pressure at a certain time is graphically represented by a pressure-time plot
3. **Spectrogram:** Shows the intensity of frequency components over time. Used to visualize the partials of a sound.

Fourier Transform:

|  |
| --- |
| X = librosa.stft(x) # x:numpy array,.stft() converts data into short term Fourier transform to know amplitude of given frequency at given time Xdb = librosa.amplitude\_to\_db(abs(X)) # converting into energy levels(dB)  plt.figure(figsize=(20, 5)) # spectrum of frequencies of a signal as it varies with time to see how energy levels (dB) vary over time. librosa.display.specshow(Xdb, sr=sr, x\_axis='time', y\_axis='hz') # .specshow is used to display a spectrogram. plt.colorbar() # time-frequency portraits of signals |

1. **Sampling rate:** Digital computers can only capture this data at discrete moments in time. The rate at which a computer captures audio data is called the sampling frequency. sr= 44100 Hz, the sampling rate of CD recordings.
2. Amplitude and Frequency: <https://groups.google.com/forum/#!topic/librosa/TIke2HVJHdc>
3. Pitch
4. **Onset:**
5. **Timbre:** The quality of sound that distinguishes the tone of different instruments and voices even if the sounds have the same pitch and loudness.One characteristic of timbre is its temporal evolution modeled by the ADSR model and another is the existence of partials and their relative strengths which is visualized with a spectrogram.
6. Temporal vs spectral indicators
7. Pure tone: is a sound with a [sinusoidal](https://en.wikipedia.org/wiki/Sine_wave#Occurrences) [waveform](https://en.wikipedia.org/wiki/Waveform); that is, a [sine](https://en.wikipedia.org/wiki/Sine) wave of any frequency, phase, and amplitude. A pure tone has the property – unique among real-valued wave shapes – that its wave shape is unchanged by [linear time-invariant systems](https://en.wikipedia.org/wiki/Linear_time-invariant_system); that is, only the phase and amplitude change between such a system's pure-tone input and its output.
8. Oboe, Clarinet
9. Sonification: Now think of visualizing for the ears. use of non-speech [audio](https://en.wikipedia.org/wiki/Sound) to convey [information](https://en.wikipedia.org/wiki/Information) : you take data of some kind and create sound with it. The information is translated into pitch, volume, stereo position, brightness, etc.
10. There are two types of features of a speech signal:

* **The temporal features** (time domain features), which are simple to extract and have easy physical interpretation, like: the energy of signal, zero crossing rate, maximum amplitude, minimum energy, etc.
* **The spectral features** (frequency based features), which are obtained by converting the time based signal into the frequency domain using the Fourier Transform, like: fundamental frequency, frequency components, spectral centroid, spectral flux, spectral density, spectral roll-off, etc. These features can be used to identify the notes, pitch, rhythm, and melody.

## **Timbre: Temporal Indicators**

**Timbre** is the quality of sound that distinguishes the tone of different instruments and voices even if the sounds have the **same pitch and loudness.**

One characteristic of timbre is its temporal evolution. The **envelope** of a signal is a smooth curve that approximates the amplitude extremes of a waveform over time.

Envelopes are often modeled by the **ADSR model** ([Wikipedia](https://en.wikipedia.org/wiki/Synthesizer#Attack_Decay_Sustain_Release_.28ADSR.29_envelope)) which describes four phases of a sound: attack, decay, sustain, release.

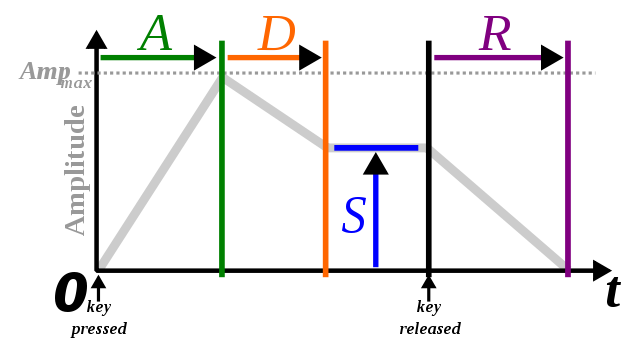
During the attack phase, the sound builds up, usually with noise-like components over a broad frequency range. **Such a noise-like short-duration sound at the start of a sound is often called a transient.**

During the decay phase, the sound stabilizes and reaches a steady periodic pattern.

During the sustain phase, the energy remains fairly constant.

During the release phase, the sound fades away.

The ADSR model is a simplification and does not necessarily model the amplitude envelopes of all sounds.



