FFT and Its Applications

EE4015 Digital Signal Processing

Dr. Lai-Man Po
Department of Electrical Engineering
City University of Hong Kong

Content

- **Fast Fourier Transform (FFT) algorithms**
  - Decimation-in-Time
  - Decimation-in-Frequency
- **Applications of FFT**
  - Fast Convolution
  - Spectral Analysis
  - Signal Denoising
    - Audio Signal Denoising
    - Notch Filtering for Image Processing (Optional)
  - Spectrogram – Time-Frequency Analysis (Optional)

Fast Fourier Transform (FFT)

Fast Fourier Transform (FFT)

- The Fast Fourier Transform (FFT) is simply a mathematical technique to accelerate the calculation of the DFT. It was invented by **Gauss** in 1805 and re-invented by **Cooley and Tukey** in 1965.
  - Typically, if the DFT is directly calculated for a block of 2^N samples e.g. 512 or 1024 samples (N) it would make the calculation of the DFT quite demanding.
  - The FFT simply uses repetition and redundancy in the calculation to speed it up.
- **The FFT is simply a FAST ALOGORITHM to calculate the DFT, NOT a different transform.**

Fast Fourier Transform (FFT)

- FFT is a fast algorithm for DFT and inverse DFT computations.

$$X[k] = \sum_{n=0}^{N-1} x[k] W_N^{kn} \qquad x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X[k] W_N^{-kn}$$

- Complexity of direct computation of DFT:
  - Each DFT coefficient requires
    - N complex multiplications
    - N-1 complex additions
  - All N DFT coefficients require
    - N^2 complex multiplications
    - N(N-1) complex additions

- Direct DFT Computation : $O(N^2)$
- FFT Computation : $O(N \log_2 N)$

Complexities of Direct DFT computation and FFT

N	Direct Computation		FFT	
	Multiplication N^2	Addition $N(N-1)$	Multiplication $0.5N\log_2(N)$	Addition $N\log_2(N)$
2	4	2	1	2
8	64	56	12	24
32	1024	922	80	160
64	4096	4022	192	384
2^{10}	1048576	1047552	5120	10240
2^{20}	~10^{12}	~10^{12}	~10^7	~2×10^7

FFT Exploits symmetries of $e^{-j\frac{2\pi}{N}kn}$

- $W_N = e^{-j\frac{2\pi}{N}}$ (Twiddle Factor)
- FFT fast algorithms are based on symmetry properties
  - Conjugate symmetry

$$W_N^{k(N-n)} = e^{-j\frac{2\pi}{N}k(N-n)} = e^{-j\frac{2\pi}{N}kN} e^{-j\frac{2\pi}{N}k(-n)} = \left(W_N^{kn}\right)^*$$

  - Periodicity in n and k

$$W_N^{kn} = W_N^{k(N+n)} = W_N^{(k+N)n}$$

Decimation-in-Time FFT Algorithm

- The basic idea is to compute the DFT according to

$$X[k] = \sum_{n=\text{even}}^{N-1} x[n]W_N^{kn} + \sum_{n=\text{odd}}^{N-1} x[n]W_N^{kn}$$

- Substituting n=2r and n=2r+1 for the first and second summation terms:

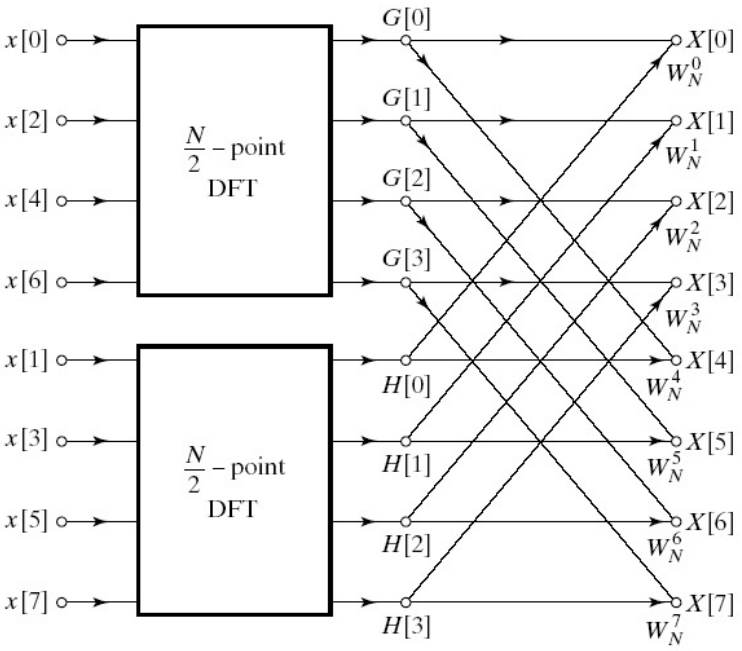
$$X[k] = \sum_{r=0}^{N/2-1} x[2r]W_N^{2rk} + \sum_{r=0}^{N/2-1} x[2r+1]W_N^{(2r+1)k}$$

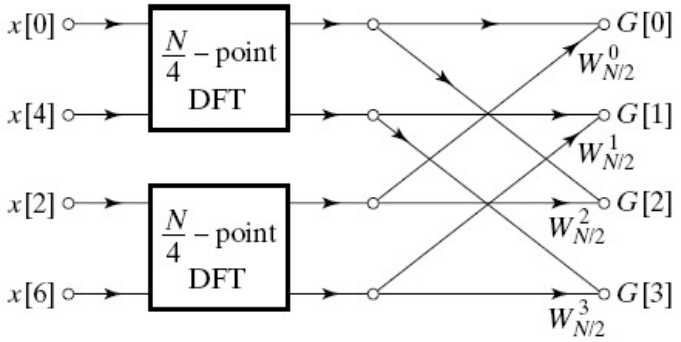
$$= \underbrace{\sum_{r=0}^{N/2-1} x[2r]\left(W_N^2\right)^{rk}}_{\text{N/2-point DFT for even samples}} + W_N^k \underbrace{\sum_{r=0}^{N/2-1} x[2r+1]\left(W_N^2\right)^{rk}}_{\text{N/2-point DFT for odd samples}}$$

$$W_N^2 = e^{-\frac{j2\pi}{N}2} = e^{-\frac{j2\pi}{N/2}} = W_{N/2}^1$$

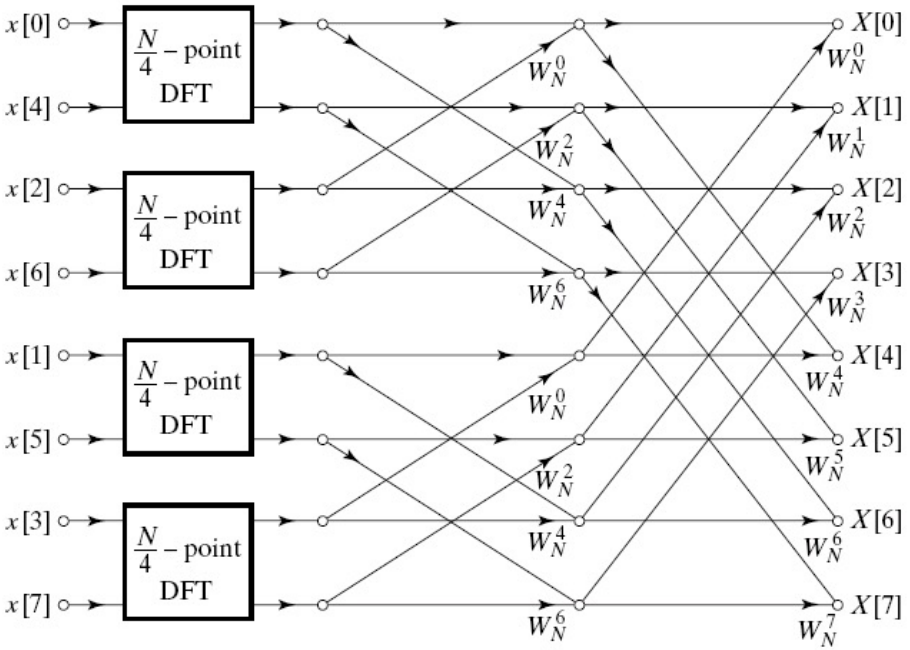
8-point DFT example using decimation-in-time

- Two N/2-point DFTs
  - $2(N/2)^2$ complex multiplications
  - $2(N/2)^2$ complex additions
- Combining the DFT outputs
  - N complex multiplications
  - N complex additions
- **Total complexity**
  - $N^2/2+N$ complex multiplications
  - $N^2/2+N$ complex additions
  - More efficient than direct DFT
- Repeat same process
  - Divide N/2-point DFTs into
    - Two N/4-point DFTs
    - Combine outputs

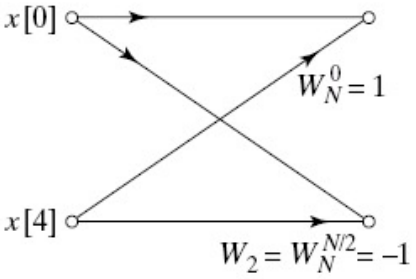




- After two steps of decimation in time

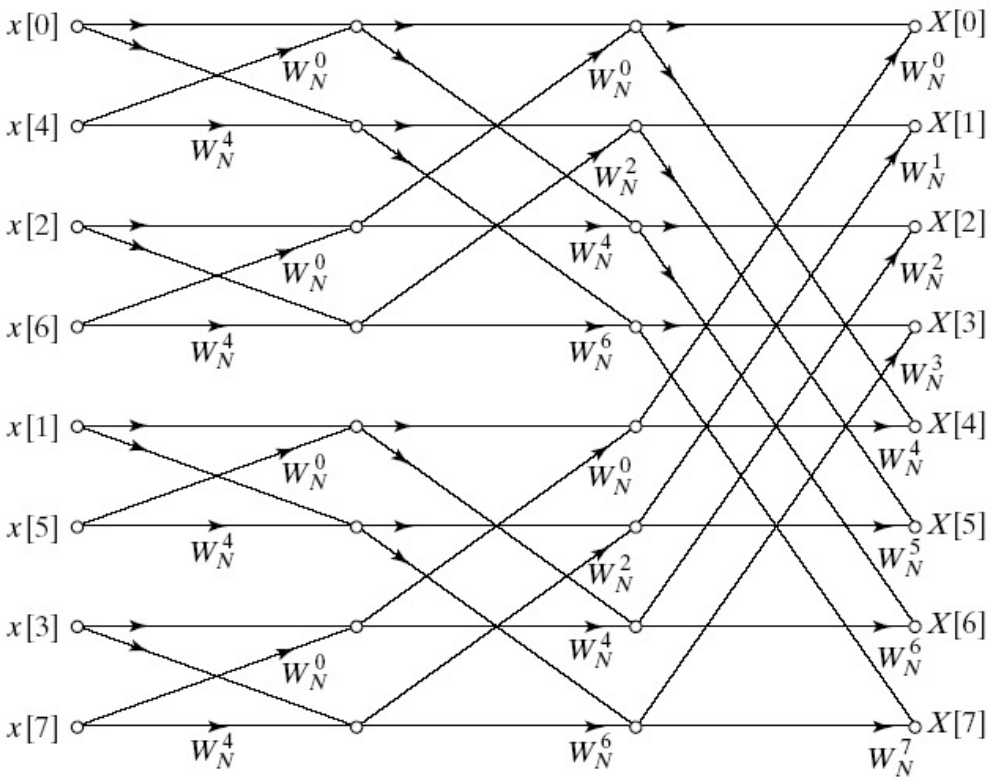


- Repeat until we are left with two-point DFT's



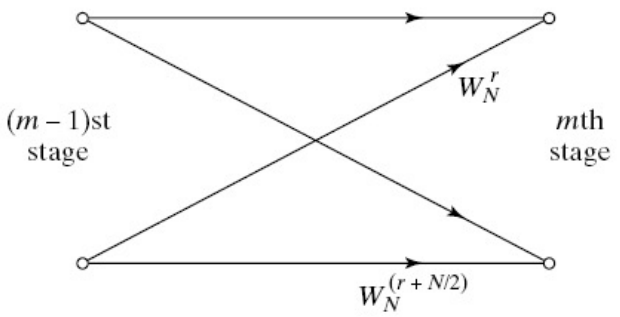
Decimation-in-Time FFT Algorithm

- Final flow graph for 8-point decimation in time
- Complexity:
  - $Nlog_2(N)$ complex multiplications and additions

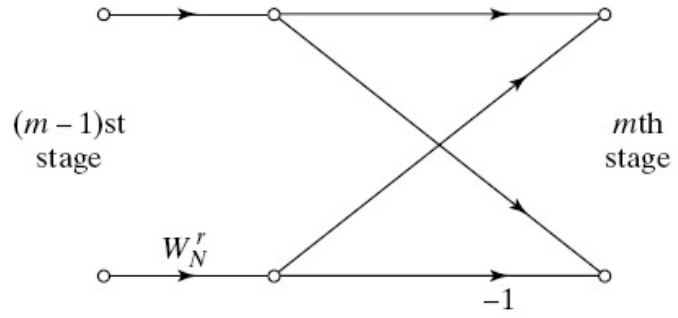


Butterfly Computation

- Flow graph constitutes of butterflies



- We can implement each butterfly with one multiplication



- Final complexity for decimation-in-time FFT
  - $(N/2)log_2(N)$ complex multiplications and additions

Decimation-in-Frequency FFT Algorithm

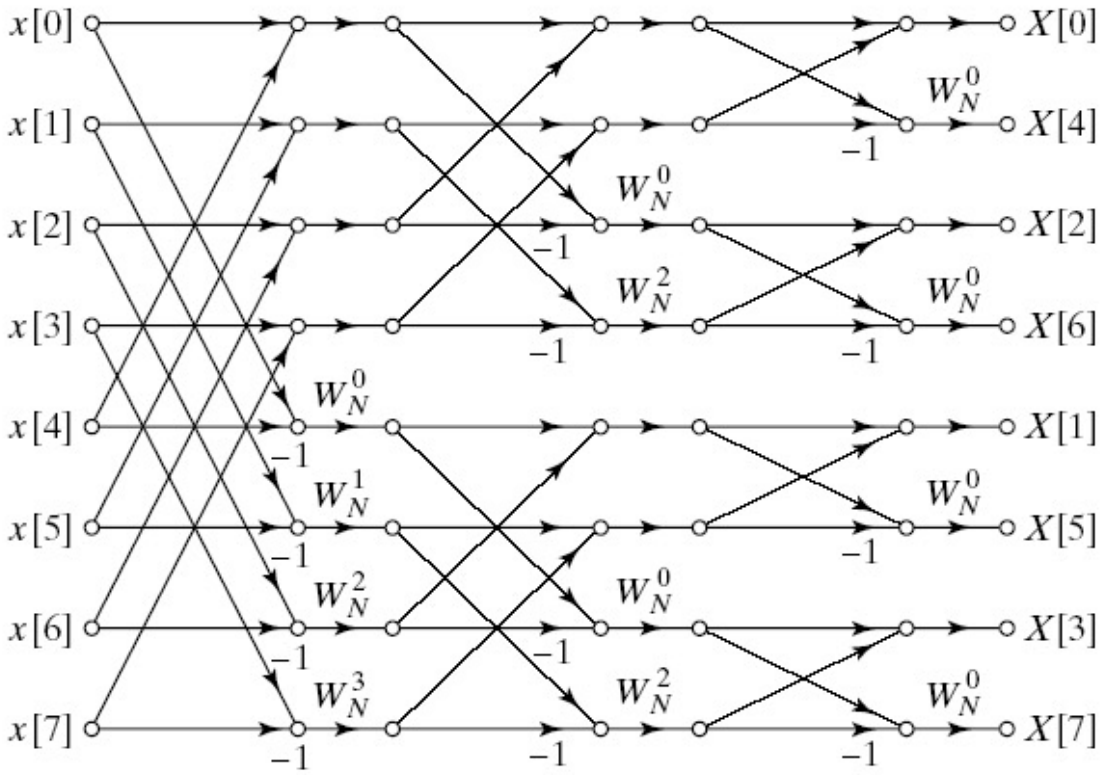
- The basic idea is to compute the frequency-domain sequence $X[k]$ into successively smaller subsequences

$$X[2r] = \sum_{n=0}^{N-1} x[n]W_N^{n(2r)} = \sum_{n=0}^{N/2-1} x[n]W_N^{2nr} + \sum_{n=N/2}^{N-1} x[n]W_N^{2nr}$$

$$= \sum_{n=0}^{N/2-1} x[n]W_N^{2nr} + \sum_{n=0}^{N/2-1} x[n+N/2]W_N^{2r(n+N/2)}$$

$$= \sum_{n=0}^{N/2-1} (x[n] + x[n+N/2]) \cdot W_{N/2}^{nr}, \quad r = 0, 1, \cdots, N/2 - 1$$

8-Point Decimation-In-Frequency FFT



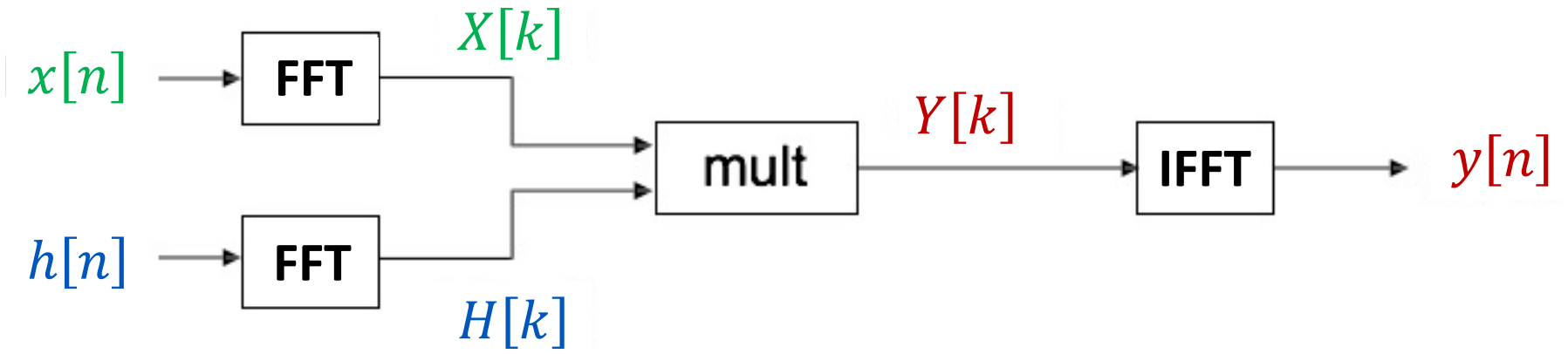
Applications of FFT

- **FFT is a very powerful tools for computer-based frequency domain analysis**
  - Fast Convolution
  - Spectral Analysis (e.g. finding periodicities)
  - Signal Denoising
    - Audio Signal Denoising
    - Notch Filtering for Image Processing (Optional)
  - Spectrogram – Time-Frequency Analysis (Optional)

