Applied Machine Learning

Course number: W207

Prof. Alexander I. Iliev, Ph.D.

Applied Machine Learning

Lecture 8 …

- *Project requirements*
- *SVM (whiteboard example)*
- *Comparison of ML algorithms discussed in class*
- *Fourier transform, DFT, FFT and IFFT, variations*
- *Audio, Speech, Phonemes, Formants, etc.*

Applied Machine Learning

- *Groups ready by next class on Monday – L9*

- *Project grading criteria:*

Late	Sensible methods (25)	Feature Engineering (20)	Error Analysis (15)	Notebook Presentation (10)	Problem Description (10)	Good results (10)	In Class Presentation (10)

Scipy Modules

- Here is a list of some of the most common Scipy modules:

  - Scipy.fftpack – contains Fast Fourier Transforms (FFTs)
  - Scipy.integrate – contains a variety of integrating functions
  - Scipy.interpolate – contains a variety of interpolation classes
  - Scipy.io – functions for reading and writing data from/to a variety of file formats
  - Scipy.io.wavfile – read/write data from/to a variety of file formats 'wav','arff', etc.
  - Scipy.linalg – contains linear algebra routines
  - Scipy.ndimage – contains many functions for multi-dimensional image processing
  - Scipy.signal – rich filtering capabilities, wavlets, spectral analysis, and much more
  - Scipy.optimize – local optimization package and root finding
  - Scipy.spatia – nearest neighbor queries and distance functions
  - Scipy.stats – large number of probability distributions and statistical functions
  - Scipy.special – large variety of functions such as: elliptic, bessel, legendre, etc.
  - Scipy.misc – variety of other functions

Scipy FFT

- The Scipy FFTpack
  - scipy.fftpack vs numpy.fft:
    - the scipy.fftpack does much more on top of what numpy.fft offers:
      - » fft and ifft - the Discrete Fourier Transform and its inverse of real or complex sequence of numbers
      - » fft2 and ifft2 - 2D discrete Fourier transform and its inverse
      - » fftn and ifftn - multidimensional discrete Fourier transform and its inverse
      - » dct and idct - Discrete Cosine Transform of arbitrary type sequence
      - » dst and idst - Discrete Sine Transform of arbitrary type sequence
      - » tilbert and itilbert - the h-Tilbert transform of a periodic sequence and its inverse
      - » hilbert and ihilbert - Hilbert Transform of a periodic sequence and its inverse
      - » fftfreq - the Discrete Fourier Transform sample frequencies
      - » convolve - performs convolution on a given signal
      - » … and more

Working with files

- Working with files – sound 1/4

reading .wav files

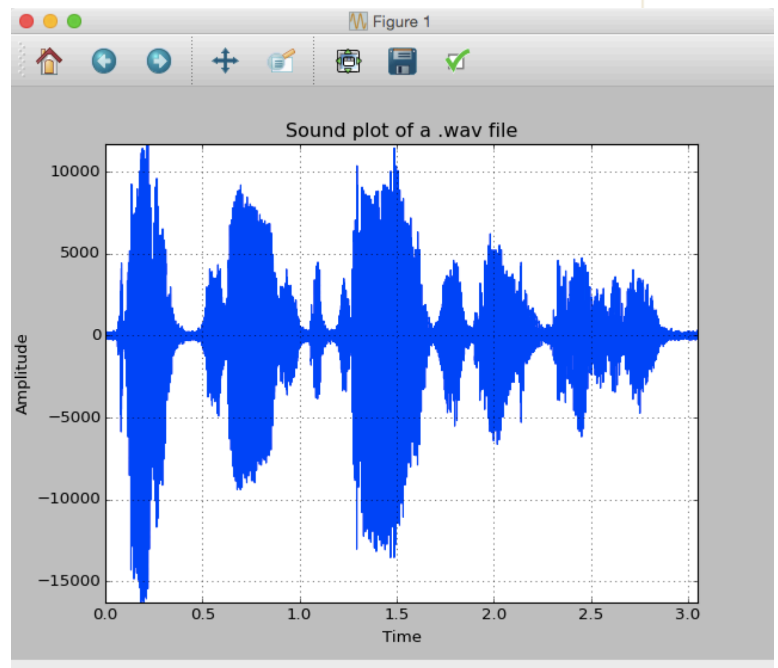
```
In [35]: from scipy.io.wavfile import read

In [36]: (fsx, x) = read('files/lecture8/alex_mono.wav')

In [37]: print(len(x.shape))   # '1' is mono
1

In [38]: print(x[:,])   # to access the channel no digit is used after ','
[-4 43 23 ..., 27 42 -3]

In [39]: x
Out[39]: array([-4, 43, 23, ..., 27, 42, -3], dtype=int16)

In [40]: (fsy, y) = read('files/lecture8/alex_stereo.wav')

In [41]: print(len(y.shape))   # '2' is stereo 2-dimensional array
2

In [42]: y
Out[42]:
array([[-4, -4],
       [44, 43],
       [20, 23],
       ...,
       [26, 29],
       [43, 41],
       [-4, -3]], dtype=int16)

In [43]: print(y[:,0])   # to access each channel separately use '0' or '1'
[-4 44 20 ..., 26 43 -4]

In [44]: print(y[:,1])
[-4 43 23 ..., 29 41 -3]
```

Working with files

- Working with files – sound 2/4

plotting .wav files

```
## More on sound:
# Example 1 - plotting sounds:
from pylab import linspace, plot, title, xlabel, ylabel, grid, axis, pause
from scipy.io.wavfile import read

(Fs, x) = read('files/lecture8/alex_mono.wav') # Fs - sampling frequency, x - signal
length = len(x)  # number of samples in 'x'
time = length/Fs # calculate the length of the .wav file in secs
t = linspace(0,time,length) # create evenly spaced numbers between [0:time]
plot(t,x) # plot signal 'x'
title('Sound plot of a .wav file')
xlabel('Time')
ylabel('Amplitude')
axis('tight')
grid(True)
```