## **Timbre: Spectral Indicators**

Another property used to characterize timbre is the existence of partials and their relative strengths. **Partials** are the dominant frequencies in a musical tone with the lowest partial being the **fundamental frequency**.

The partials of a sound are visualized with a **spectrogram**. A spectrogram shows the intensity of frequency components over time. (See [Fourier Transform](https://musicinformationretrieval.com/fourier_transform.html) and [Short-Time Fourier Transform](https://musicinformationretrieval.com/stft.html) for more.)

### **Pure Tone**

Let's synthesize a pure tone at 1047 Hz, concert C6:

In [5]:

|  |
| --- |
| T = 2.0 # seconds f0 = 1047.0 sr = 22050 t = numpy.linspace(0, T, int(T\*sr), endpoint=False) # time variable x = 0.1\*numpy.sin(2\*numpy.pi\*f0\*t) ipd.Audio(x, rate=sr) |

Display the spectrum of the pure tone:

In [6]:

|  |
| --- |
| X = scipy.fft(x[:4096]) X\_mag = numpy.absolute(X) # spectral magnitude f = numpy.linspace(0, sr, 4096) # frequency variable plt.figure(figsize=(14, 5)) plt.plot(f[:2000], X\_mag[:2000]) # magnitude spectrum plt.xlabel('Frequency (Hz)') |

### **Oboe**

Let's listen to an oboe playing a C6:

|  |
| --- |
| In [7]: x, sr = librosa.load('audio/oboe\_c6.wav') ipd.Audio(x, rate=sr) print(x.shape) |

Display the spectrum of the oboe:

In [9]:

|  |
| --- |
| X = scipy.fft(x[10000:14096]) X\_mag = numpy.absolute(X) plt.figure(figsize=(14, 5)) plt.plot(f[:2000], X\_mag[:2000]) # magnitude spectrum plt.xlabel('Frequency (Hz)') |

### **Clarinet**

Let's listen to a clarinet playing a concert C6:

In [10]:

|  |
| --- |
| x, sr = librosa.load('audio/clarinet\_c6.wav') ipd.Audio(x, rate=sr) print(x.shape) |

(51386,)

|  |
| --- |
| X = scipy.fft(x[10000:14096]) X\_mag = numpy.absolute(X) plt.figure(figsize=(14, 5)) plt.plot(f[:2000], X\_mag[:2000]) # magnitude spectrum plt.xlabel('Frequency (Hz)') |

Notice the difference in the relative amplitudes of the partial components. All three signals have approximately the same pitch and fundamental frequency, yet their timbres differ

# **Understanding Audio Features through Sonification**

We will segment, feature extract, and analyze audio files. Goals:

1. Detect onsets in an audio signal.
2. Segment the audio signal at each onset.
3. Compute features for each segment.
4. Gain intuition into the features by listening to each segment separately.

**Step-1 Data retrieval and preparation**

Compute the short-time Fourier transform:

|  |
| --- |
| X = librosa.stft(x) |

For display purposes, compute the log amplitude of the STFT:

|  |
| --- |
| Xmag = librosa.logamplitude(X) |

Display the spectrogram.

|  |
| --- |
| # Play with the parameters, including x\_axis and y\_axis librosa.display.specshow(Xmag, sr=sr, x\_axis='time', y\_axis='log') |

## **Step 2: Detect Onsets**

Find the times, in seconds, when onsets occur in the audio signal.

|  |
| --- |
| onset\_frames = librosa.onset.onset\_detect(x, sr=sr) print onset\_frame onset\_times = librosa.frames\_to\_time(onset\_frames, sr=sr) print onset\_times |

Convert the onset frames into sample indices.

In [11]:

|  |
| --- |
| onset\_samples = librosa.frames\_to\_samples(onset\_frames) print onset\_samples |

Play a "beep" at each onset.

|  |
| --- |
| # Use the `length` parameter so the click track is the same length as the original signal clicks = librosa.clicks(times=onset\_times, length=len(x)) |

|  |
| --- |
| # Play the click track "added to" the original signal ipd.Audio(x+clicks, rate=sr) |

## **Step 4: Extract Features**

For each segment, compute the zero crossing rate.

|  |
| --- |
| zcrs = [sum(librosa.core.zero\_crossings(segment)) for segment in segments] print zcrs |

Use **argsort** to find an index array, **ind**, such that **segments[ind]** is sorted by zero crossing rate.

|  |
| --- |
| ind = numpy.argsort(zcrs) print ind |

Sort the segments by zero crossing rate, and concatenate the sorted segments.

|  |
| --- |
| concatenated\_signal = concatenate\_segments(segments[ind], sr) |

## **Step 5: Listen to Segments**

Listen to the sorted segments. What do you hear?

|  |
| --- |
| ipd.Audio(concatenated\_signal, rate=sr) |

# **Basic Feature Extraction**

Somehow, we must extract the characteristics of our audio signal that are most relevant to the problem we are trying to solve. For example, if we want to classify instruments by timbre, we will want features that distinguish sounds by their timbre and not their pitch. If we want to perform pitch detection, we want features that distinguish pitch and not timbre.