Fast Convolution with FFT

- The convolution of two finite-duration sequences
  - $y[n] = x_1[n] \otimes x_2[n]$
- where $x_1[n]$ is of length N_1 and $x_2[n]$ is of length N_2 requires computation of $(N_1 + N_2 - 1)$ samples which corresponds to $N_1 N_2 - \min\{(N_1, N_2\}$ complex multiplications
- An alternate approach is to use FFT:
  - $y[n] = IFFT\{FFT\{x_1[n]\} \cdot FFT\{x_2[n]\}\}$
- In practice:
  - Choose the minimum $N \geq N_1 + N_2 - 1$ and is power of 2
  - Zero-pad $x_1[n]$ and $x_2[n]$ to length N, say, $x_1[n]$ and $x_2[n]$
  - $y[n] = IFFT\{FFT\{x_1[n]\} \cdot FFT\{x_2[n]\}\}$

Block Diagram of Fast Convolution with FFT



- If the impulse response is NOT the same size as the input sequence $x[n]$, we have to pad the $h[n]$ with zeros to match the length
- Multiplication is point-by-point, of complex numbers

Complexity of Fast Convolution with FFT

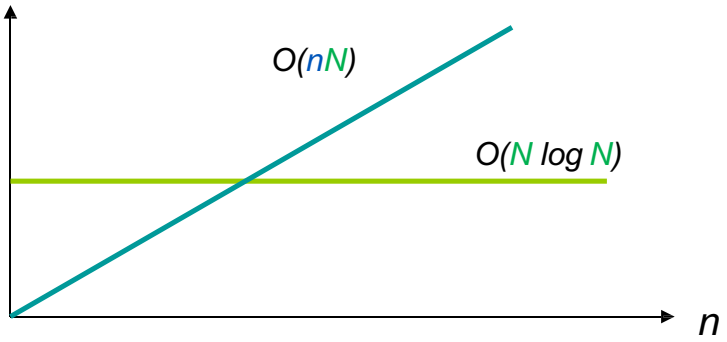
- The inverse DFT has a factor of $1/N$, the IFFT thus requires $N + (N/2)log_2(N)$ multiplications.
- As a result, the total multiplications for $y[n]$ is $2N + (3N/2)log_2(N)$
- Using FFT is more computationally efficient than direct convolution computation for longer data lengths:

N_1	N_2	N	$N_1N_2 - \min\{N_1, N_2\}$	$2N + (3N/2)\log_2(N)$
2	5	8	8	52
10	15	32	140	304
50	80	256	3950	3584
50	1000	2048	49950	37888
512	10000	16384	4119488	376832

Why Fast Convolution with FFT?

- Cost (number of operations) of convolution in time domain ($x[n] * h[n]$)
  - **O(n N)**
  - N = number of points of input sequence $x[n]$
  - n = the length of the impulse response $h[n]$
- Cost of the FFT based convolution computation in frequency domain ($X[k]H[k]$)
  - **O(N log N)**
  - The total cost of signal processing in the frequency domain is dominated by FFT

Plot of cost vs n, with input size N fixed



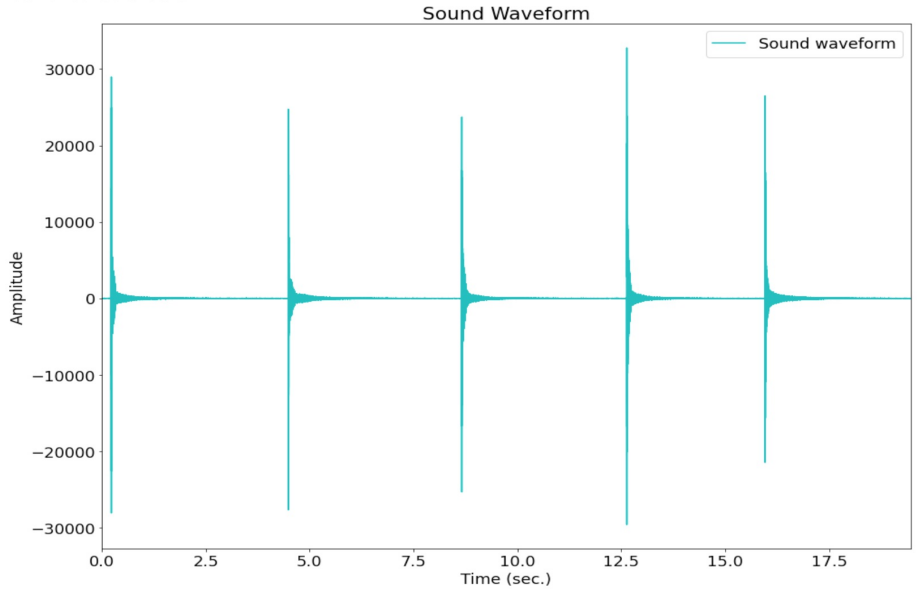
Convolution in frequency domain is faster for long impulse response $h[n]$ (when n gets much larger than log(N))

Spectral Analysis Using FFT

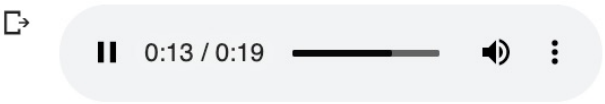
Load Audio File and Play the Sound

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.io import wavfile

# Load sound file from GitHub
!wget https://github.com/R6500/Python-bits/raw/master/Colaboratory/Sounds/Bicycle%20bell%203.wav

# Load the file on an object
data = wavfile.read('Bicycle bell 3.wav')

# Separete the object elements
Fs = data[0] # Sample Rate
x = data[1] # Signal Data
n = x.size # Number of samples
T = 1/Fs # Sample Period T
t = np.arange(0, n)*T # Time vector

# Show information about the object
print('Sample rate:',Fs,'Hz')
print('Total time:',n*T,'s')

plt.plot(t,x,color='c',LineWidth=1.5,label='Sound wave')
plt.xlim(t[0],t[-1])
plt.title('Sound Waveform')
plt.ylabel('Amplitude')
plt.xlabel('Time (sec.)')
plt.legend()
plt.show()
```

```
Sample rate: 48000 Hz
Total time: 19.4594375 s
```

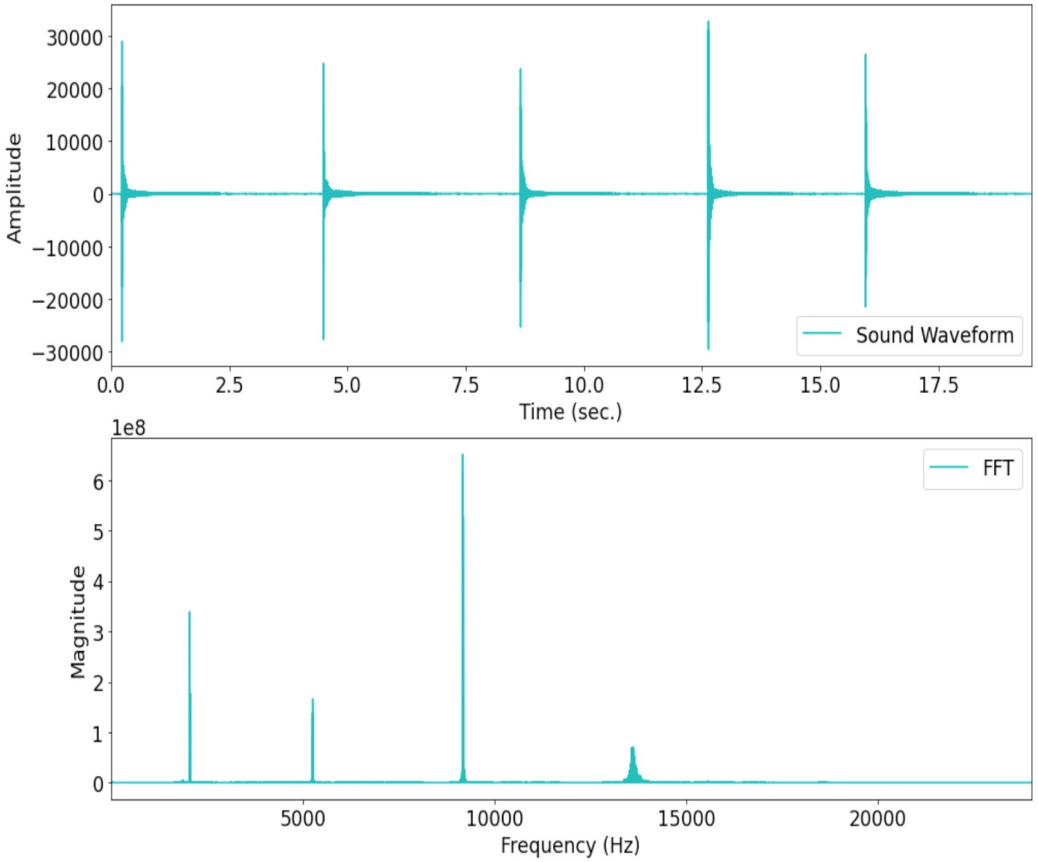



```
[246] # Play the mono sound
      from IPython.display import Audio
      Audio(x,rate=Fs, autoplay=True)
```



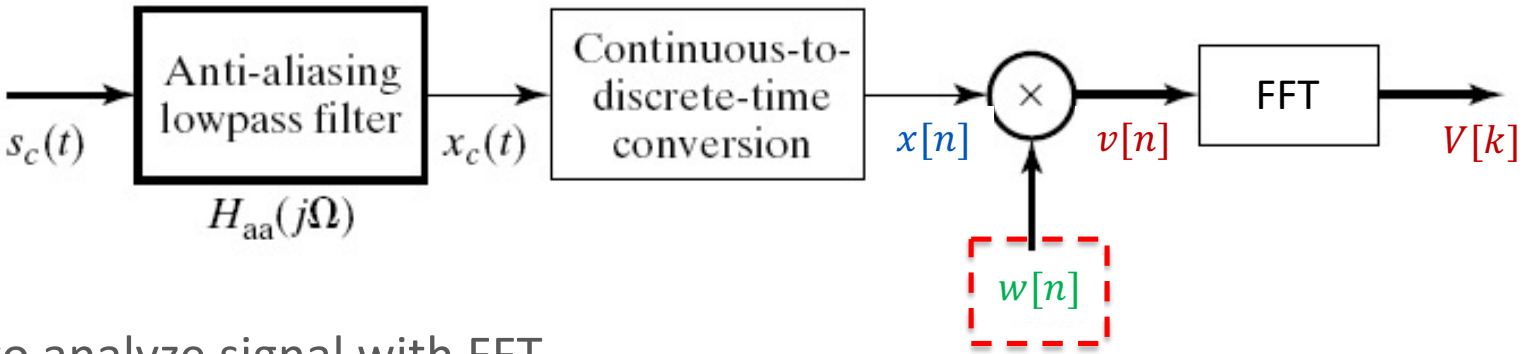
FFT Signal Analysis using Numpy FFT

```
n = x.size
X = np.fft.fft(x,n)
PSD = X * np.conj(X)/n
freq = (Fs/n)*np.arange(n)
L = np.arange(1,np.floor(n/2),dtype='int')

fig,axs = plt.subplots(2,1)

plt.sca(axs[0])
plt.plot(t,x,color='c',LineWidth=1.5,label='Sound Waveform')
plt.xlim(t[0],t[-1])
plt.ylabel('Amplitude')
plt.xlabel('Time (sec.)')
plt.legend()