Working with files

- Working with files – sound 3/4
  - lets create a (pseudo) stereo music from a single (mono) channel

manipulating .wav files

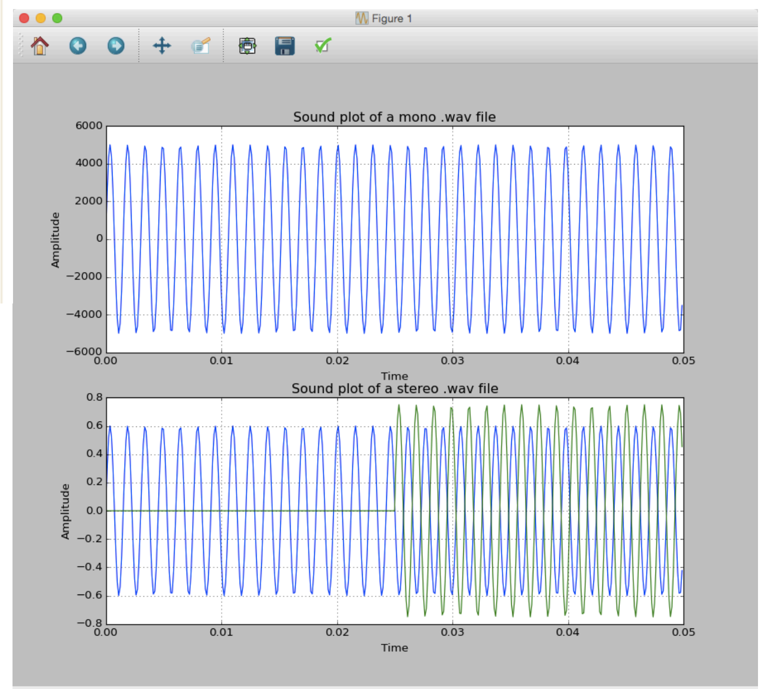
Note: this is not a true stereo Signal

```python
## Example 3 - manipulating sounds - creating pseudo-stereo from mono:
from numpy import zeros, concatenate
from scipy import fft, arange, ifft, sin, pi
from scipy.io.wavfile import read,write

(Fs, x) = read('files/lecture8/melody.wav')
x      # contains all the samples
y = x  # we create the second channel
z=zeros([200])  # create an array of zeros
# 1. Time/Phase shift the two channel:
L=concatenate((x,z)) # we add zeros after the 'x' signal to create Left channel
R=concatenate((z,y)) # we add zeros before the 'y' signal to create Right channel
A=zeros([len(L),2])  # we now create the array to store 'x' and 'y' as L and R
A[:,0]=L  # we assign the Left channel
A[:,1]=R  # we assign the Right channel
# 2. Amplitude change:
A[:,0]=A[:,0]*1.2e-4 # we decrease the amplitude on the Left to avoid clipping
A[:,1]=A[:,1]*1.5e-4 # ampl. decrease on Right channel is more since it is delayed

# we write the file:
write('files/lecture8/melody_stereo.wav',Fs,A)
```

Working with files

- Working with files – sound 4/4
  - we plot the mono and stereo music we just created

```python
## Lets plot the mono and pseudo-stereo sounds:
from pylab import linspace, plot, subplot, title, xlabel, ylabel, grid, pause

length = len(L)  # number of smaples in either channel 'L' (they are equal)
time = length/Fs # calculate the length of the .wav file in seconds
t = linspace(0,time,length) # create evenly spaced numbers between [0:time]

subplot(2,1,1)
plot(t[0:400],L[0:400]) # plot the first 400 samples from the mono signal 'L'
title('Sound plot of a mono .wav file')
xlabel('Time'); ylabel('Amplitude'); grid(True)

subplot(2,1,2)
plot(t[0:400],A[0:400]) # plot the first 400 samples from the stereo signal 'A'
title('Sound plot of a stereo .wav file')
xlabel('Time'); ylabel('Amplitude'); grid(True)
pause(1)
```



manipulating
.wav files

The Fast Fourier Transform

- The Fast Fourier Transform – quick intro
  - FFT is the faster implementation of the DFT
  - FFT is the basis for frequency analysis that converts any signal in time to frequency domain
  - fft is the Fast Fourier Transform (FFT) coverts time-domain signals to frequency-domain
  - ifft is the Inverse Fast Fourier Transform (IFFT) and coverts frequency to time-domain
  - the Fourier transform is represented like this (ex: an audio signal):

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i 2\pi k \frac{n}{N}} \quad \text{or} \quad \hat{f}(\xi) = \int_{-\infty}^{\infty} f(x)\, e^{-2\pi i x \xi}\, dx$$

where: [$x_0 , \ldots , x_{N-1}$] are complex conjugate numbers and [$k=0, \ldots, N-1$]

  - the most commonly used FFT is the Cooley–Tukey algorithm

The Fast Fourier Transform

- The Fast Fourier Transform – quick intro
  - FFT is the faster implementation of the DFT
  - FFT is the basis for frequency analysis that converts any signal in time to frequency domain
  - fft is the Fast Fourier Transform (FFT) coverts time-domain signals to frequency-domain
  - ifft is the Inverse Fast Fourier Transform (IFFT) and coverts frequency to time-domain
  - the Fourier transform is represented like this (ex: an image):

$$F(k,l) = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} f(i,j)\; e^{-\iota 2\pi \left(\frac{ki}{N} + \frac{lj}{N}\right)}$$

where: where *f(a,b)* is the image in the spatial domain and *F(k,l)* corresponds to each pixel

  - the most commonly used FFT is the Cooley–Tukey algorithm

The Fast Fourier Transform

- The Fast Fourier Transform

Example:

- notice that we only import what we need

- we add three simple tones to create one complex tone

- to go from time domain to frequency domain we use fft in two ways

```python
# FFT example:
from numpy.fft import rfft
from scipy import arange, sin, pi, fft, real ,imag, log10
from scipy.io.wavfile import write
from matplotlib.pyplot import figure, plot, subplot, axis, grid
from pylab import xticks, yticks, xlim, ylim, xlabel, ylabel, title

Fs=4000 # sampling frequency
a = arange(1024) # create a vector holding the number of bins
signal1 = sin(2*pi*a*(650/Fs))   # create a tone with frequency = 650Hz
signal2 = sin(2*pi*a*(1150/Fs))   # create a tone with frequency = 1.15kHz
signal3 = sin(2*pi*a*(1450/Fs))   # create a tone with frequency = 1.425kHz
signal4 = sin(2*pi*a*(1250/Fs))   # create a tone with frequency = 1.25kHz

# Create a complex tone:
signal = signal1 + signal2 + signal3

# Take the FFT of the complex signal:
freq_domain_npy = rfft(signal)  # using npy
freq_domain_cpy = fft(signal)   # using cpy
bin_val = Fs/len(a)    # calculate the value of each frequeny bin in [Hz]
bin_val # each bin in [Hz]
t = arange(1,Fs/2+2,bin_val)  # create a vector of frequency bins in [Hz]

# Save into a wav file:
write('files/lecture8/fft_file_example.wav',Fs,signal) # save to file
```

- each frequency bin represents frequency in [Hz] ->

Python

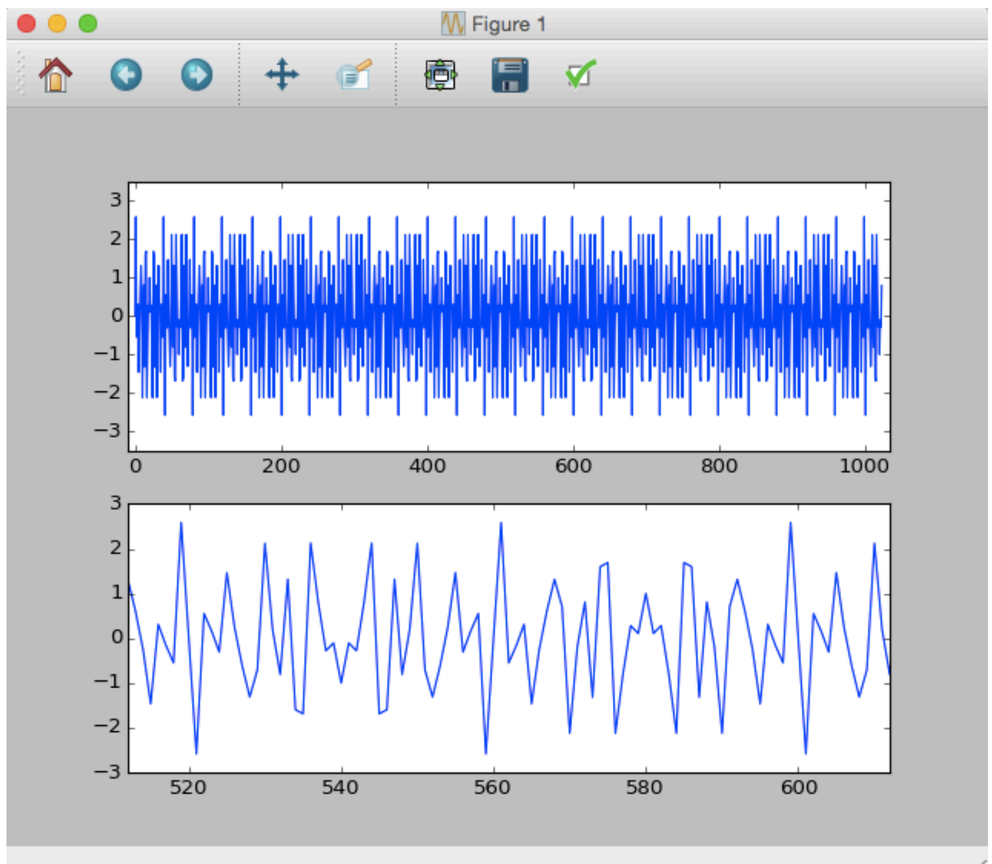
```
In [113]: bin_val # each bin in [Hz]
Out[113]: 3.90625
```

The Fast Fourier Transform

- The Fast Fourier Transform

Example:

```
# Plot the raw Time domain signal:
figure(1), subplot(2,1,1), plot(signal), xlim(-10, len(a)+10), ylim(-3.5,3.5)
subplot(2,1,2), plot(signal), xlim(len(a)/2,len(a)/2+100) # just a snipped of the signal
```