This process is known as feature extraction.

## **Constructing a Feature Vector**

A *feature vector* is simply a collection of features. Here is a simple function that constructs a two-dimensional feature vector from a signal:

|  |
| --- |
| def extract\_features(signal):  return [  librosa.feature.zero\_crossing\_rate(signal)[0, 0],  librosa.feature.spectral\_centroid(signal)[0, 0],  ] |

If we want to aggregate all of the feature vectors among signals in a collection, we can use a list comprehension as follows:

|  |
| --- |
| kick\_features = numpy.array([extract\_features(x) for x in kick\_signals]) snare\_features = numpy.array([extract\_features(x) for x in snare\_signals]) |

Visualize the differences in features by plotting separate histograms for each of the classes:

|  |
| --- |
| plt.figure(figsize=(14, 5)) plt.hist(kick\_features[:,0], color='b', range=(0, 0.2), alpha=0.5, bins=20) plt.hist(snare\_features[:,0], color='r', range=(0, 0.2), alpha=0.5, bins=20) plt.legend(('kicks', 'snares')) plt.xlabel('Zero Crossing Rate') plt.ylabel('Count') |

|  |
| --- |
| plt.figure(figsize=(14, 5)) plt.hist(kick\_features[:,1], color='b', range=(0, 4000), bins=30, alpha=0.6) plt.hist(snare\_features[:,1], color='r', range=(0, 4000), bins=30, alpha=0.6) plt.legend(('kicks', 'snares')) plt.xlabel('Spectral Centroid (frequency bin)') plt.ylabel('Count') |

## **Feature Scaling**[**¶**](https://musicinformationretrieval.com/basic_feature_extraction.html#Feature-Scaling)

The features that we used in the previous example included zero crossing rate and spectral centroid. These two features are expressed using different units. This discrepancy can pose problems when performing classification later. Therefore, we will normalize each feature vector to a common range and store the normalization parameters for later use.

Many techniques exist for scaling your features. For now, we'll use [**sklearn.preprocessing.MinMaxScaler**](http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html). **MinMaxScaler** returns an array of scaled values such that each feature dimension is in the range -1 to 1.

Let's concatenate all of our feature vectors into one *feature table*:

|  |
| --- |
| feature\_table = numpy.vstack((kick\_features, snare\_features)) print(feature\_table.shape) |

(20, 2)

Scale each feature dimension to be in the range -1 to 1:

|  |
| --- |
| scaler = sklearn.preprocessing.MinMaxScaler(feature\_range=(-1, 1)) training\_features = scaler.fit\_transform(feature\_table) print(training\_features.min(axis=0)) print(training\_features.max(axis=0)) |

[-1. -1.]

[1. 1.]

Plot the scaled features:

|  |
| --- |
| plt.scatter(training\_features[:10,0], training\_features[:10,1], c='b') plt.scatter(training\_features[10:,0], training\_features[10:,1], c='r') plt.xlabel('Zero Crossing Rate') plt.ylabel('Spectral Centroid') |

# **Segmentation**

# In audio processing, it is common to operate on one frame at a time using a constant frame size and hop size (i.e. increment). Frames are typically chosen to be 10 to 100 ms in duration.

# Let's create an audio signal consisting of a pure tone that gradually gets louder. Then, we will segment the signal and compute the root mean square (RMS) energy for each frame.

# First, set our parameters:

# 

|  |
| --- |
| ***T = 3.0 # duration in seconds sr = 22050 # sampling rate in Hertz amplitude = numpy.logspace(-3, 0, int(T\*sr), endpoint=False, base=10.0) # time-varying amplitude print amplitude.min(), amplitude.max() # starts at 110 Hz, ends at 880 Hz*** |

# Create the signal

|  |
| --- |
| **t = numpy.linspace(0, T, int(T\*sr), endpoint=False) x = amplitude\*numpy.sin(2\*numpy.pi\*440\*t)** |

# 

# Listen to the signal

|  |
| --- |
| **ipd.Audio(x, rate=sr)** |

# 

# Plot the signal:

|  |
| --- |
| **librosa.display.waveplot(x, sr=sr)** |

## **Segmentation Using Python List Comprehensions**

# In Python, you can use a standard [list comprehension](https://docs.python.org/2/tutorial/datastructures.html#list-comprehensions) to perform segmentation of a signal and compute RMSE at the same time.