plt.sca(axs[1])
plt.plot(freq[L],PSD[L],color='c',LineWidth=1.5,label='FFT')
plt.xlim(freq[L[0]],freq[L[-1]])
plt.ylabel('Magnitude')
plt.xlabel('Frequency (Hz)')
plt.legend()
plt.show()
```



Fourier Analysis of Signals Using FFT

- One major application of the FFT: Analyze Signals
- Let's analyze frequency content of a continuous-time signal



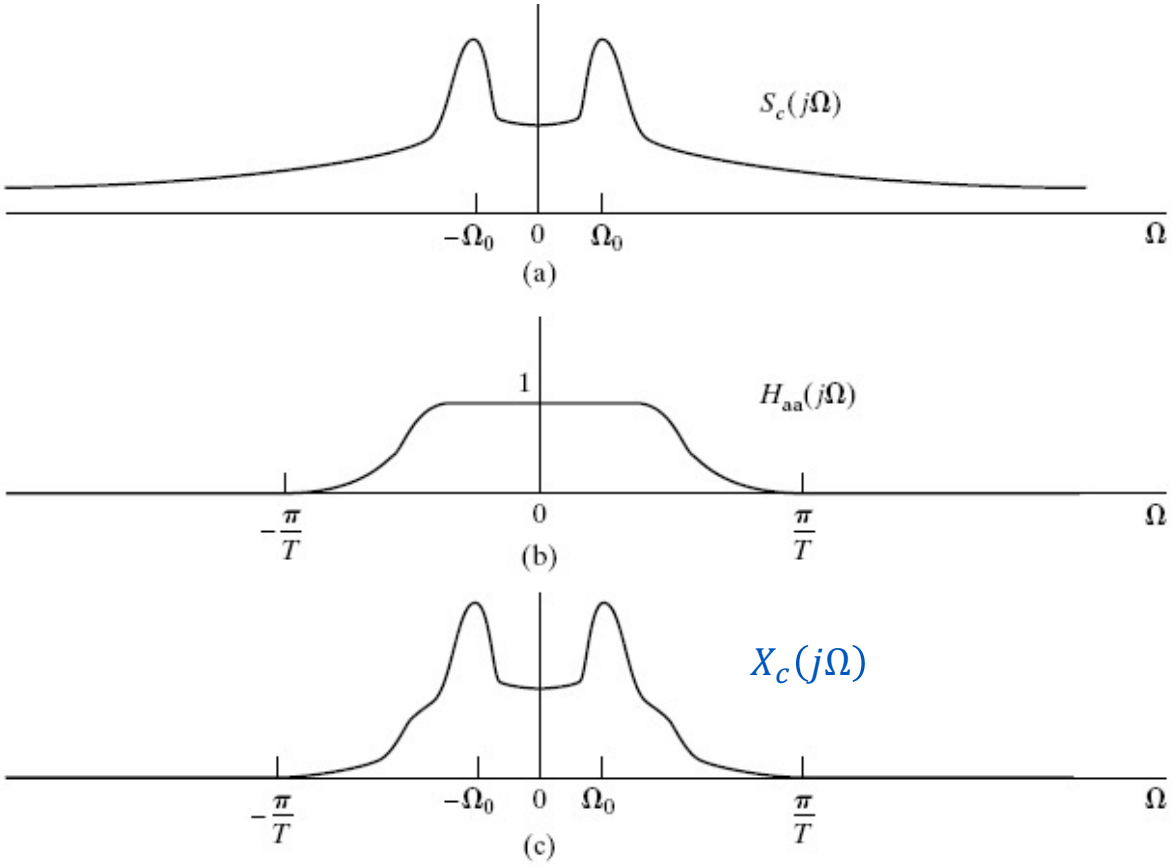
- Steps to analyze signal with FFT
  1. Remove high-frequencies to prevent aliasing after sampling
  2. Sample the signal $x_c(t)$ to convert to discrete-time signal $x[n]$
  3. Window to limit the duration of the signal
  4. Take FFT of the resulting signal

Example of Signal Analysis Using FFT (1)

Continuous-time signal

Anti-aliasing filter

Signal after filtering



Example of Signal Analysis Using FFT (2)

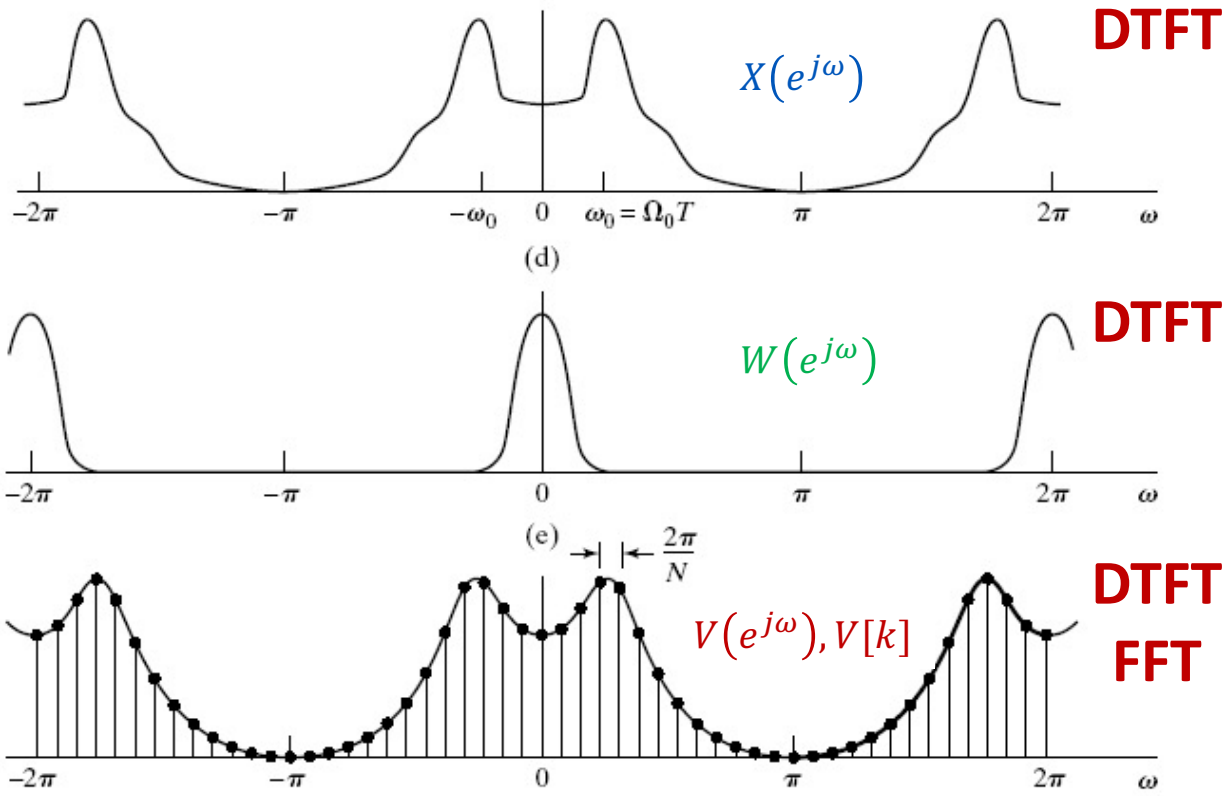
DTFT of the Sampled DT signal

Frequency response of window

Windowed and sampled Fourier Transform

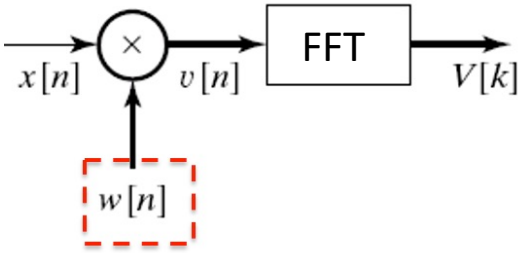
$V(e^{j\omega}) = X(e^{j\omega}) * W(e^{j\omega})$

$V[k] = V(e^{j\omega})\Big|_{\omega=\frac{2\pi}{N}k}$



Effect of Windowing on Sinusoidal Signals

- The effects of anti-aliasing filtering and sampling is known
- We will analyze the effect of windowing
- Choose a simple signal to analyze this effect: sinusoids
  - $s_c(t) = \mathrm{A}_0\cos(\Omega_0 t + \theta_0) + \mathrm{A}_1\cos(\Omega_1 t + \theta_1)$
- Assume ideal sampling and no aliasing we get
  - $x[n] = \mathrm{A}_0\cos(\omega_0 n + \theta_0) + \mathrm{A}_1\cos(\omega_1 n + \theta_1)$
- And after windowing we have
  - $v[n] = x[n]w[n] = \mathrm{A}_0 w[n]\cos(\omega_0 n + \theta_0) + \mathrm{A}_1 w[n]\cos(\omega_1 n + \theta_1)$

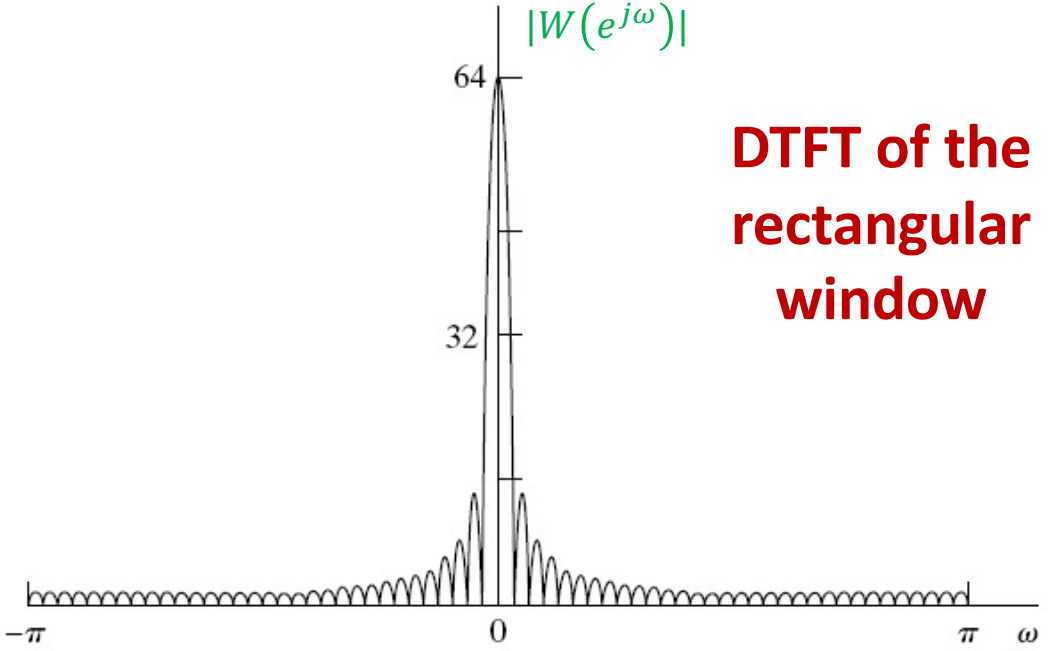


- Calculate the DTFT of $v[n]$ by writing out the cosines as

$$v[n] = \frac{A_0}{2} w[n] e^{j\theta_0} e^{j\omega_0 n} + \frac{A_0}{2} w[n] e^{-j\theta_0} e^{-j\omega_0 n} + \frac{A_1}{2} w[n] e^{j\theta_1} e^{j\omega_1 n} + \frac{A_1}{2} w[n] e^{-j\theta_1} e^{-j\omega_1 n}$$

$$V(e^{j\omega}) = \frac{A_0}{2} e^{j\theta_0} W(e^{j(\omega-\omega_0)n}) + \frac{A_0}{2} e^{-j\theta_0} W(e^{j(\omega+\omega_0)n}) + \frac{A_1}{2} e^{j\theta_1} W(e^{j(\omega-\omega_1)n}) + \frac{A_1}{2} e^{-j\theta_1} W(e^{j(\omega+\omega_1)n})$$

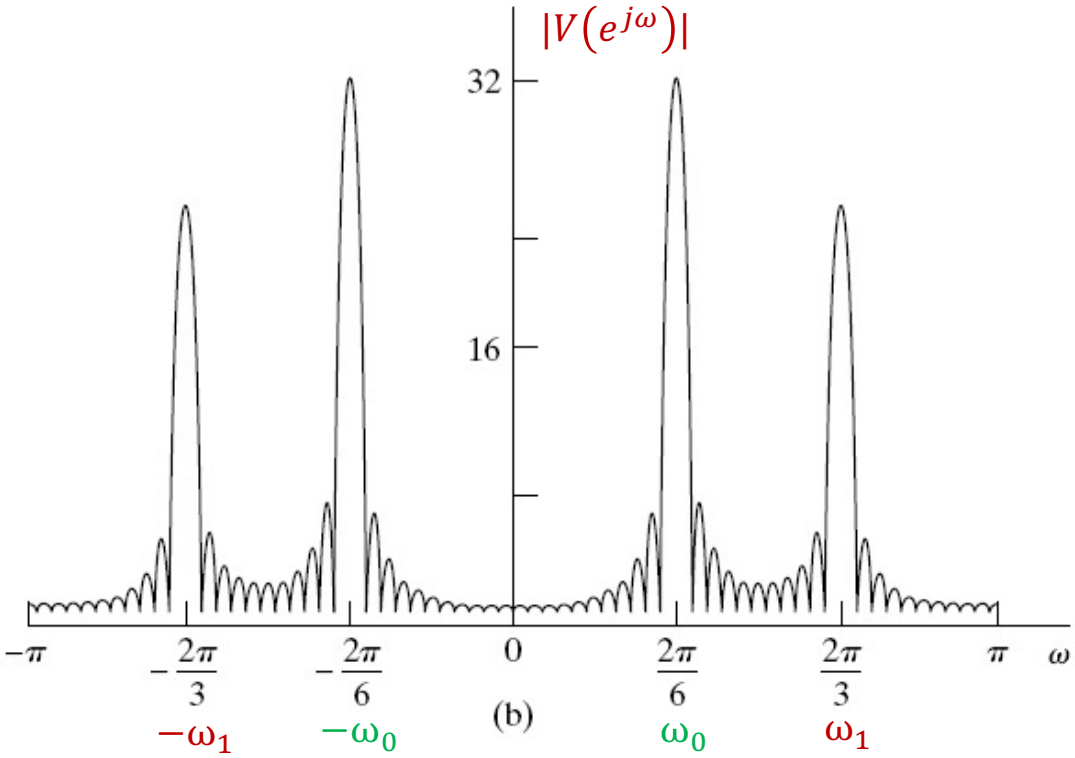
- Consider a rectangular window w[n] of length 64
- Assume 1/T=10 kHz, A_0=1 and A_1=0.75 and phases to be zero
- Magnitude of the DTFT of the window



DTFT of the rectangular window

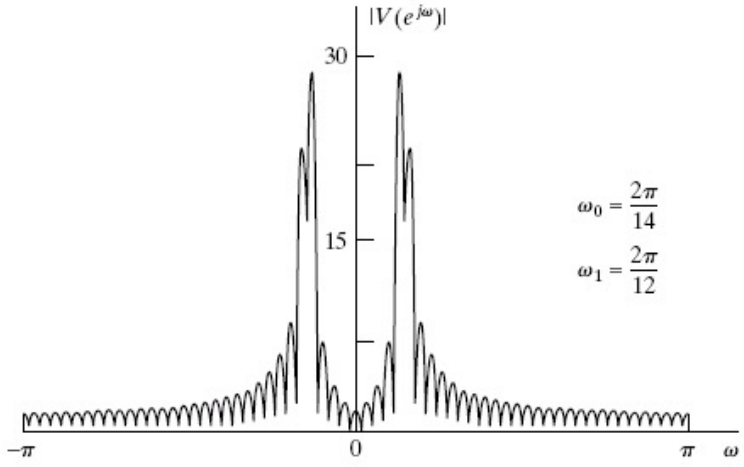
Magnitude of the DTFT of the sampled signal

- We expect to see dirac function at input frequencies
- Due to windowing we see, instead, the response of the window
- Note that both tones will affect each other due to the smearing
  - This is called **leakage**: small in this example

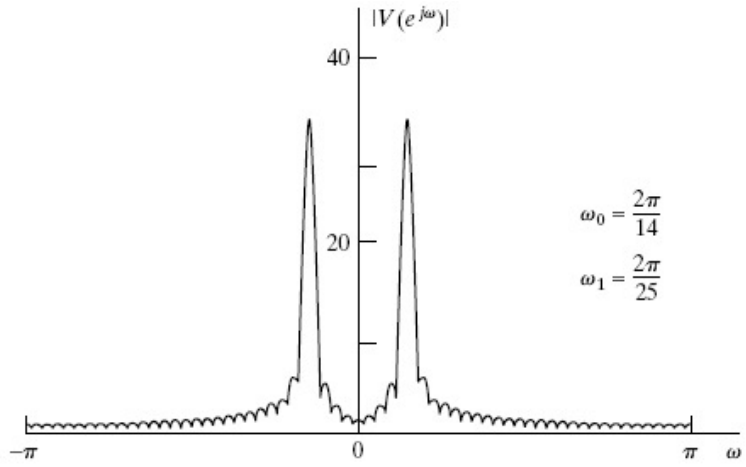