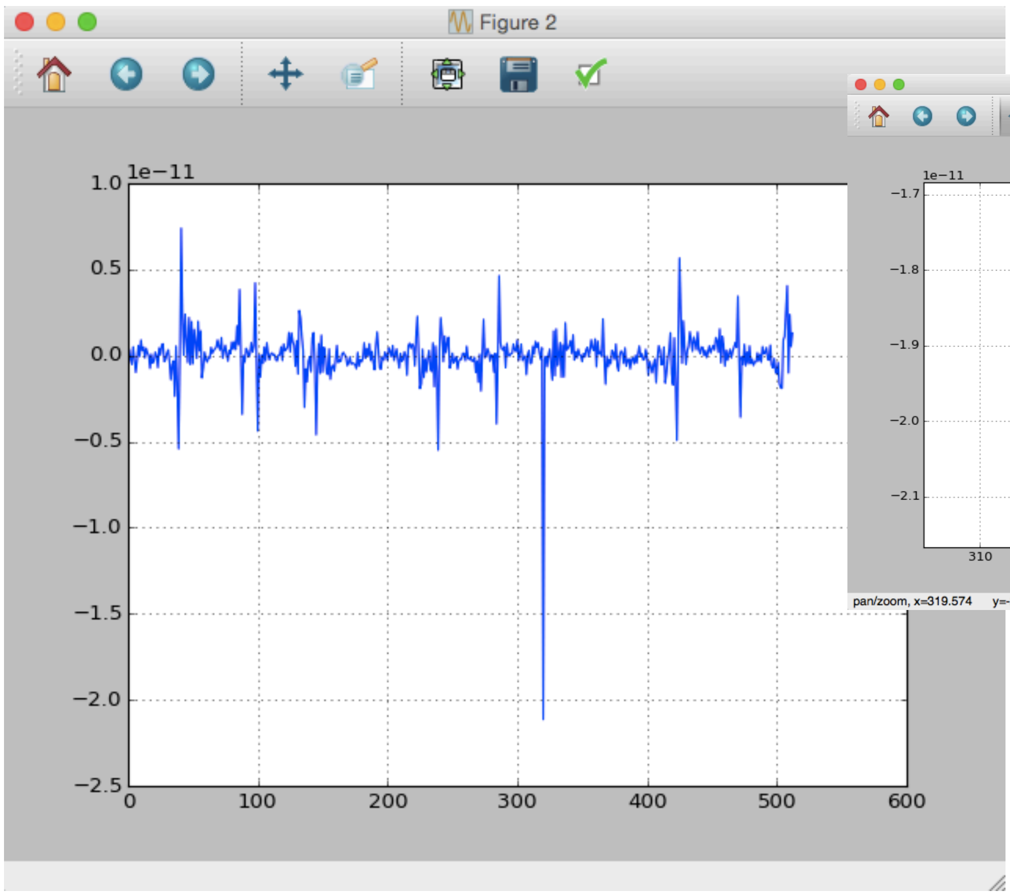


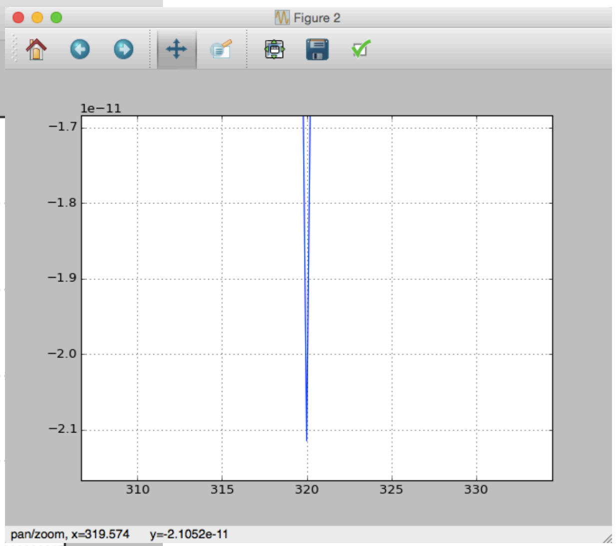
The Fast Fourier Transform

- The Fast Fourier Transform

Example:

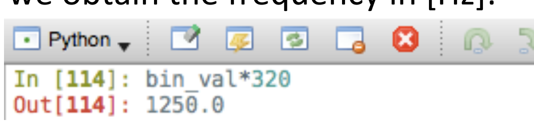
```
# Plot the Frequency domain of 'signal4':
figure(2), plot(rfft(signal4)) # observe the quantization noise due to rounding errors
grid(True)
```





- observe the quantization noise due to rounding errors

we obtain the frequency in [Hz]:



The Fast Fourier Transform

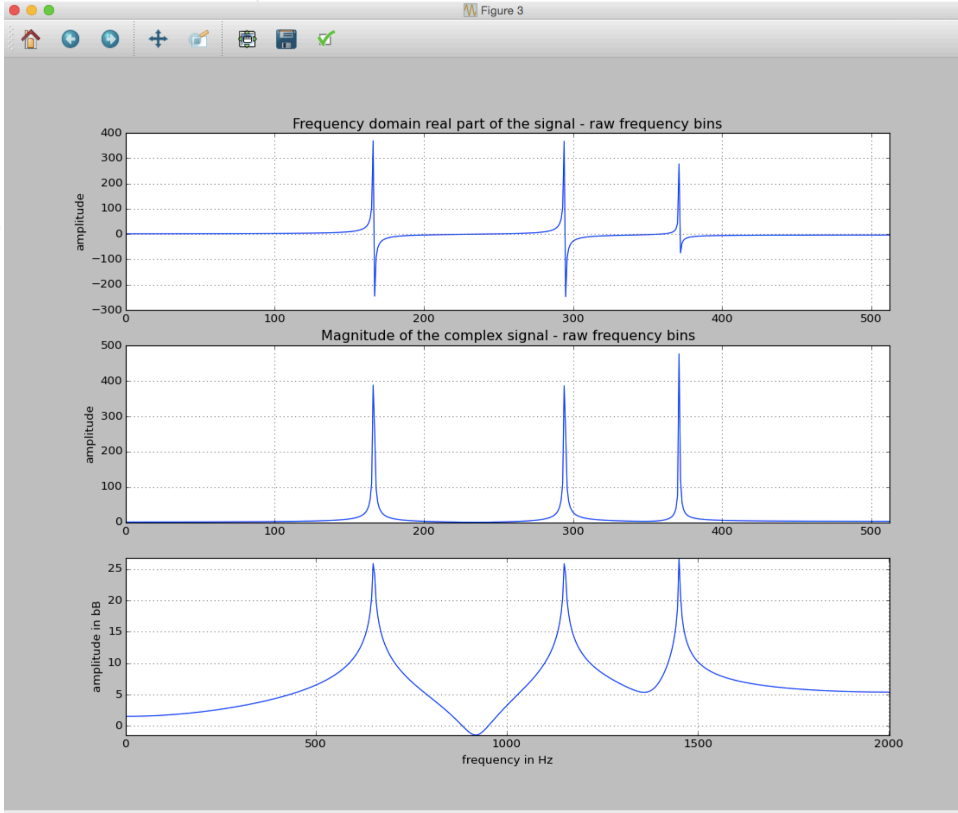
- The Fast Fourier Transform

Example:

```
# Plot the Frequency domain signal using numpy:
figure(3)
subplot(3,1,1),
title('Frequency domain real part of the signal - raw frequency bins'),
plot(freq_domain_npy),
xlim(0,len(a)/2),
ylabel('amplitude'),
grid(True)

# Plot the magnitude of the complex signal output:
subplot(3,1,2),
plot(abs(freq_domain_cpy)),
title('Magnitude of the complex signal - raw frequency bins'),
xlim(0,len(a)/2),
ylabel('amplitude'),
grid(True)

# Plot in dB scale:
subplot(3,1,3),
plot(t,10*log10(freq_domain_npy)),
xlabel('frequency in Hz'),
ylabel('amplitude in bB'),
axis('tight'),
grid(True)
```

```
In [11]: bin_val*166
Out[11]: 648.4375

In [12]: bin_val*294
Out[12]: 1148.4375

In [13]: bin_val*371
Out[13]: 1449.21875
```

```
reate a vec
*(650/Fs))
*(1150/Fs))
*(1450/Fs))
*(1250/Fs))
```