# Initialize segmentation parameters:

|  |
| --- |
| **frame\_length = 1024 hop\_length = 512** |

# 

# Define a helper function:

|  |
| --- |
| **def rmse(x):  return numpy.sqrt(numpy.mean(x\*\*2))** |

# 

# Using a list comprehension, plot the RMSE for each frame on a log-y axis:

|  |
| --- |
| **plt.semilogy([rmse(x[i:i+frame\_length])  for i in range(0, len(x), hop\_length)])** |

## **librosa.util.frame**

# Given a signal, [librosa.util.frame](https://librosa.github.io/librosa/generated/librosa.util.frame.html#librosa.util.frame) will produce a list of uniformly sized frames:

|  |
| --- |
| frames = librosa.util.frame(x, frame\_length=frame\_length, hop\_length=hop\_length) plt.semilogy([rmse(frame) for frame in frames.T]) |

# That being said, in librosa, manual segmentation of a signal is often unnecessary, because the feature extraction methods themselves do segmentation for you

# **Energy and RMSE**

The **energy** ([Wikipedia](https://en.wikipedia.org/wiki/Energy_(signal_processing%29); FMP, p. 66) of a signal corresponds to the total magnitude of the signal. For audio signals, that roughly corresponds to how loud the signal is. The energy in a signal is defined as

$$ \sum\_n \left| x(n) \right|^2 $$

The **root-mean-square energy (RMSE)** in a signal is defined as

$$ \sqrt{ \frac{1}{N} \sum\_n \left| x(n) \right|^2 } $$

Compute the short-time energy using a list comprehension:

|  |
| --- |
| hop\_length = 256 frame\_length = 512 |

|  |
| --- |
| energy = numpy.array([  sum(abs(x[i:i+frame\_length]\*\*2))  for i in range(0, len(x), hop\_length) ]) |

|  |
| --- |
| energy.shape |

(194,)

Compute the RMSE using [**librosa.feature.rmse**](https://librosa.github.io/librosa/generated/librosa.feature.rmse.html):

|  |
| --- |
| rmse = librosa.feature.rmse(x, frame\_length=frame\_length, hop\_length=hop\_length, center=True)  rmse.shape # (1, 194)  rmse = rmse[0] |

Plot both the energy and RMSE along with the waveform:

|  |
| --- |
| frames = range(len(energy)) t = librosa.frames\_to\_time(frames, sr=sr, hop\_length=hop\_length) |

# Write a function, strip, that removes leading silence from a signal. Make sure it works for a variety of signals recorded in different environments and with different signal-to-noise ratios (SNR).

|  |
| --- |
| **def strip(x, frame\_length, hop\_length):   # Compute RMSE.  rmse = librosa.feature.rmse(x, frame\_length=frame\_length, hop\_length=hop\_length, center=True)    # Identify the first frame index where RMSE exceeds a threshold.  thresh = 0.01  frame\_index = 0  while rmse[0][frame\_index] < thresh:  frame\_index += 1    # Convert units of frames to samples.  start\_sample\_index = librosa.frames\_to\_samples(frame\_index, hop\_length=hop\_length)    # Return the trimmed signal.  return x[start\_sample\_index:]** |

# 

# Let's see if it works.

|  |
| --- |
| **In [18]: y = strip(x, frame\_length, hop\_length)** |

# **Zero Crossing Rate**

# The [zero crossing rate](https://en.wikipedia.org/wiki/Zero-crossing_rate) indicates the number of times that a signal crosses the horizontal axis.

|  |
| --- |
| **librosa.display.waveplot(x, sr=sr)** |

# Let's zoom in:

|  |
| --- |
| n0 = 6500 n1 = 7500 plt.figure(figsize=(14, 5)) plt.plot(x[n0:n1]) |

# 

# I count five zero crossings. Let's compute the zero crossings using librosa.

# 

|  |
| --- |
| **zero\_crossings = librosa.zero\_crossings(x[n0:n1], pad=False)  zero\_crossings.shape** |