Effect of Windowing on Sinusoidal Signals

- **If the input tones are close to each other**



The tones are **so close** that they have considerable affect on each others magnitude



The tones are **too close** to even separate in this case

- They cannot be resolved using this particular window

The Effect of Spectral Sampling

- FFT samples the DTFT with *N* equally spaced samples at

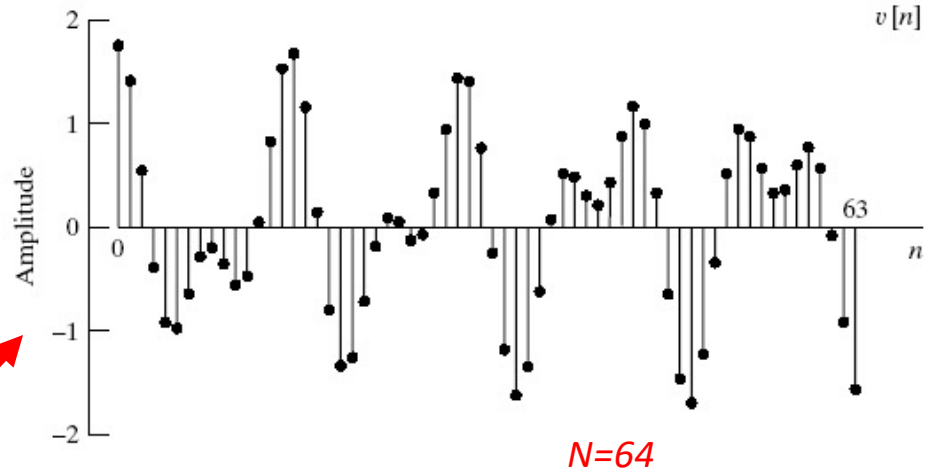
$$\omega_k = \frac{2\pi k}{N} \qquad k = 0,1,\ldots,N-1$$

- Or in terms of continuous-frequency

$$\Omega_k = \frac{2\pi k}{NT} \qquad k = 0,1,\ldots,N/2$$

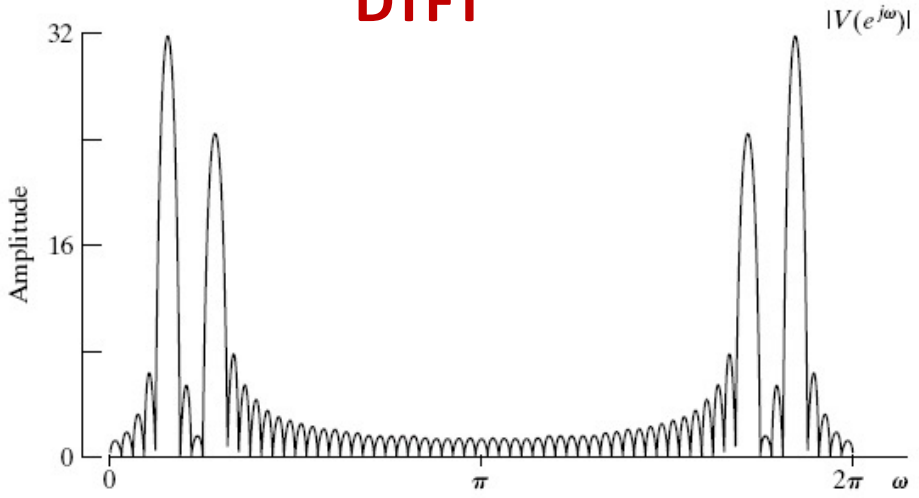
- Example: Signal after rectangular windowing

$$v[n] = \begin{cases} \cos\left(\frac{2\pi}{14}n\right) + 0.75\cos\left(\frac{4\pi}{15}n\right) & 0 \le n \le 63 \\ 0 & else \end{cases}$$

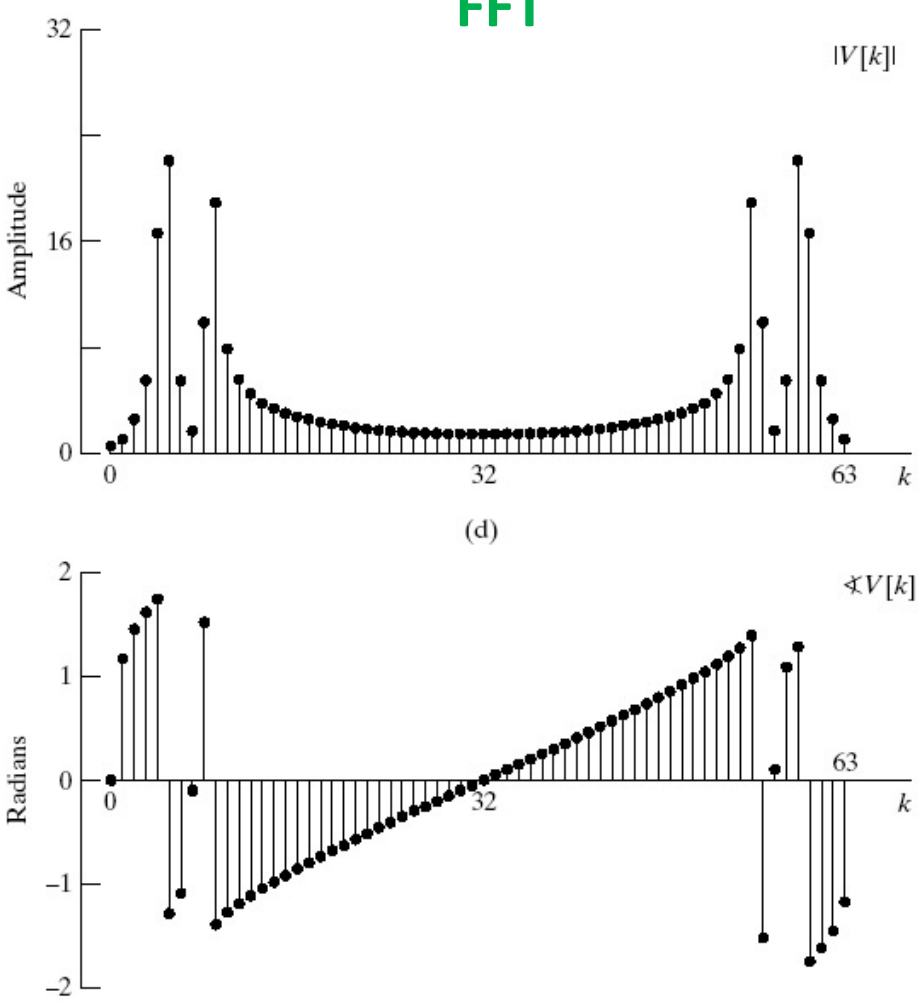


N=64

DTFT



FFT



- Note the peak of the DTFTs are in between samples of the FFT

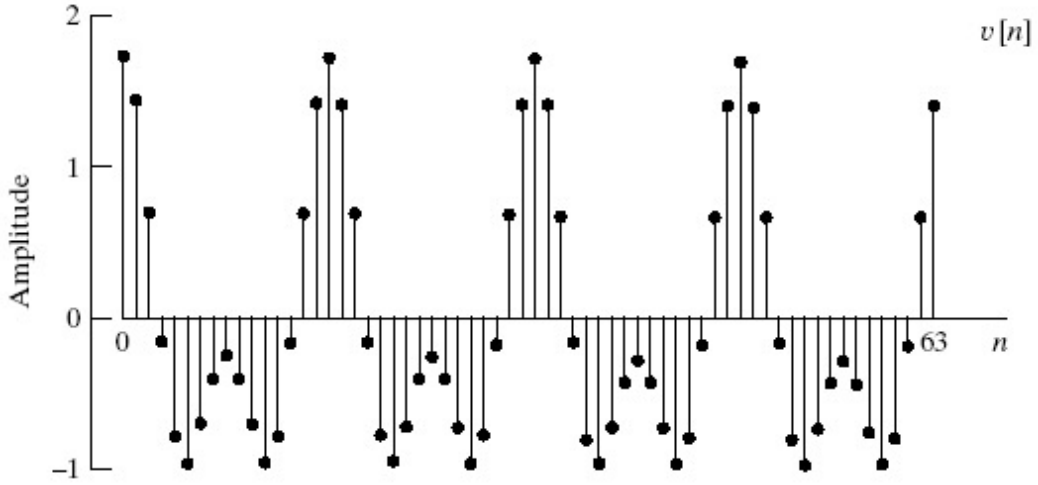
$$\omega_1 = \frac{2\pi}{14}n = \frac{2\pi}{64}k \Rightarrow k = 4.5714$$

$$\omega_2 = \frac{4\pi}{15}n = \frac{2\pi}{64}k \Rightarrow k = 8.5333$$

- FFT does not necessary reflect real magnitude of spectral peaks

- Let's consider another sequence after windowing we have

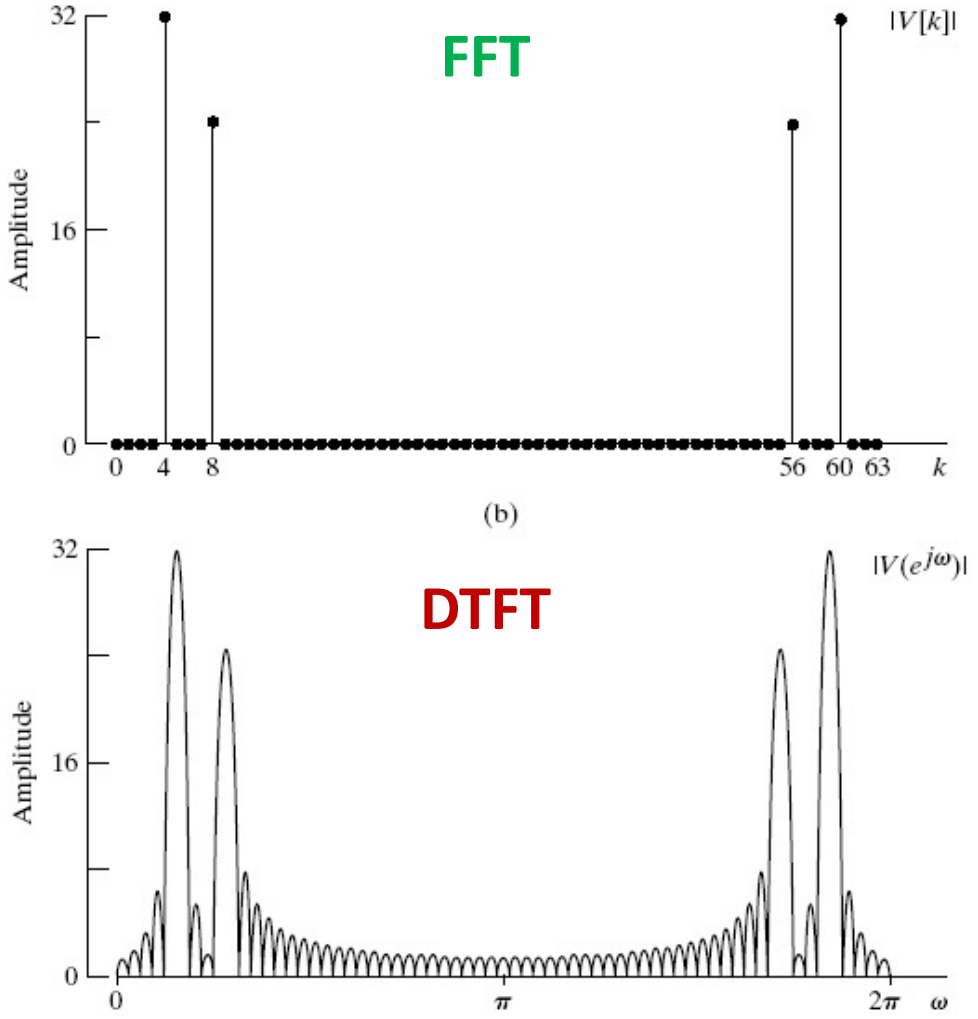
$$v[n] = \begin{cases} \cos\left(\frac{2\pi}{16}n\right) + 0.75\cos\left(\frac{2\pi}{8}n\right) & 0 \le n \le 63 \\ 0 & else \end{cases}$$



$$\omega_1 = \frac{2\pi}{16}n = \frac{2\pi}{64}k \Rightarrow k = 4$$

$$\omega_2 = \frac{2\pi}{8}n = \frac{2\pi}{64}k \Rightarrow k = 8$$

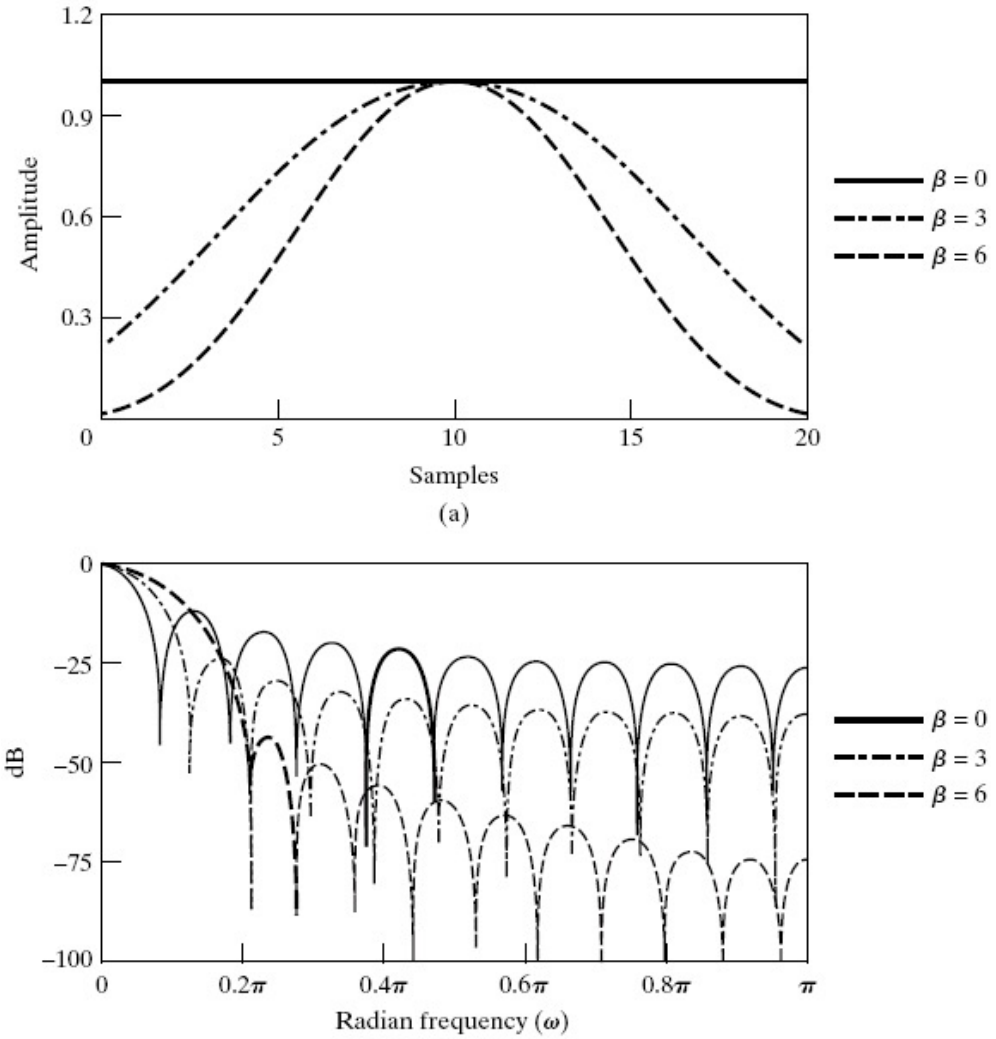
- In this case *N* samples cover exactly 4 and 8 periods of the tones
- The samples correspond to the peak of the lobes
- The magnitude of the peaks are accurate
- Note that we don't see the side lobes in this case



Window Functions

- Two factors are determined by the window function
  - **Resolution**: influenced mainly by the main lobe width
  - **Leakage**: relative amplitude of side lobes versus main lobe
- We will learn more in FIR filter design lecture that we can choose various windows to trade-off these two factors
- Example: Kaiser window

Kaiser window

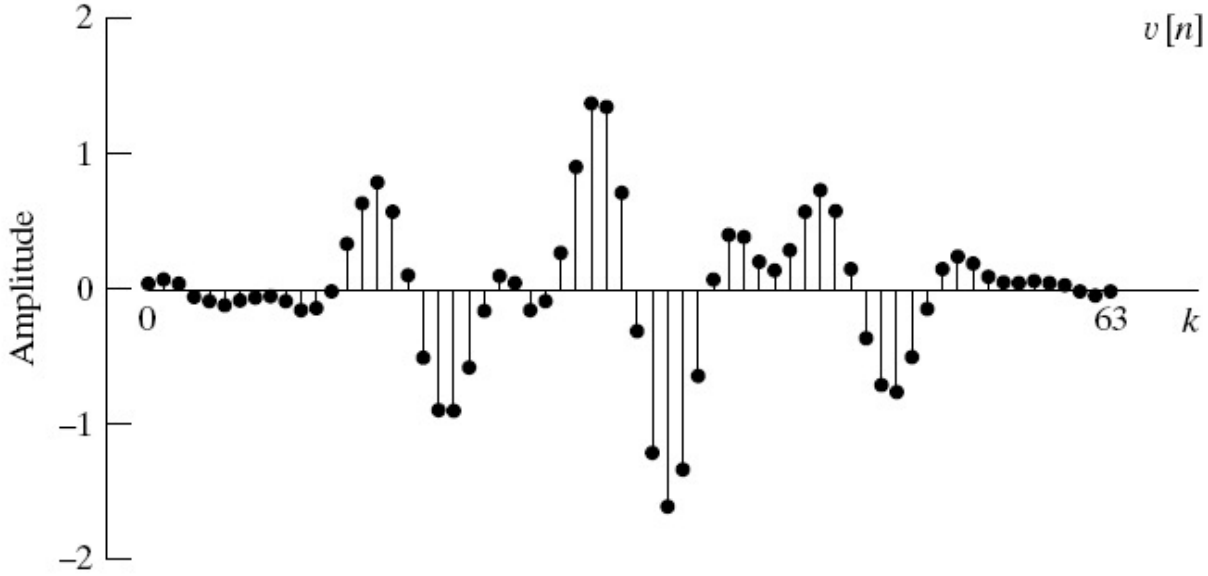


Example: FFT Analysis with Kaiser Window (1)

- The windowed signal is given as

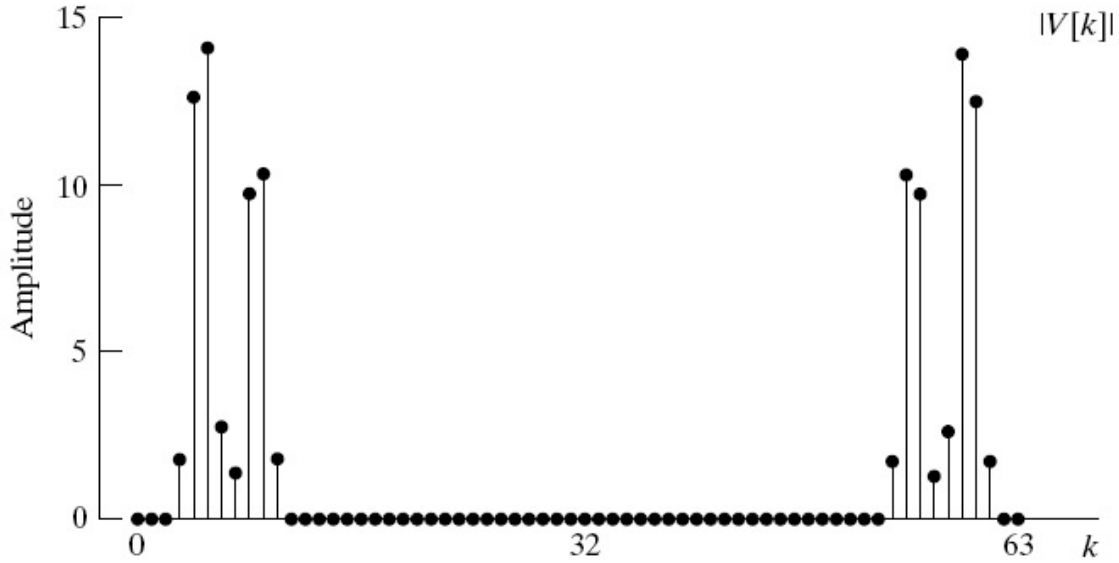
$$v[n] = w_k[n]\cos\left(\frac{2\pi}{14}n\right) + 0.75w_k[n]\cos\left(\frac{4\pi}{15}n\right)$$

- Where $w_k[n]$ is a Kaiser window with β=5.48 for a relative side lobe amplitude of -40 dB
- The windowed signal



Example: FFT Analysis with Kaiser Window (2)

- FFT with this Kaiser window

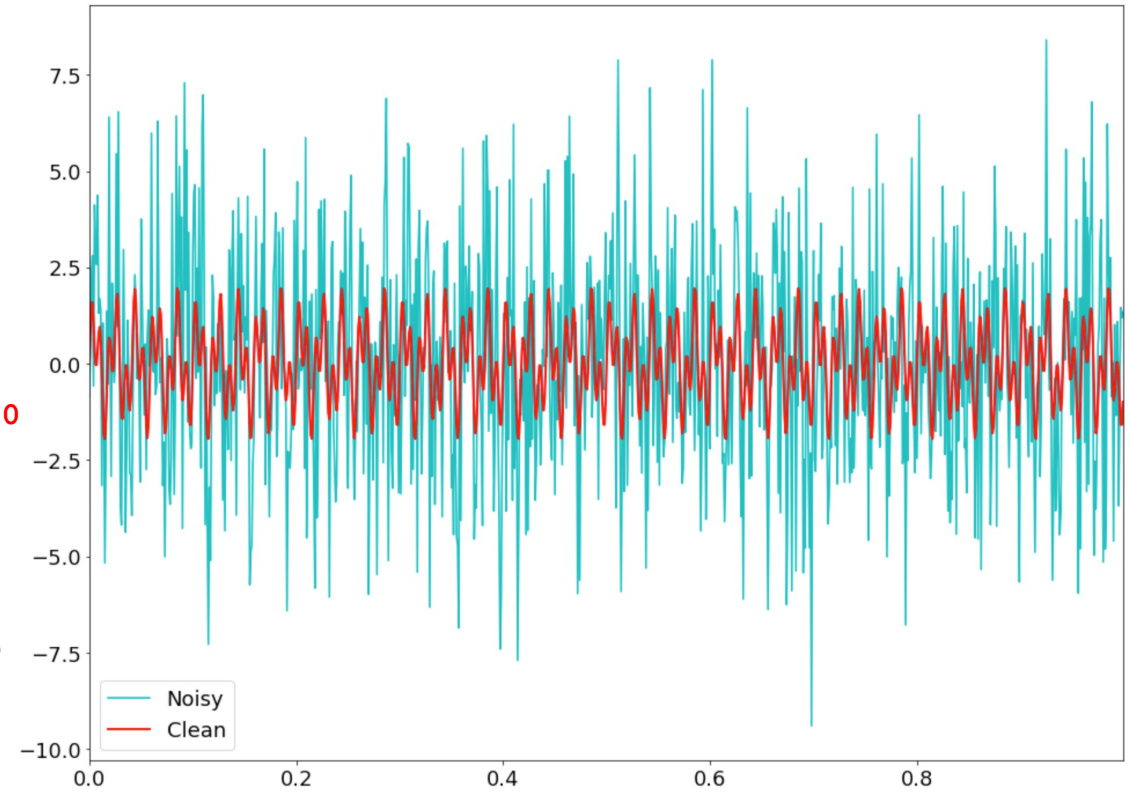


- The two tones are clearly resolved with the Kaiser window

Signal Denoising Using FFT

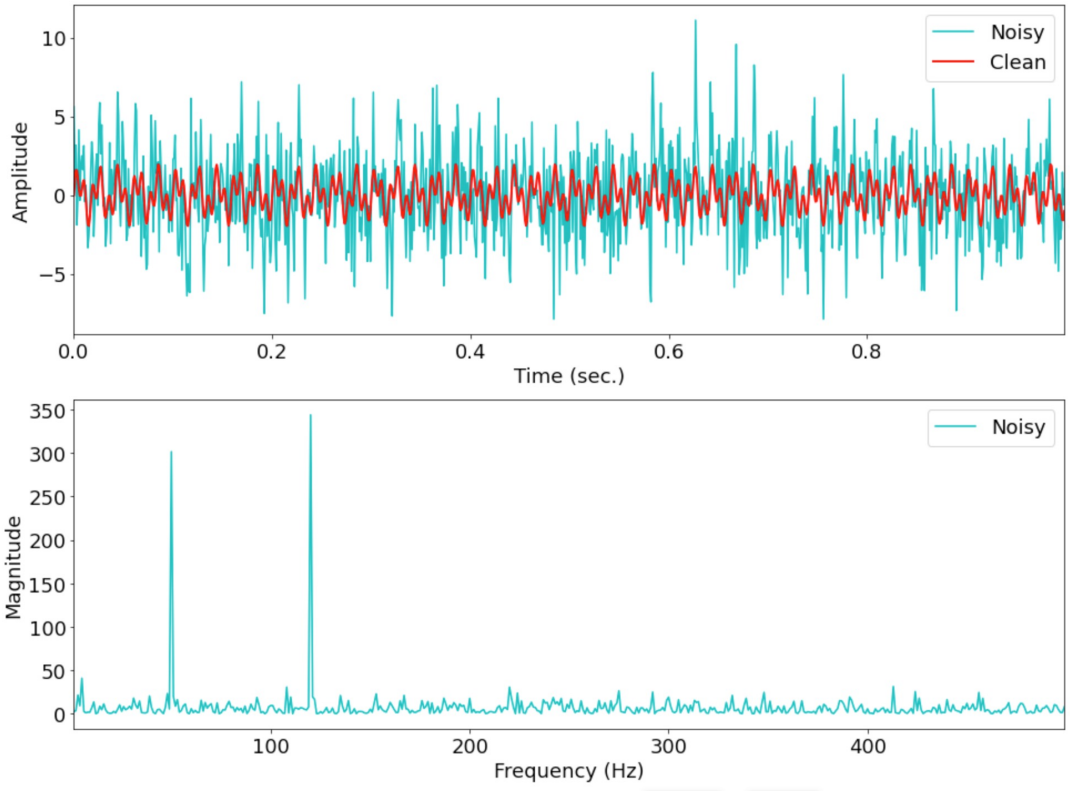
Audio Signal Denoising Using FFT