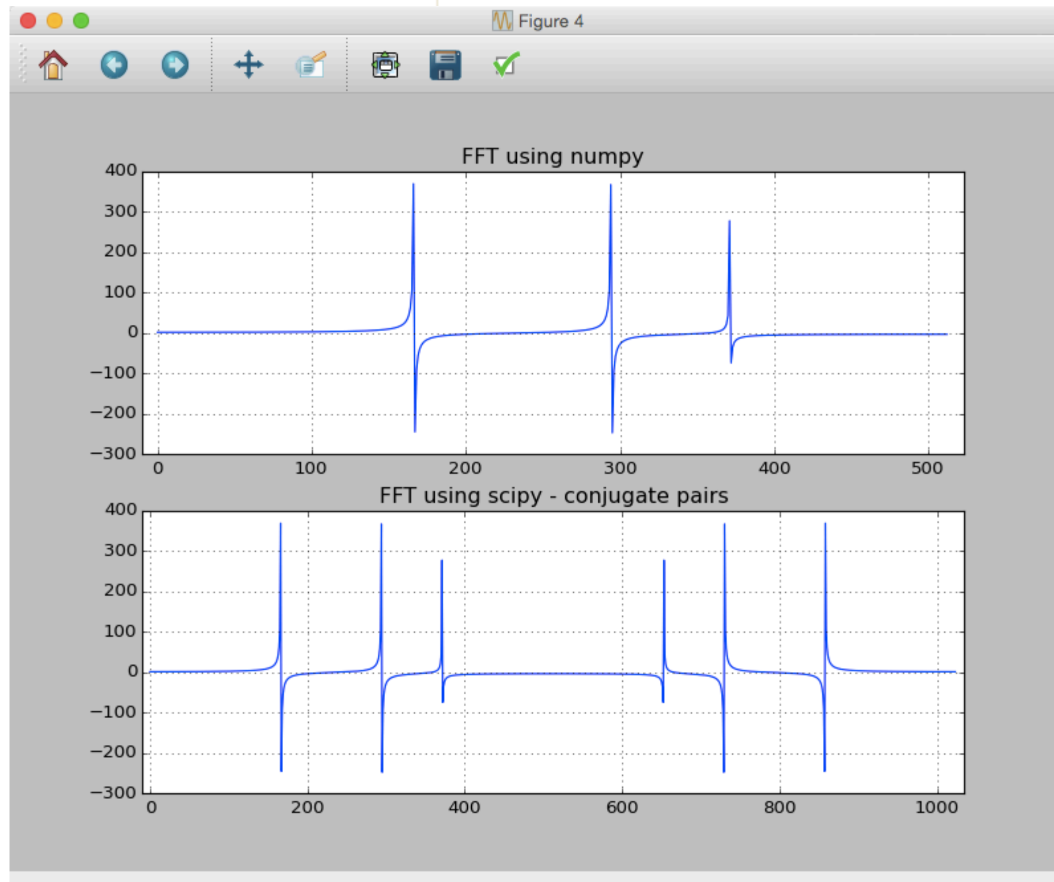
- notice the difference
in the x scales

The Fast Fourier Transform

- The Fast Fourier Transform

  Example:

```
# Plot the Frequency domain signal using numpy:
figure(4)
subplot(2,1,1),
plot(freq_domain_npy),
xlim(-10,len(t)+10),
title('FFT using numpy'),
grid(True)

# Plot the Frequency domain signal using scipy:
subplot(2,1,2),
plot(freq_domain_cpy),
xlim(-10,len(freq_domain_cpy)+10),
title('FFT using scipy - conjugate pairs'),
grid(True)
pause(1)
```



- notice the difference between:
  - rfft from NumPy and
  - fft from Scipy

Signal Processing

- Signal Processing – sound processing: spectrogram
  - spectrogram is a 3-D way of visualizing the frequency domain of any given signal
  - spectrograms are 3-D because they represent frequencies and their magnitudes over time in a given signal
  - signals are usually: sounds, music and speech, but can also be image signals
  - sometimes spectrograms are referred to as waterfalls, voiceprints, or voicegrams
  - they are the perfect tool for phonetic analysis in visualizing spoken words
  - spectrogram visualizing functionality can be uploaded in one of two ways:

    In [1]: **from pylab import specgram** … or

    In [1]: **from matplotlib.pyplot import specgram**
  - sometimes the spectrogram format varies and the vertical and horizontal axes can be switched
  - spectrograms are usually generated in two ways, by using:
    - FFT calculated from a given time signal
    - Filterbanks resulting from a sequence of bandpass filters

Signal Processing

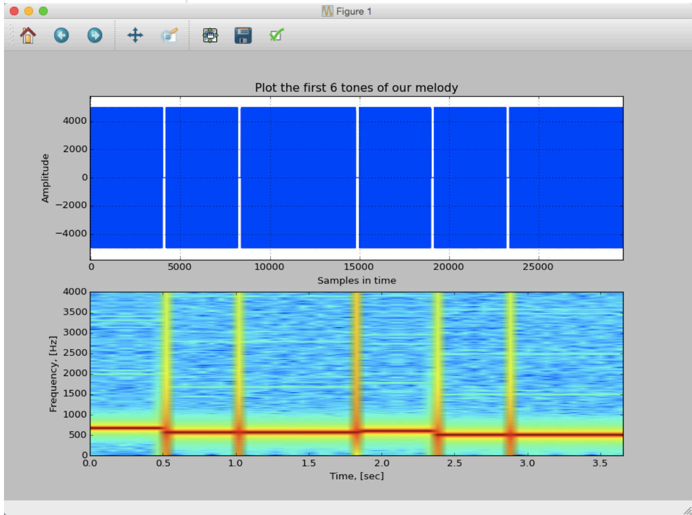
- Signal Processing – sound processing: spectrogram
  - here are some of the parameters users have control over:
    - Fs – the sampling frequency calculating the Fourier frequencies, in cycles per time
    - NFFT – the amount of frequency bins represented in each FFT window
    - window type used – Hamming, Hanning, Bratlett, Blackman, Kaiser
    - noverlap – amount of overlap between window blocks. the default overlap is 128 samples
    - mode – type of spectrogram to visualize: { 'psd' | 'magnitude' | 'angle' | 'phase’ }, where:
      - psd – is the power spectral density
      - magnitude – is the magnitude spectrum
      - angle – represents the phase spectrum without unwrapping
      - phase - is the phase spectrum with unwrapping
    - scale – how the data should be displayed {'default' | 'linear' | 'dB’ }, where:
      - default – linear
      - linear – means no scaling will be used
      - dB – when mode=psd’ the dB scale is (10*log10), otherwise it is (20*log10)
    - Fc – the center frequency can be controlled
    - cmap – is the colormap chosen