# (1000,)

# That computed a binary mask where True indicates the presence of a zero crossing. To find the total number of zero crossings, use sum:

|  |
| --- |
| **print(sum(zero\_crossings))** |

# To find the *zero-crossing rate* over time, use zero\_crossing\_rate:

# 

|  |
| --- |
| **zcrs = librosa.feature.zero\_crossing\_rate(x) print(zcrs.shape)** |

# 

# Plot the zero-crossing rate:

# 

|  |
| --- |
| **plt.figure(figsize=(14, 5)) plt.plot(zcrs[0])** |

# Note how the high zero-crossing rate corresponds to the presence of the snare drum.

# The reason for the high rate near the beginning is because the silence oscillates quietly around zero:

# In [11]:

|  |
| --- |
| plt.figure(figsize=(14, 5)) plt.plot(x[:1000]) plt.ylim(-0.0001, 0.0001) |

# A simple hack around this is to add a small constant before computing the zero crossing rate:

|  |
| --- |
| In [12]: zcrs = librosa.feature.zero\_crossing\_rate(x + 0.0001) plt.figure(figsize=(14, 5)) plt.plot(zcrs[0]) |

[[Does the zero-crossing rate still return something useful in polyphonic mixtures?]]

## **Fourier Transform**[**Â¶**](https://musicinformationretrieval.com/fourier_transform.html#Fourier-Transform)

The *Fourier Transform* ([Wikipedia](https://en.wikipedia.org/wiki/Fourier_transform)) is one of the most fundamental operations in applied mathematics and signal processing.

It transforms our time-domain signal into the *frequency domain*. Whereas the time domain expresses our signal as a sequence of samples, the frequency domain expresses our signal as a *superposition of sinusoids* of varying magnitudes, frequencies, and phase offsets.

To compute a Fourier transform in NumPy or SciPy, use [**scipy.fft**](http://docs.scipy.org/doc/scipy/reference/generated/scipy.fftpack.fft.html#scipy.fftpack.fft):

In [5]:

|  |
| --- |
| X = scipy.fft(x) X\_mag = numpy.absolute(X) f = numpy.linspace(0, sr, len(X\_mag)) # frequency variable |

Plot the spectrum:

In [6]:

|  |
| --- |
| plt.figure(figsize=(13, 5)) plt.plot(f, X\_mag) # magnitude spectrum plt.xlabel('Frequency (Hz)') |

Zoom in:

In [7]:

|  |
| --- |
| plt.figure(figsize=(13, 5)) plt.plot(f[:5000], X\_mag[:5000]) plt.xlabel('Frequency (Hz)') |

# **Short-Time Fourier Transform**

Musical signals are highly non-stationary, i.e., their statistics change over time. It would be rather meaningless to compute a single Fourier transform over an entire 10-minute song.

The **short-time Fourier transform (STFT)** ([Wikipedia](https://en.wikipedia.org/wiki/Short-time_Fourier_transform); FMP, p. 53) is obtained by computing the Fourier transform for successive frames in a signal.

X(m,ω)=∑nx(n)w(n−m)e−jωn

X(m,ω)=∑nx(n)w(n−m)e−jωn

As we increase mm, we slide the window function ww to the right. For the resulting frame, x(n)w(n−m)x(n)w(n−m), we compute the Fourier transform. Therefore, the STFT XX is a function of both time, mm, and frequency, ωω.

# **What is Music Information Retrieval?**

Naming as many of its musical characteristics as you can the

* genre?
* tempo?
* Instruments?
* mood?
* time signature?
* key signature?
* chord progression?
* tuning frequency?
* song structure?

Here is a sampling of tasks found in music information retrieval:

* fingerprinting
* cover song detection
* genre recognition
* transcription
* recommendation
* symbolic melodic similarity
* mood
* source separation
* instrument recognition
* pitch tracking
* tempo estimation
* score alignment
* song structure/form
* beat tracking
* key detection
* query by humming

## **Why MIR?**[**¶**](https://musicinformationretrieval.com/why_mir.html#Why-MIR?)

* discover, organize, monetize media collections
* search ("find me something that sounds like this") songs, loops, speech, environmental sounds, sound effects
* workflows in consumer products through machine hearing
* automatic control of software and mobile devices

### **Commercial Applications**

Example: [RiffStation](http://www.riffstation.com/)

Example: [Melodyne](http://www.celemony.com/en/start)

Example: [Auto-Tune](http://www.antarestech.com/)

Example: Key Detection and Auto-harmonization with [iZotope Nectar 2](https://www.izotope.com/nectar2)