```
import numpy as np
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize']=[16,12]
plt.rcParams.update({'font.size':18})

# Create a simple signal with two frequencies
Fs = 1000 # The sampling frequency is 1kHz
dt = 1/Fs # 0.001
t = np.arange(0,1,dt)

# Sum of two sin waves (50Hz and 120Hz)
x = np.sin(2*np.pi*50*t) + np.sin(2*np.pi*120
x_clean = x

x = x + 2.5*np.random.randn(len(t))

plt.plot(t,x,color='c',LineWidth=1.5,label='Noisy')
plt.plot(t,x_clean, color='r',LineWidth=2,label='Clean')
plt.xlim(t[0],t[-1])
plt.legend()
plt.show()
```



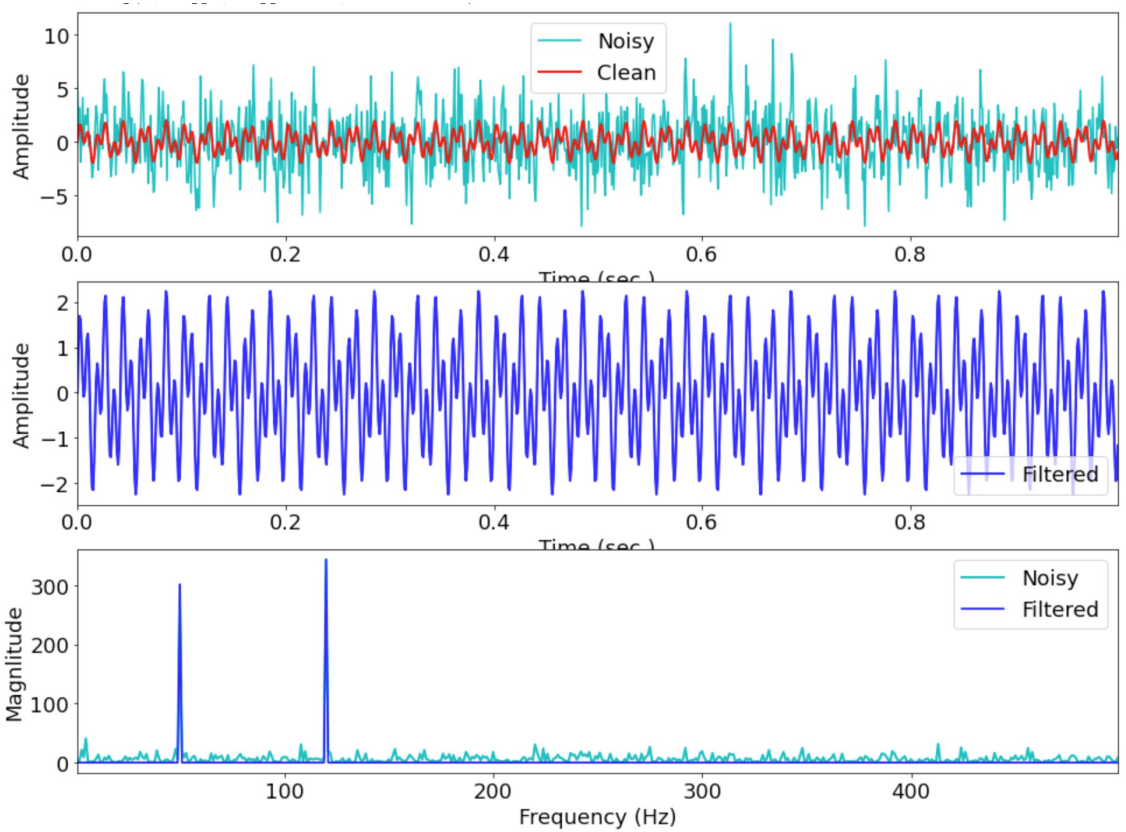
Compute Power Spectrum Density of the Noisy Signal by FFT

```
n = len(t)
X = np.fft.fft(x,n)
PSD = X * np.conj(X)/n
freq = (1/(dt*n))*np.arange(n)
L = np.arange(1,np.floor(n/2),dtype='int')

fig,axs = plt.subplots(2,1)

plt.sca(axs[0])
plt.plot(t,x,color='c',LineWidth=1.5,label='Noisy')
plt.plot(t,x_clean,color='r',LineWidth=2,label='Clean')
plt.xlim(t[0],t[-1])
plt.ylabel('Amplitude')
plt.xlabel('Time (sec.)')
plt.legend()

plt.sca(axs[1])
plt.plot(freq[L],PSD[L],color='c',LineWidth=1.5,label='Noisy')
plt.xlim(freq[L[0]],freq[L[-1]])
plt.ylabel('Magnitude')
plt.xlabel('Frequency (Hz)')
plt.legend()
plt.show()
```



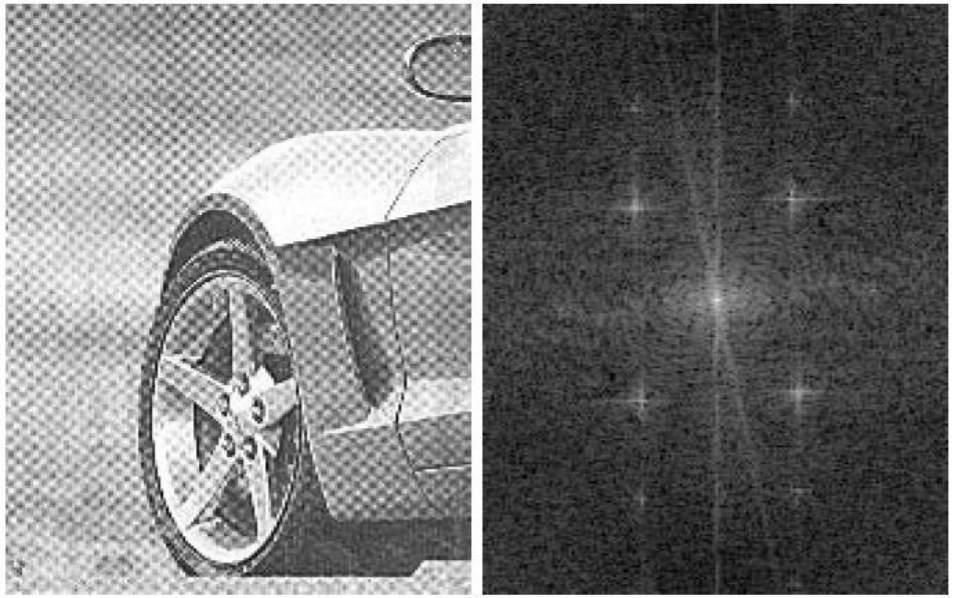
Use the PSD to Filter Out Noise

```
indices = PSD > 100
PSDclean = PSD * indices
X = indices * X
x_filtered = np.fft.ifft(X)