Signal Processing

- Signal Processing – sound processing: spectrogram

Example:

```python
# Signal Processing:
## 1. Spectrogram:

from pylab import specgram, plot, subplot, title, xlabel, ylabel, grid, axis, xlim, ylim, pause
from scipy.io.wavfile import read

(Fs, x) = read('files/lecture9/melody.wav') # Fs - sampling frequency, x - signal

# Lets plot signal 'x':
subplot(2,1,1)
plot(x)
xlim([-50,29750]); ylim([-5800,5800])    # limit it to the first 6 tones only
title('Plot the first 6 tones of our melody')
xlabel('Samples in time'); ylabel('Amplitude')
grid(True)

# Now we plot the spectrogram of the first 6 tones:
subplot(2,1,2)
xlabel('Time, [sec]'); ylabel('Frequency, [Hz]')
specgram(x[0:29750], NFFT=512, Fs=Fs, noverlap=64, mode='magnitude')
pause(1)
```



Signal Processing

- Signal Processing – sound processing: spectrogram

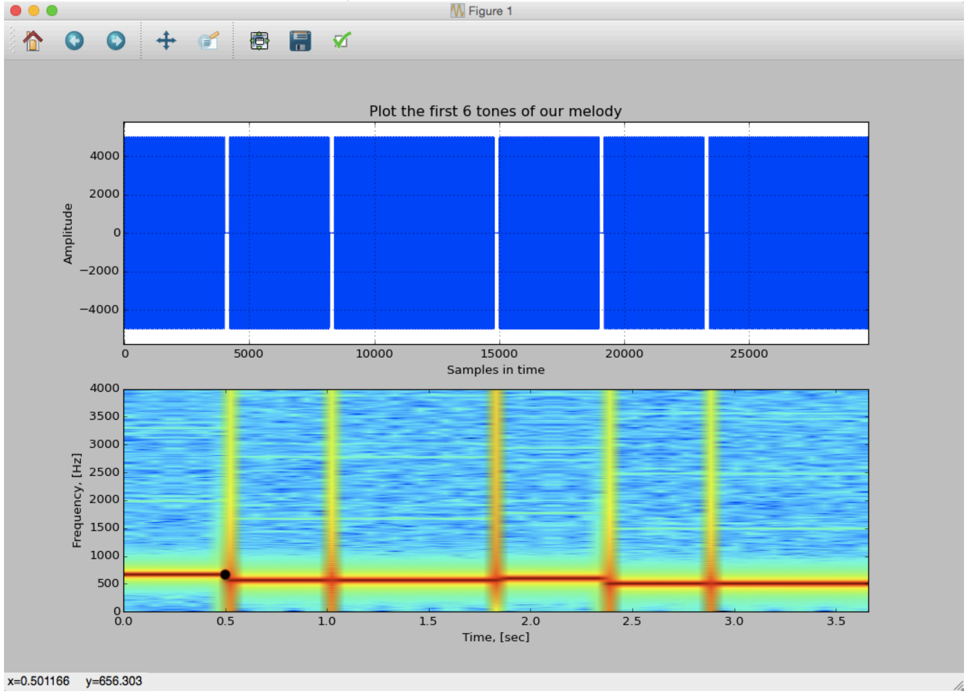
Example:

lets recall the code snipped from the melody:

```
# Creating each tone with Fs samples per second and length 0.5 or 0.8 seconds:
Es = note(659,Fs,0.5,amplitude=5000)
El = note(659,Fs,0.8,amplitude=5000)
Css = note(554,Fs,0.5,amplitude=5000)
Csl = note(554,Fs,0.8,amplitude=5000)
D = note(587,Fs,0.5,amplitude=5000)
Bs = note(494,Fs,0.5,amplitude=5000)
Bl = note(494,Fs,0.8,amplitude=5000)
As = note(440,Fs,0.5,amplitude=5000)
Al = note(440,Fs,0.8,amplitude=5000)
```

and the melody: E C# C# D B B A B C# D E E E
E C# C# D B B A C# E E A A A

– notice how the frequency tones are represented by the 'hottest' red line in the spectrum
– we also notice how the timing of each tone corresponds to 0.5 and 0.8 sec
– for E: x=0.501166 [sec], y=656.303 [Hz]



Signal Processing

- Signal Processing – sound processing: speech

Vowels and phonemes
in American – English

We use five letters to
represent the vowel sounds:
a, e, i, o, u

IPA fonts in Ladefoged (2006) and Roach (2009)

Words	Ladefoged (2006)	Roach (2009)	Words	Ladefoged (2006)	Roach (2009)
feet ★	/i/	/iː/	bird★	/ɜ/,/ɝ/	/ɜː/
hard★	/ɑ/	/ɑː/	bed★	/ɛ/	/e/
food★	/u/	/uː/	attend	/ə/	/ə/
lord★	/ɔ/	/ɔː/	book	/ʊ/	/ʊ/
hot★	/ɑ/(GA), /ɒ/ (RP)	/ɒ/	go★	/oʊ/(GA) /əʊ/(RP),	/əʊ/
bus,	/ʌ/	/ʌ/	boy	/ɔɪ/	/ɔɪ/
book	/ʊ/	/ʊ/	big	/ɪ/	/ɪ/
dear,	/ɪə/	/ɪə/	care★	/ɛə/	/eə/
bike	/aɪ/	/aɪ/	how	/aʊ/	/aʊ/
cake	/eɪ/	/eɪ/	tour	/ʊə/	/ʊə/