## Plots
fig,axs = plt.subplots(3,1)
plt.sca(axs[0])
plt.plot(t,x,color='c',LineWidth=1.5,label='Noisy')
plt.plot(t,x_clean,color='r',LineWidth=2,label='Clean')
plt.xlim(t[0],t[-1])
plt.ylabel('Amplitude')
plt.xlabel('Time (sec.)')
plt.legend()
plt.sca(axs[1])
plt.plot(t,x_filtered,color='b',LineWidth=2,label='Filtered')
plt.xlim(t[0],t[-1])
plt.ylabel('Amplitude')
plt.xlabel('Time (sec.)')
plt.legend()
plt.sca(axs[2])
plt.plot(freq[L],PSD[L],color='c',LineWidth=2,label='Noisy')
plt.plot(freq[L],PSDclean[L],color='b',LineWidth=1.5,label='Filtered')
plt.xlim(freq[L[0]],freq[L[-1]])
plt.ylabel('Magnlitude')
plt.xlabel('Frequency (Hz)')
plt.legend()
plt.show()
```



Notch Filters by 2D-FFT (Optional)

- A filter that rejects (or passes) specific frequencies
- Example: periodic noise corresponds to spikes or lines in the Fourier domain
- Can design a filter with zeros at those frequencies … this will remove the noise
- Examples:
  - Image mosaics
  - Scan line noise
  - Halftoning noise
    - (moire patterns)



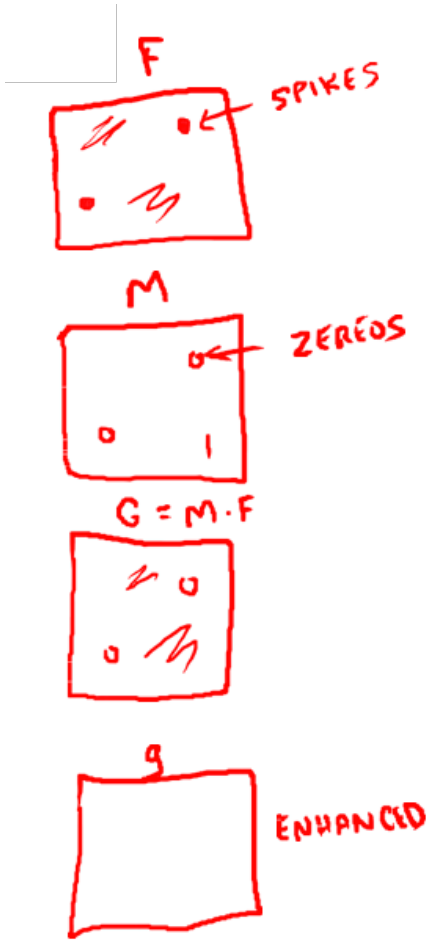
Steps in Notch Filtering

- Look at spectrum $|F(u,v)|$ of noisy image $f(x,y)$, find frequencies corresponding to the noise
- Create a mask image $M(u,v)$ with notches (zeros) at those places, 1's elsewhere
- Multiply mask with original image transform; this zeros out noise frequencies

$$G(u,v) = M(u,v)\, F(u,v)$$

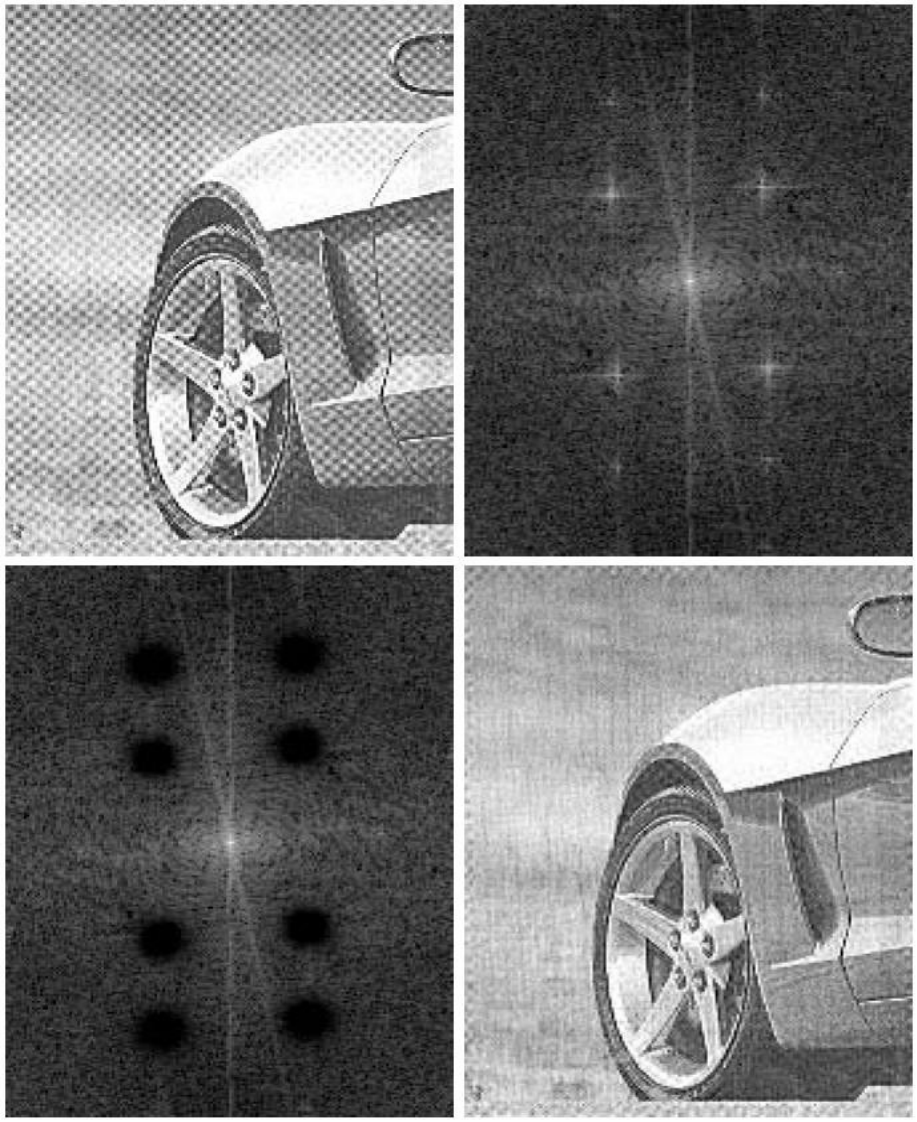
- Take inverse Fourier transform to get restored image

$$g(x,y) = \mathcal{F}^{-1}(G(u,v))$$



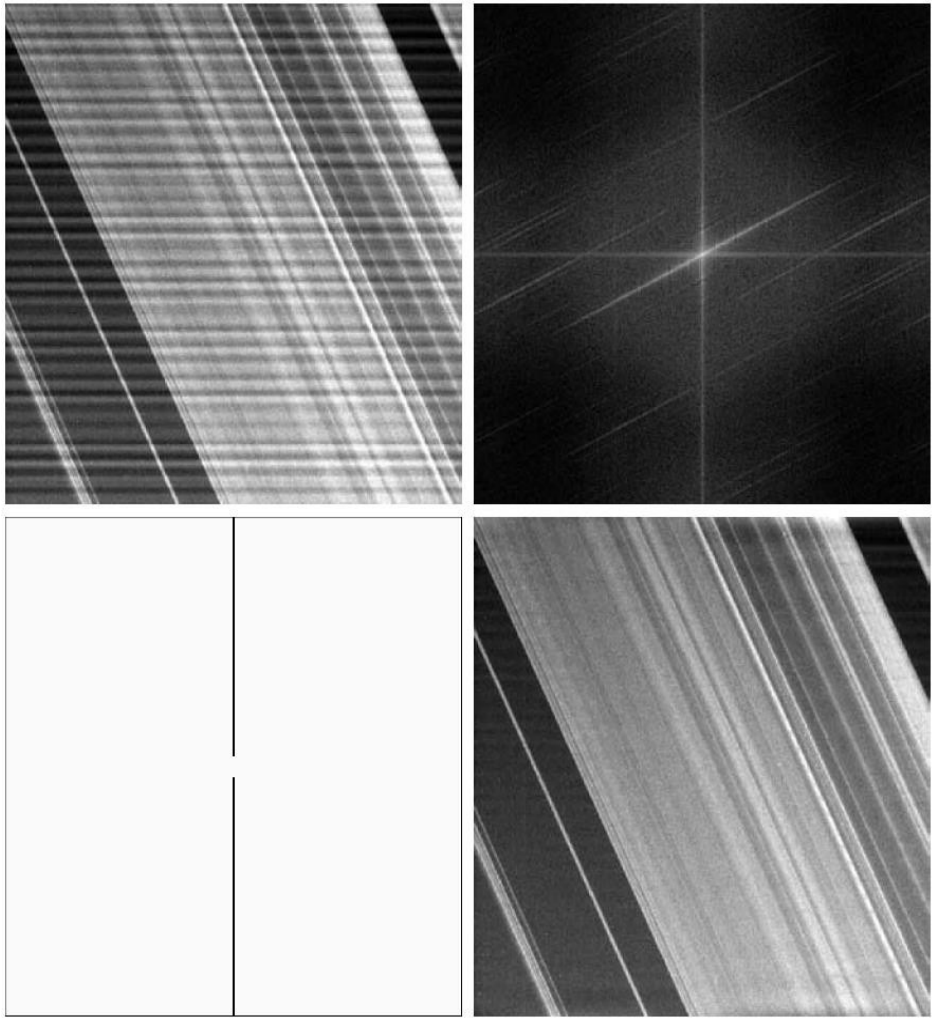
Example Notch Filters (1)

- A Butterworth notch reject filter $D_0 = 3$ and $n = 4$ for notch pairs

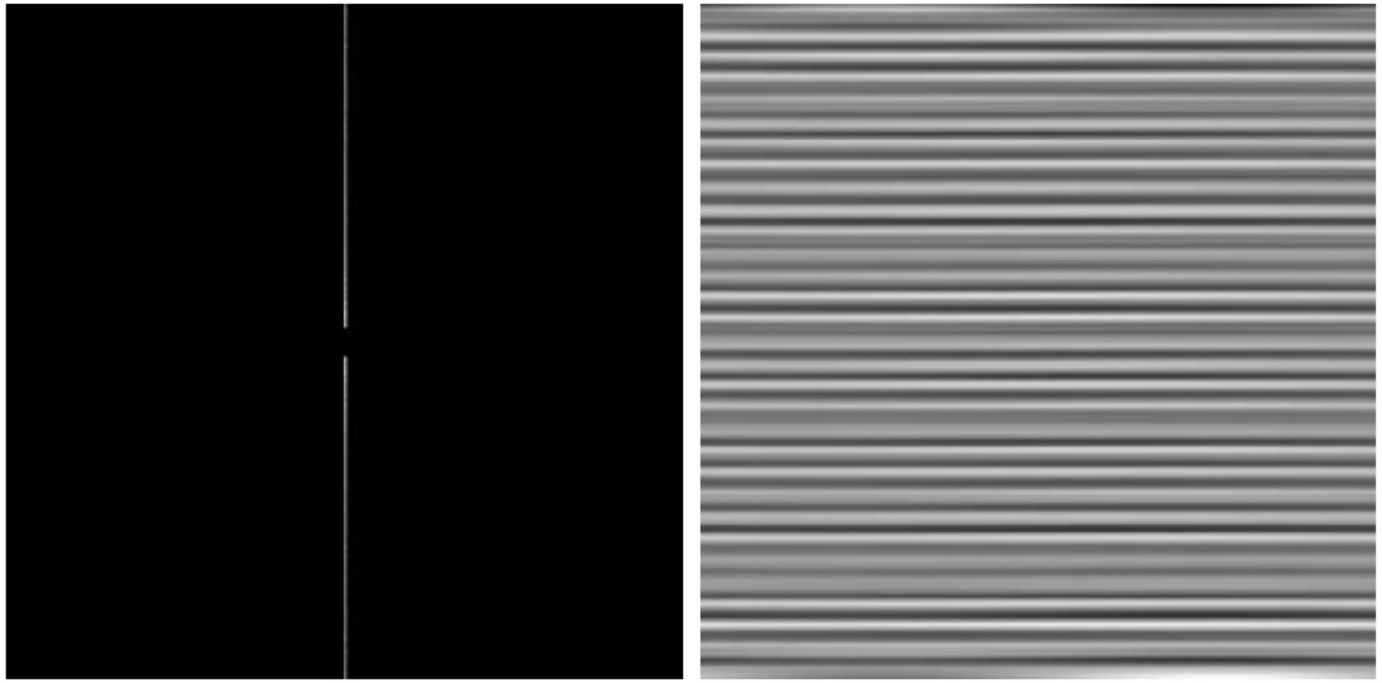


Example Notch Filters (2)

- Example of horizontal scan lines
- Create a notch of vertical lines in frequency domain



Example Notch Filters (3)



a b

FIGURE 4.66
(a) Result (spectrum) of applying a notch pass filter to the DFT of Fig. 4.65(a). (b) Spatial pattern obtained by computing the IDFT of (a).

Spectrogram (Optional)

Short-Time Fourier Transform (STFT)

- STFT can be computed as a *N*-point windowed DFT as follows (in practice FFT is usually used to compute the DFT in each frame):

$$X[k,m] = \sum_{n=0}^{N-1} x[m\,s + n]w[n]e^{-j\omega_k n}$$

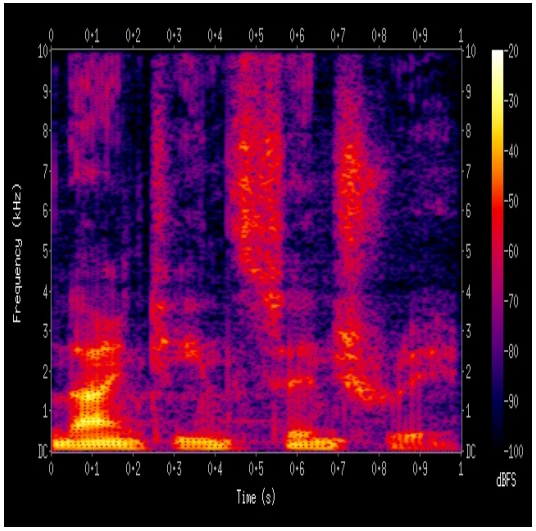
  - ω_k is the discrete angular frequency, $\omega_k = \frac{2\pi k}{N}, k = 0,1,\dots,N-1$
  - m is the time-frame index
  - s is the hop size
  - $w[n]$ is a window function, such as rectangular, Hann windows

Spectrogram : Time-Frequency Analysis Using STFT

- Spectrogram of a digital signal can be computed as magnitude squared STFT:

$$Spectrogram\{x[n]\} = |X[k,m]|^2$$

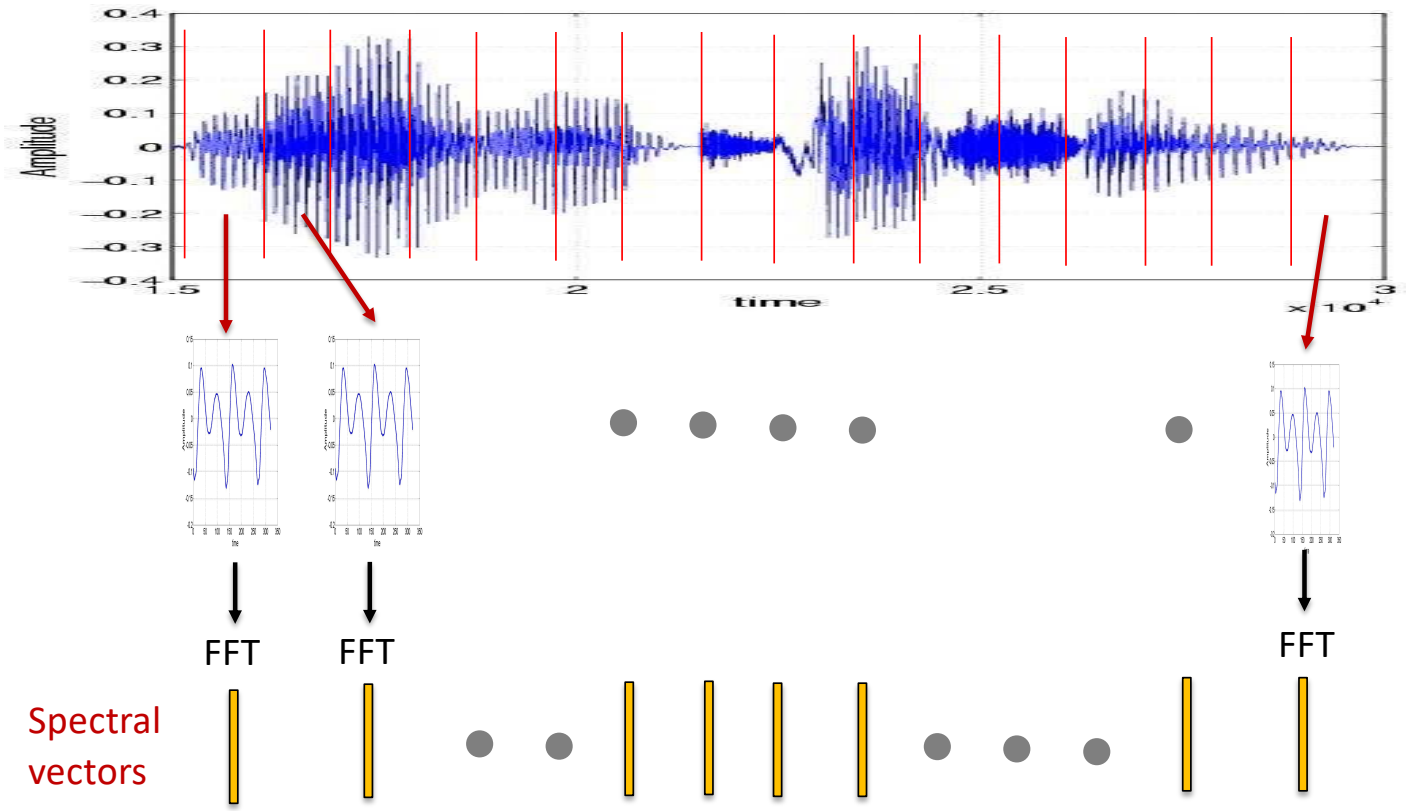
- Begin to be used since 1940s
- Spectrograms are a way of viewing hundreds of sequential power spectra, so we can see how the spectrum of a signal changes over time.
  - This is done by turning the spectrum sideways, reducing each bar on the spectrum to a dot, and using the color of the dot to symbolize the height of the bar (i.e., the amplitude or intensity of the component frequency)

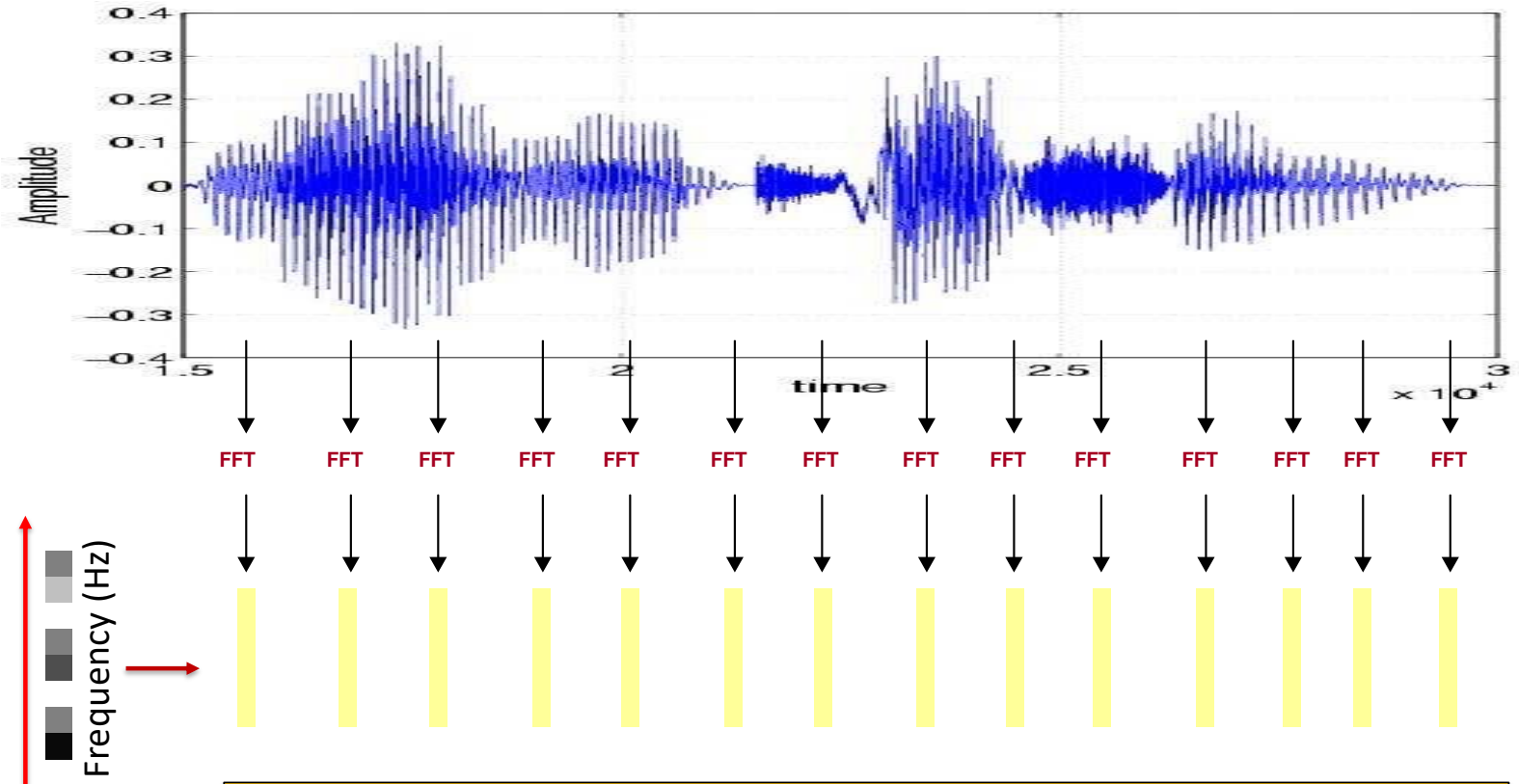


Spectrogram of the spoken words "nineteenth century"

Construction of Spectrogram

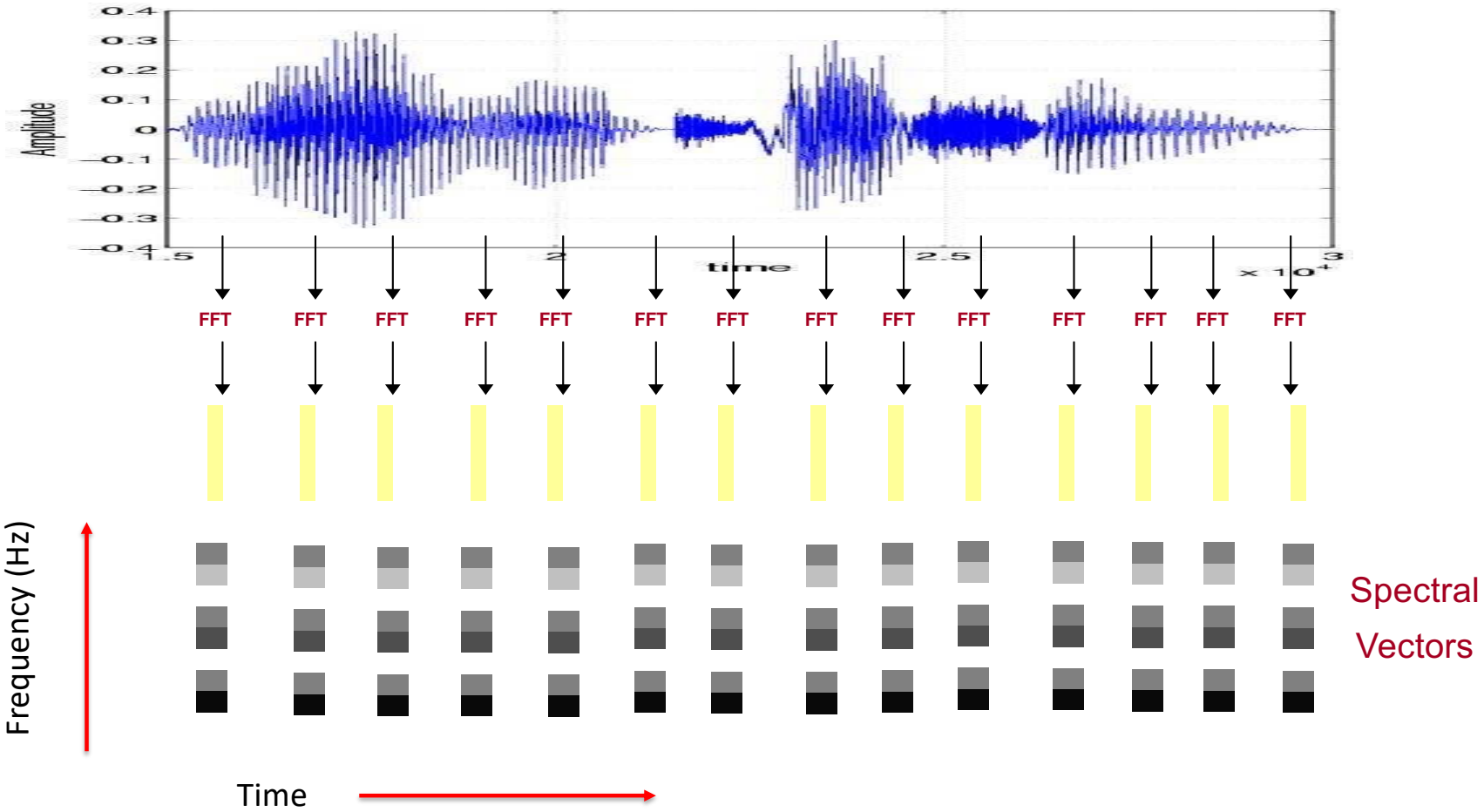
- Spectrogram is a digital signal representation by **a sequence of spectral vectors**





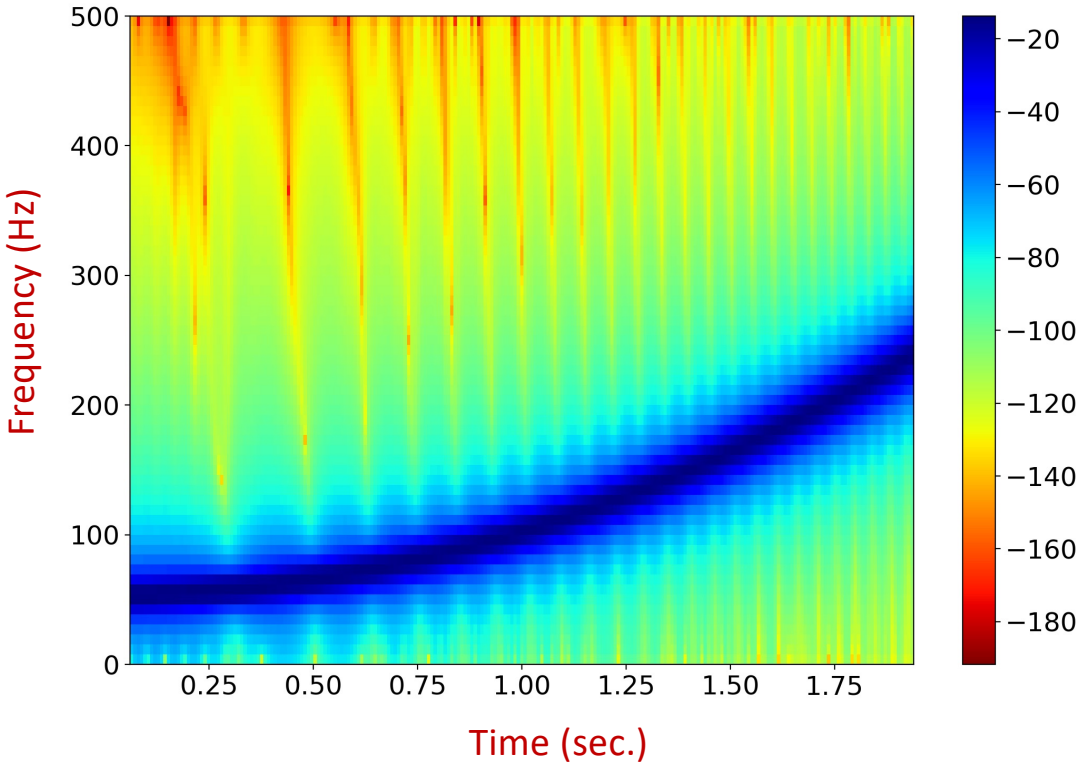
- MAP spectral magnitude to a grey level (0-255) value. 0 represents black and 255 represents white.
- Higher the magnitude, darker the corresponding region.
- Color heat map is commonly used today.

Spectrogram: Digital Signal Represented a Sequence of Spectral Vectors



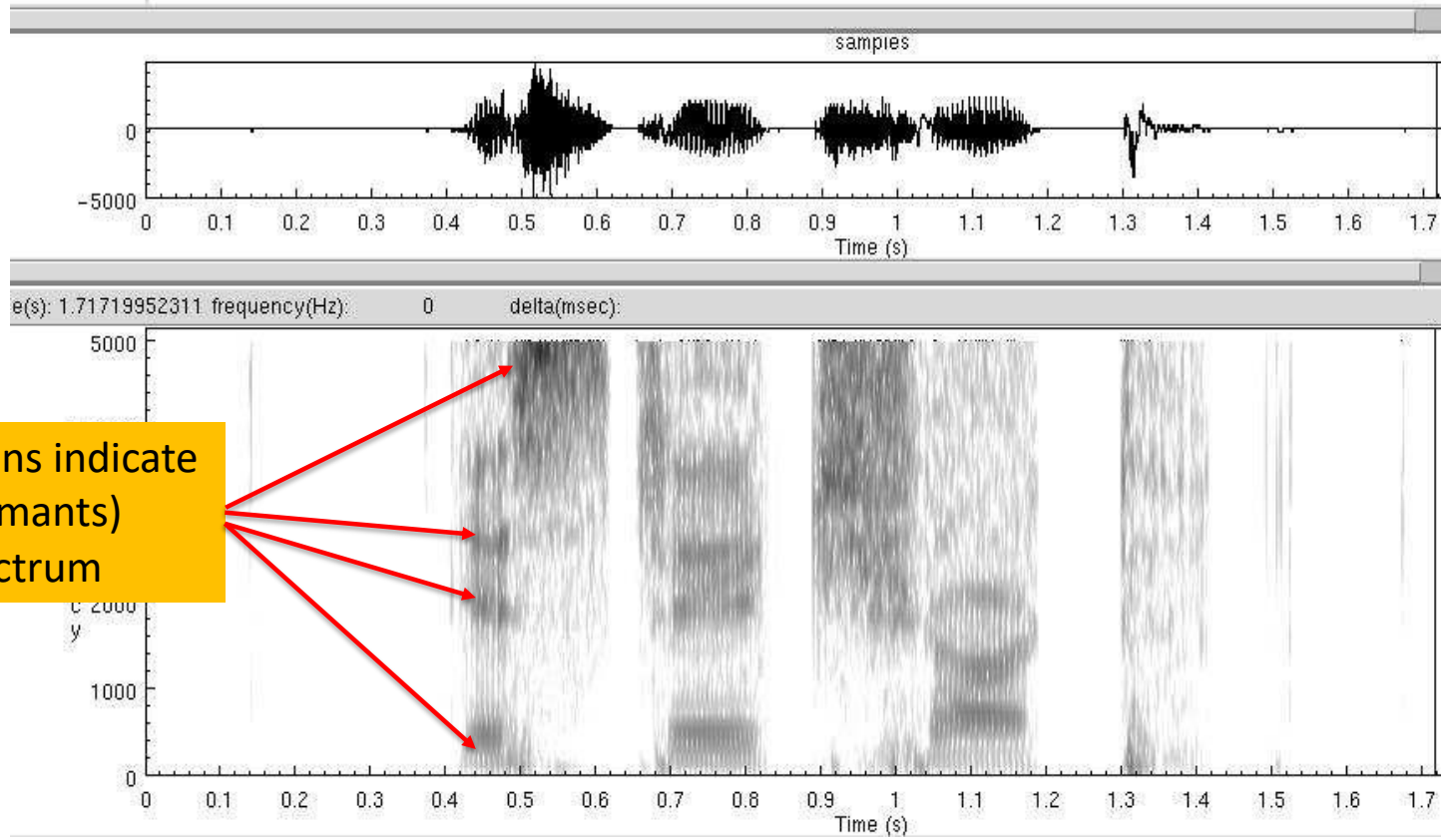
Spectrogram : Python Code