★ =There is a difference between two systems

Signal Processing

- Signal Processing – sound processing: speech

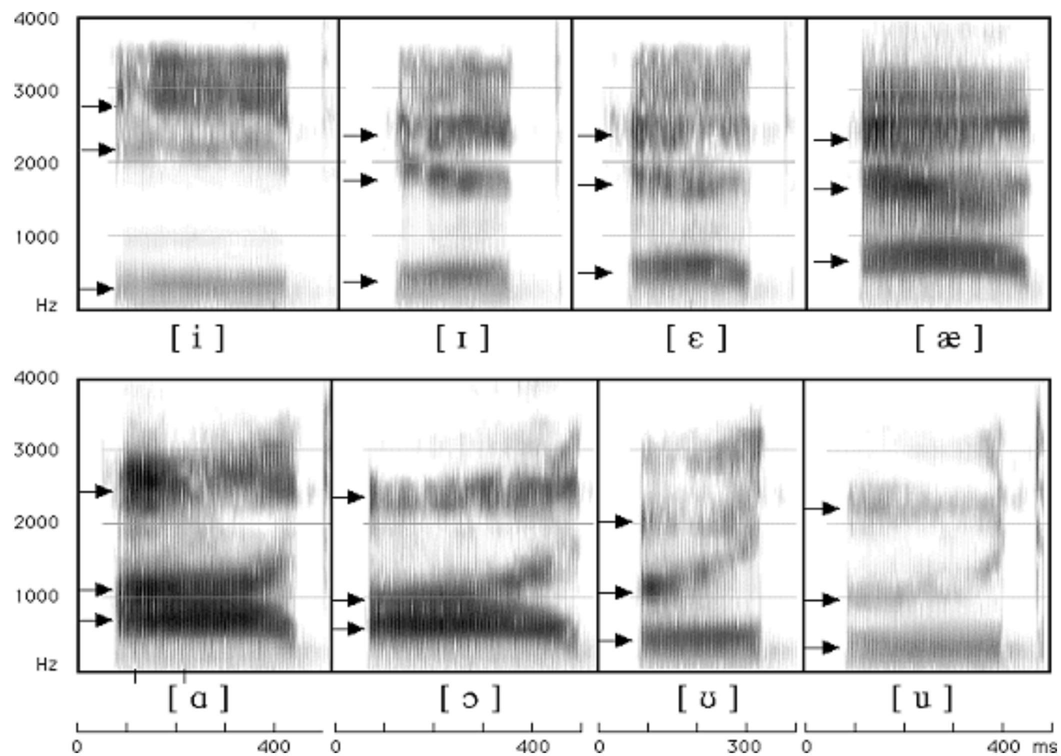
Formants are resonances in the vocal tract

Each vowel is formed by formants: a concentration of acoustic energy around a specific frequency

Each formant has a different center frequency with higher amplitude

Formants for different genders or kids vary

Spectrograms of the American English Vowels



(Ladeforged 2006:185-187)

Signal Processing

- Signal Processing – sound processing: speech

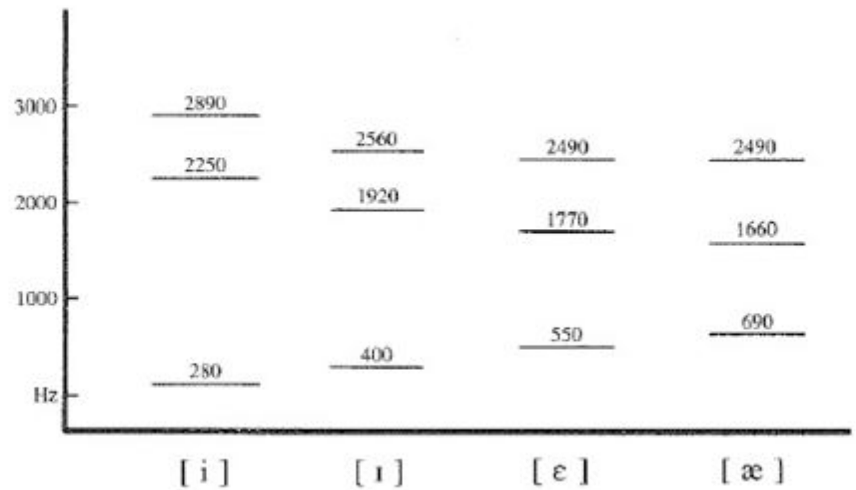
Formants are resonances in the vocal tract

Each vowel is formed by formants: a concentration of acoustic energy around a specific frequency

Each formant has a different center frequency with higher amplitude

Formants for different genders or kids vary

Spectrograms of the American English Vowels



(Ladefoged & Johnson, 2011:193)

Signal Processing

- Signal Processing – sound processing: speech

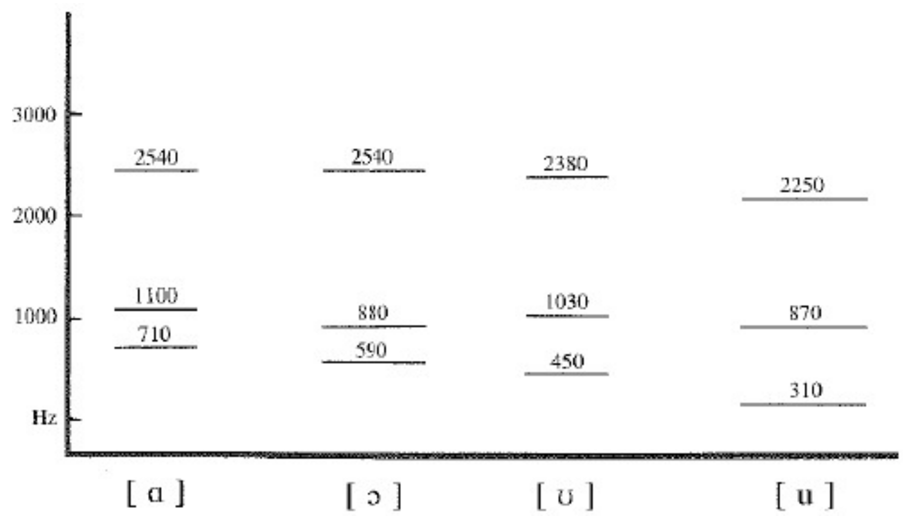
Formants are resonances in the vocal tract