```
import numpy as np
from IPython.display import Audio
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize']=[12,8]
plt.rcParams.update({'font.size':18})

Fs = 1000
dt = 1/Fs
t = np.arange(0,2,dt)
f0 = 50
f1 = 250
t1 = 2
x = np.cos(2*np.pi*t*(f0 + (f1-
f0)*np.power(t,2)/(3*t1**2)))

plt.specgram(x, NFFT=128, Fs=1/dt, noverlap=120,
cmap='jet_r')
plt.colorbar()
plt.show()
```



https://www.youtube.com/watch?v=TJGlxdW7Fb4&list=PLMrJAkhIeNNT_Xh3Oy0Y4LTj0Oxo8GqsC&index=29

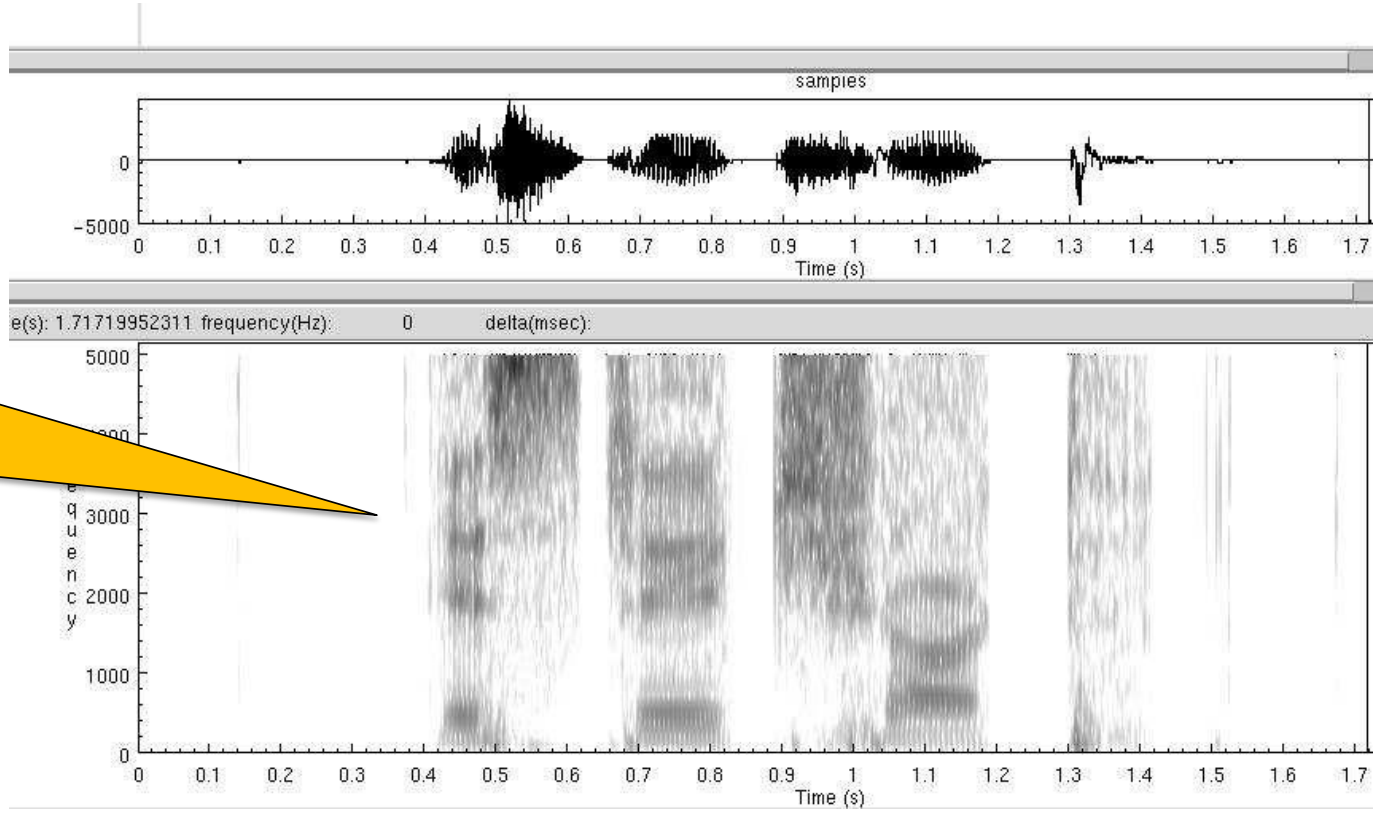
Some Real Spectrograms



Dark regions indicate peaks (formants) in the spectrum

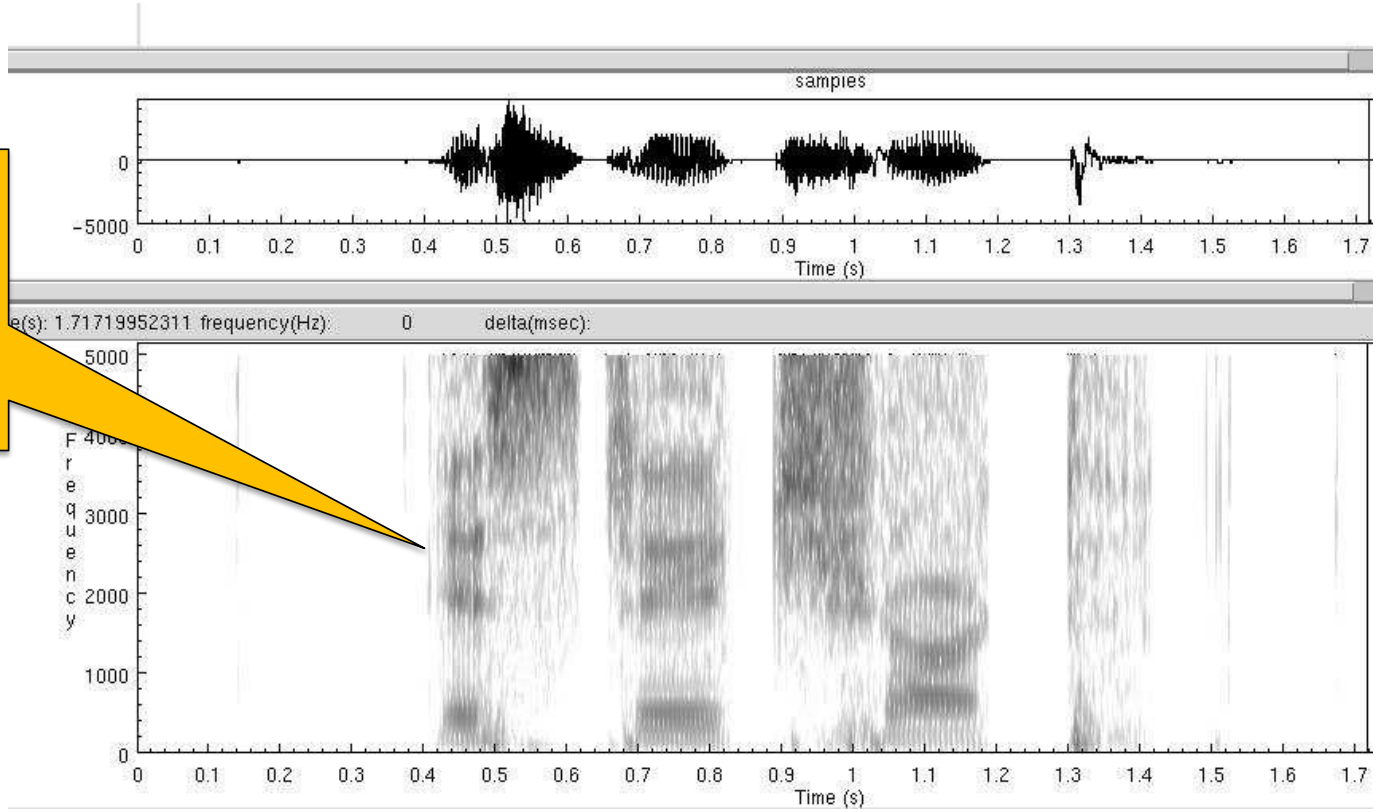
Why we are bothered about spectrograms?

Phones and their properties are better observed in spectrogram



Why we are bothered about spectrograms?

Sounds can be identified much better by the Formants and by their transitions



Before deep neural network ear, Hidden Marko Models implicitly model these spectrograms to perform speech recognition.

Usefulness of Spectrogram

- Time-Frequency representation of the digital signal
- Spectrogram is a tool to study speech and audio signals
- Phones and their properties are visually studied by phoneticians
- Hidden Markov Models implicitly model spectrograms for speech to text systems
- Useful for evaluation of text to speech systems
  - A high-quality text to speech system should produces synthesized speech whose spectrograms should nearly match with the natural sentences.

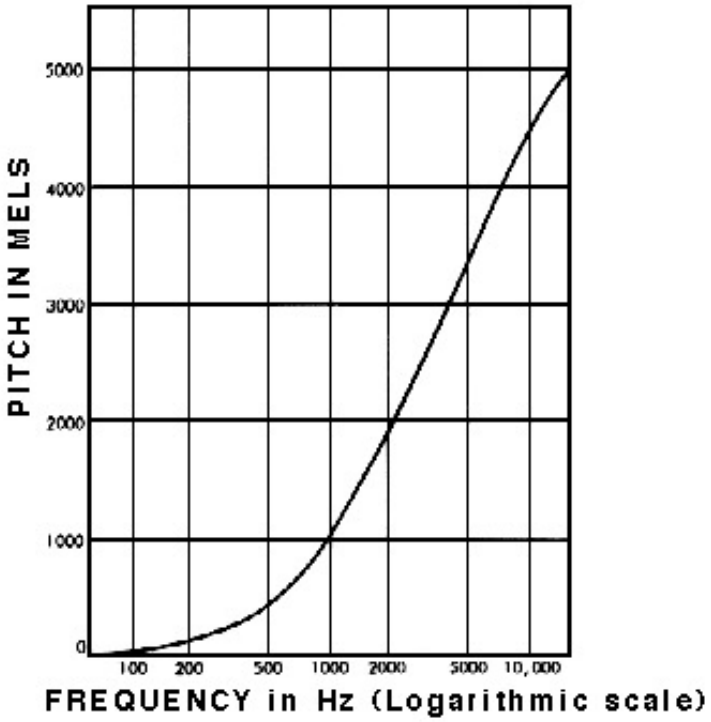
Mel-Spectrogram and MFCCs

https://www.youtube.com/watch?v=hF72sY70_IQ

https://www.youtube.com/watch?v=9GHCiiDLHQ4

Mel Scale

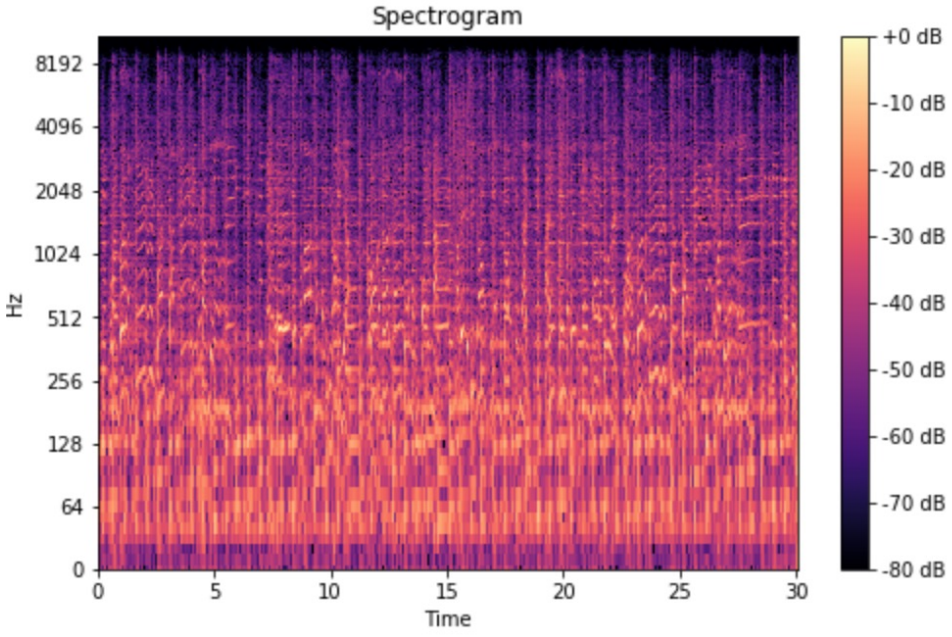
- Studies have shown that humans do not perceive frequencies on a linear scale. We are better at detecting differences in lower frequencies than higher frequencies.
- For example, we can easily tell the difference between 500 and 1000 Hz, but we will hardly be able to tell a difference between 10,000 and 10,500 Hz, even though the distance between the two pairs are the same.
- In 1937, Stevens, Volkmann, and Newmann proposed a unit of pitch such that equal distances in pitch sounded equally distant to the listener. This is called the **mel scale**.

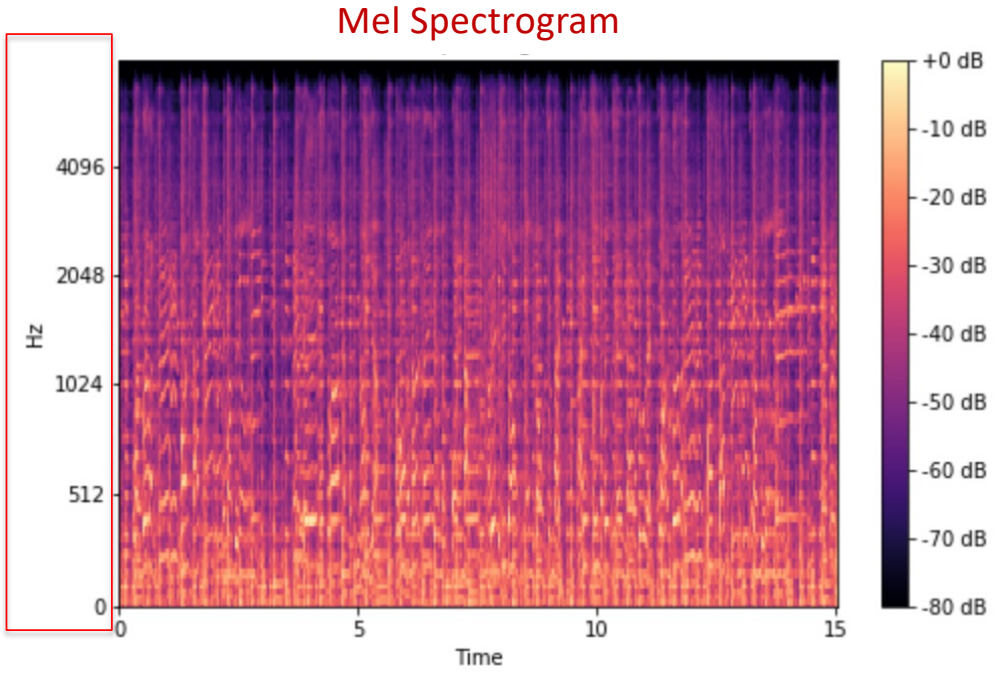


https://medium.com/analytics-vidhya/understanding-the-mel-spectrogram-fca2afa2ce53

Mel Spectrogram

- A **mel spectrogram** is a spectrogram where the frequencies are converted to the mel scale.



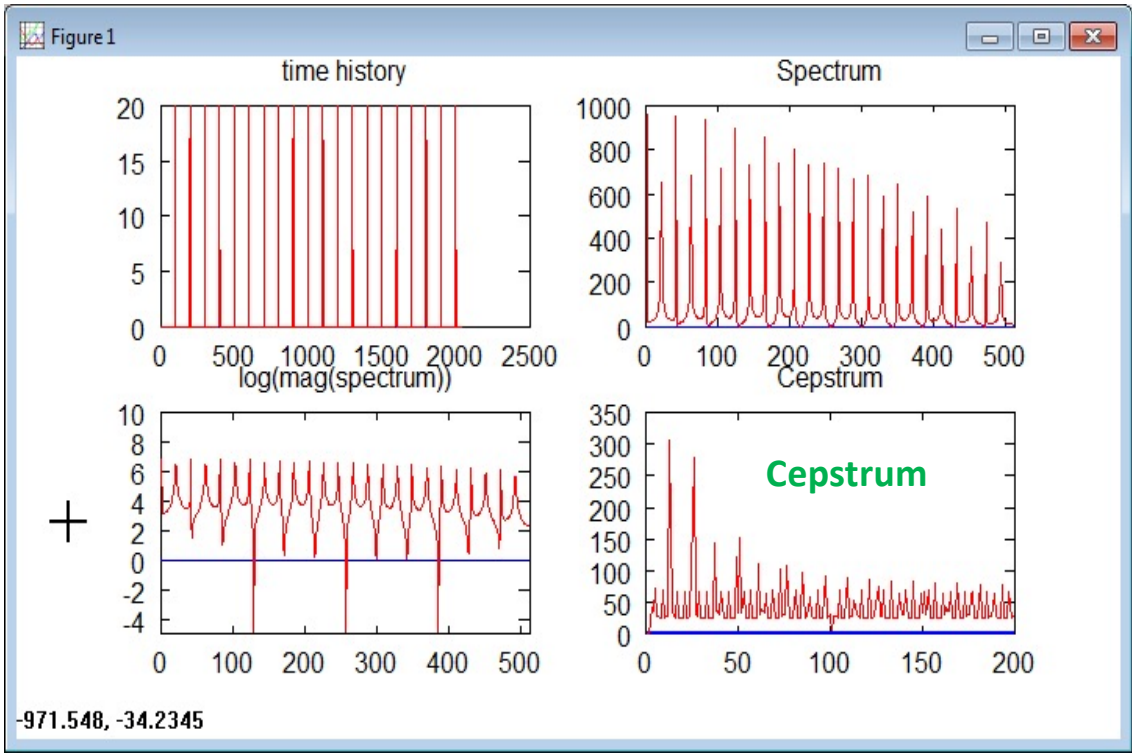


Mel Spectrogram Computation using librosa