Each vowel is formed by formants: a concentration of acoustic energy around a specific frequency

Each formant has a different center frequency with higher amplitude

Formants for different genders or kids vary

Spectrograms of the American English Vowels



(Ladefoged & Johnson, 2011:193)

Signal Processing

- Signal Processing – sound processing: speech

We use five letters to represent the vowel sounds: a, e, i, o, u

use LPC for formant estimation from Audiolazy:

pip install audiolazy

Formant frequencies for each vowel

Vowel	F1(Hz)	F2(Hz)	F3(Hz)
iː	280	2620	3380
ɪ	360	2220	2960
e	600	2060	2840
æ	800	1760	2500
ʌ	760	1320	2500
ɑː	740	1180	2640
ɒ	560	920	2560
ɔː	480	760	2620
ʊ	380	940	2300
uː	320	920	2200
ɜː	560	1480	2520

Adult male formant frequencies in Hertz collected by J.C.Wells around 1960.
Note how F1 and F2 vary more than F3.

Signal Processing

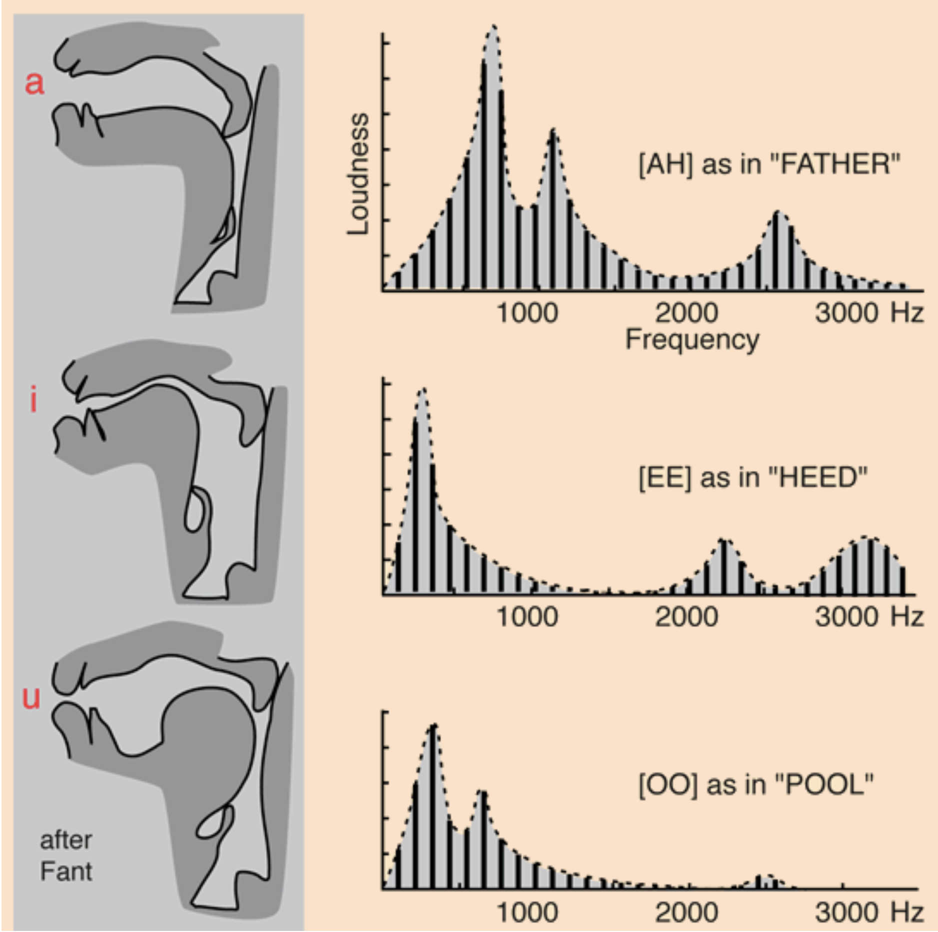
- Signal Processing – sound processing: speech

See levels for each formant

Each vowel is formed by formants

Each formant has a different frequency and amplitude

Formants for different genders or kids vary



Feature Extraction – glottal signal

The Glottal Signal:

conveys speaker identity, mode of speaking

airflow passing through the glottis creates voiced sounds: *vowels, semivowels, nasals, diphthongs, consonants*

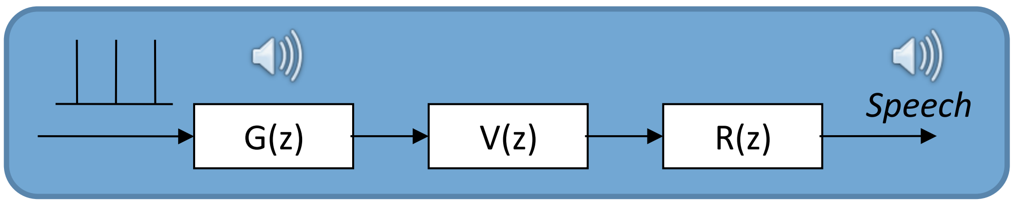
Glottal waveform is greatly affected by the emotional state - *Laukkanen et al., 1996*

MRI - collected from **Centre for Speech Technology** and I'Institute da la Communication Parlee in Grenoble

Feature Extraction – glottal signal

Speech production model:

3 concatenated linear time-varying subsystems



$$S(z) = G(z)V(z)R(z)$$

$$G(z) = \frac{S(z)}{V(z)R(z)}$$