```
import librosa
import librosa.display
import matplotlib.pyplot as plt

y, sr = librosa.load('./0eeaebcb_5.wav')

spect = np.abs(librosa.stft(y, hop_length=512))
spect = librosa.amplitude_to_db(spect, ref=np.max)

librosa.display.specshow(spect, sr=sr, x_axis='time', y_axis='log');
plt.colorbar(format='%+2.0f dB');
plt.title('Spectrogram');

mel_spect = librosa.feature.melspectrogram(y=y, sr=sr, n_fft=2048, hop_length=1024)
mel_spect = librosa.power_to_db(spect, ref=np.max)librosa.display.specshow(mel_spect,
y_axis='mel', fmax=8000, x_axis='time');
plt.title('Mel Spectrogram');
plt.colorbar(format='%+2.0f dB');
```

MFCC (Mel Frequency Cepstral Coefficients

- Ever heard the word *cepstral* before?
- It's *spec*tral with the *spec* reversed!
- For a basic understanding, cepstrum is the information of rate of change in spectral bands.



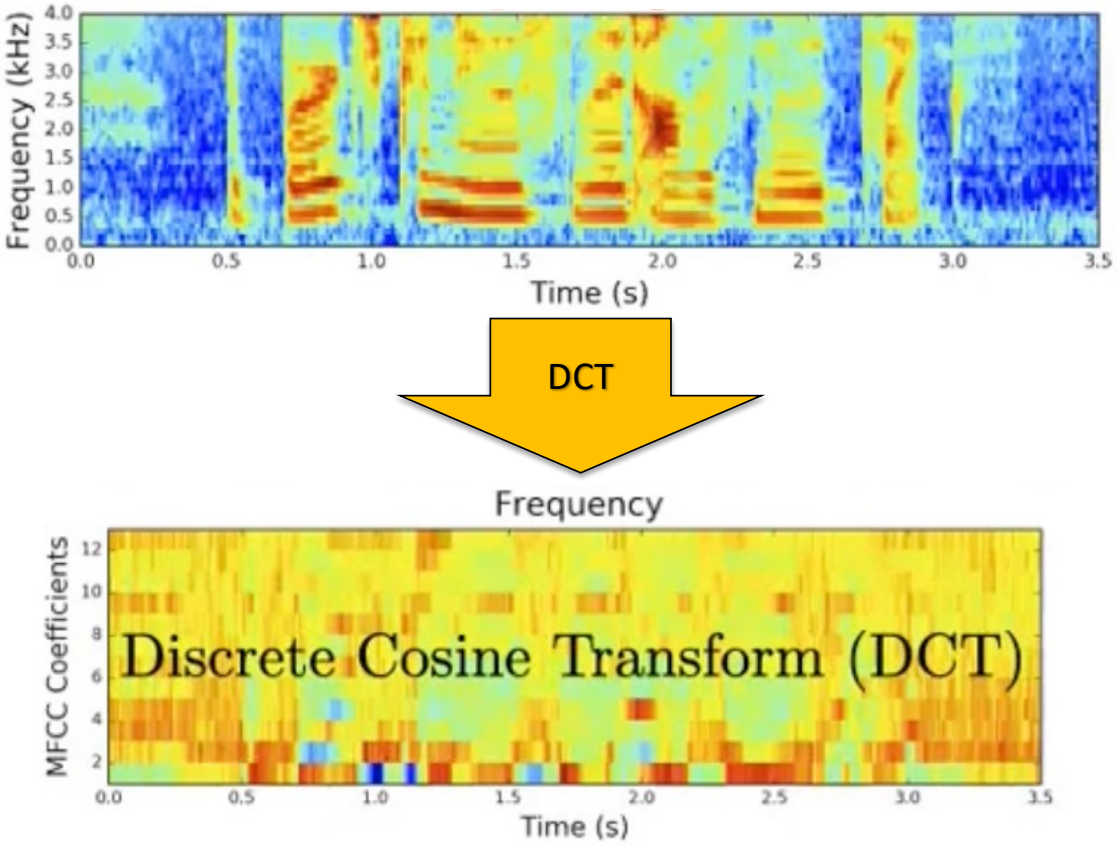
https://medium.com/prathena/the-dummys-guide-to-mfcc-aceab2450fd

MFCC

- On taking the log of the magnitude of this Fourier spectrum, and then again taking the spectrum of this log by a cosine transformation, we observe a peak wherever there is a periodic element in the original time signal.
- Since we apply a transform on the frequency spectrum itself, the resulting spectrum is neither in the frequency domain nor in the time domain and hence Bogert et al. decided to call it the *quefrency domain*.
- And this spectrum of the log of the spectrum of the time signal was named *cepstrum*.

MFCC

Mel Spectrogram

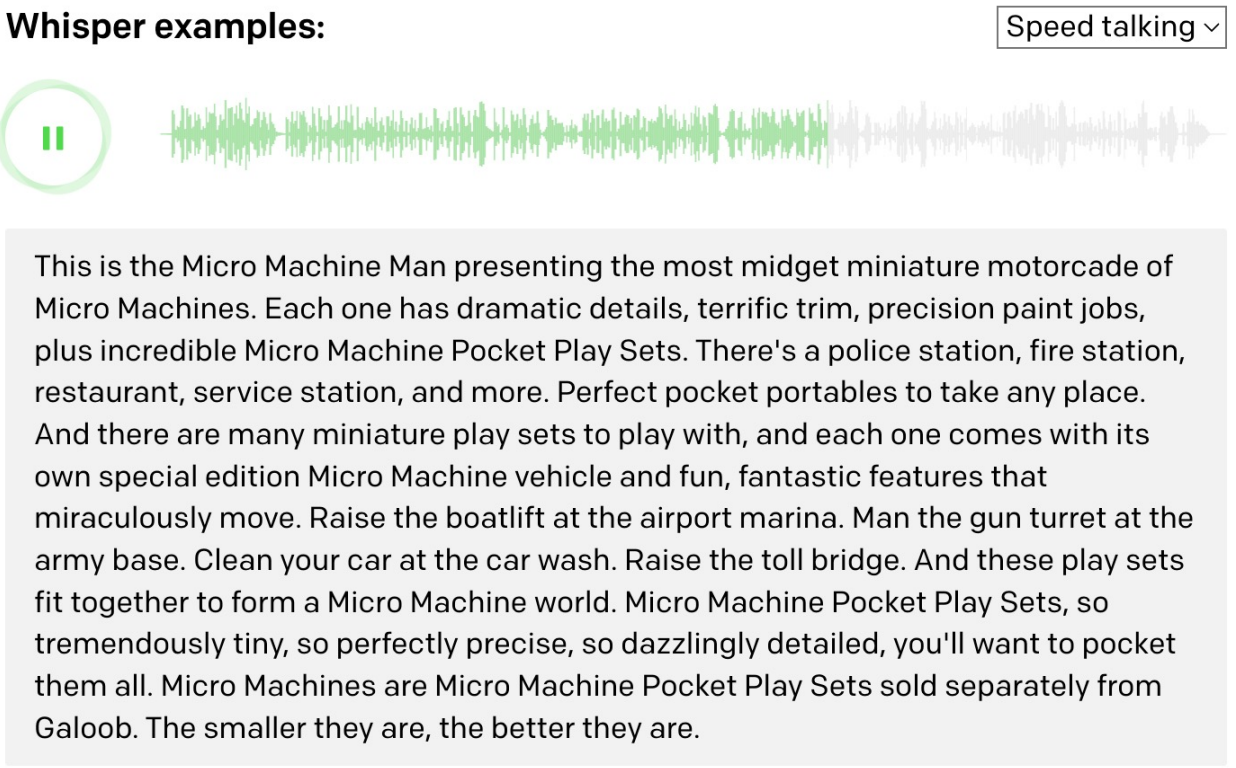


OpenAI Whisper
(Automatic Speech Recognition)

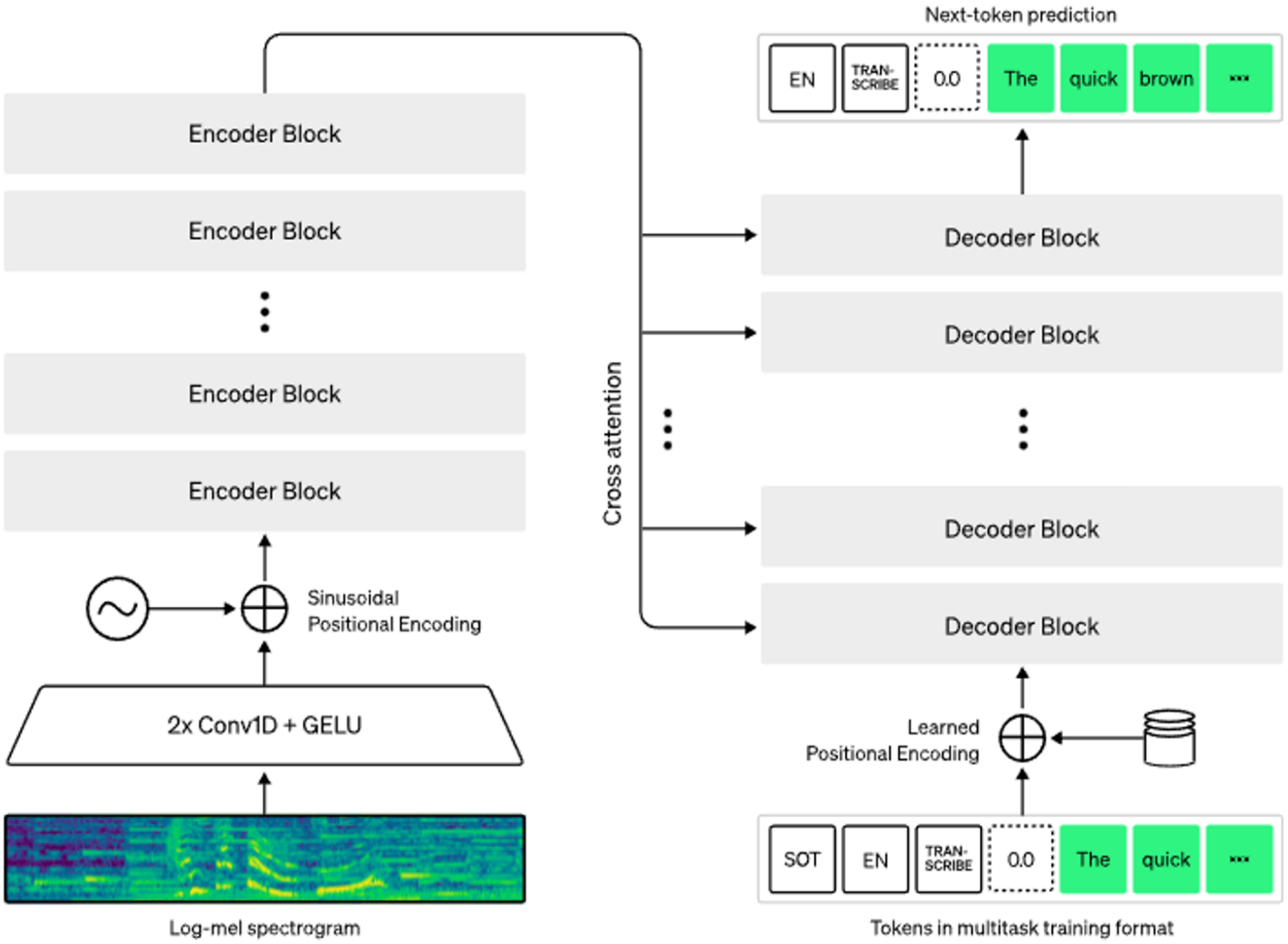
Whisper : ASR

- Whisper is an **automatic speech recognition** (ASR) system trained on 680,000 hours (77 years) of multilingual and multitask supervised data collected from the web
- It has added robustness to accents, background noise and technical sound analysis and modelling
- It has enabled transcription for multiple languages
- OpenAI has sourced it for further application building and development around it.

Whisper Examples



https://openai.com/blog/whisper/



https://openai.com/blog/whisper/

Whisper's Architecture

- It is implemented as encoder-decoder architecture
- It takes in 30-second chunks of audio input and converts into a log-Mel spectrogram
- Decoder predicts corresponding text captions
- Special tokens have been added which helps in performing further tasks like language identification, phrase-level timestamps, multilingual speech transcription, and to-English speech translation.
- One-third of whisper's dataset is non-English
  - It does a great job of transcribing audio in the original language or translating to English
  - It does a good job of learning speech-to-text tasks and outperforms supervised SOTA on CoVoST2 to English zero-shot.
  - Whisper performs better on zero-shots on different datasets because it is trained on fashion.
  - It's not fine-tuned on any specific dataset, so it doesn't beat the LibriSpeech benchmark, but it performs better when tested on different datasets.

Whisper Models: Tiny, Base, Small, Medium, Large

Size	Parameters	English-only model	Multilingual model	Required VRAM	Relative speed
tiny	39 M	`tiny.en`	`tiny`	~1 GB	~32x
base	74 M	`base.en`	`base`	~1 GB	~16x
small	244 M	`small.en`	`small`	~2 GB	~6x
medium	769 M	`medium.en`	`medium`	~5 GB	~2x
large	1550 M	N/A	`large`	~10 GB	1x

Hugging Face's Whisper Demo

- https://huggingface.co/spaces/openai/whisper