UC Berkeley Extension

Python for Data Analysis and Scientific Computing

X433.3 (2 semester units in COMPSCI)

Instructor Alexander I. Iliev, Ph.D.

Course Content Outline

- **NumPy 2/3**
- Array operations
- Reductions
- Broadcasting
- Array: shaping, reshaping, flattening, resizing, dimension changing
- Data sorting — Midterm / Project proposal due

- **NumPy 3/3**
- Type casting
- Masking data
- Organizing arrays
- Loading data files
- Dealing with polynomials
- Good coding practices

- **Scipy 1/2**
- What is Scipy?
- Working with files
- Algebraic operations
- The Fast Fourier Transform
- Signal Processing — HW4

- **Scipy 2/2**
- Interpolation
- Statistics
- Optimization

- **Project**
- Project Presentation — Final Project

Working with files

- Working with files – playing sounds:

PyAudio: for playing and recording sounds with Python

Installation steps for PyAudio on Mac OS X:

Step 1: execute this line in your terminal:

```
ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Step 2: execute this line in your terminal:

```
brew install portaudio
```

Step 3: execute this line in your Python setup (PyZo):

```
pip install pyaudio
```

Step 4: write your routine to play the audio and save it as a .py module

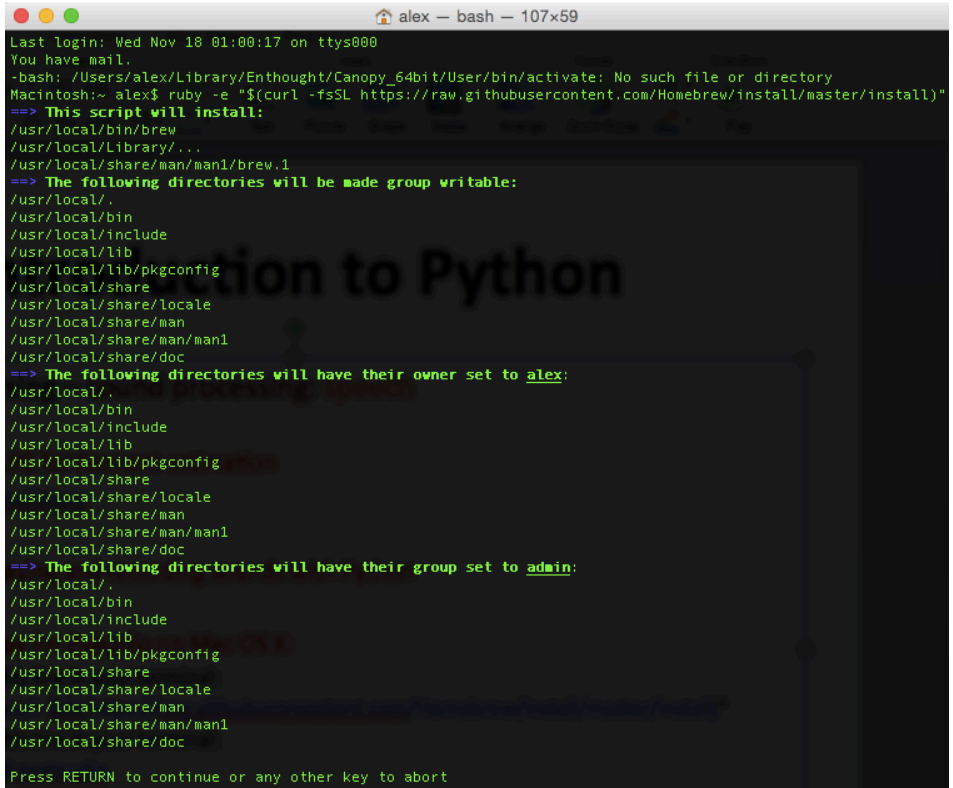
Working with files

- Working with files – playing sounds:

Installation steps for PyAudio on Mac OS X:

Step 1: execute this line in your terminal:

ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"



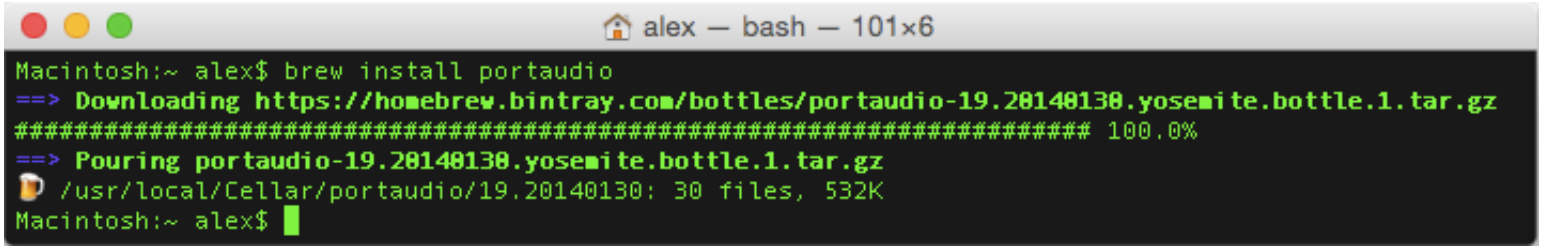
Working with files

- Working with files – playing sounds:

Installation steps for PyAudio on Mac OS X:

Step 2: execute this line in your terminal:

brew install portaudio



Step 3: execute this line in your Python setup (Pyzo):

pip install pyaudio

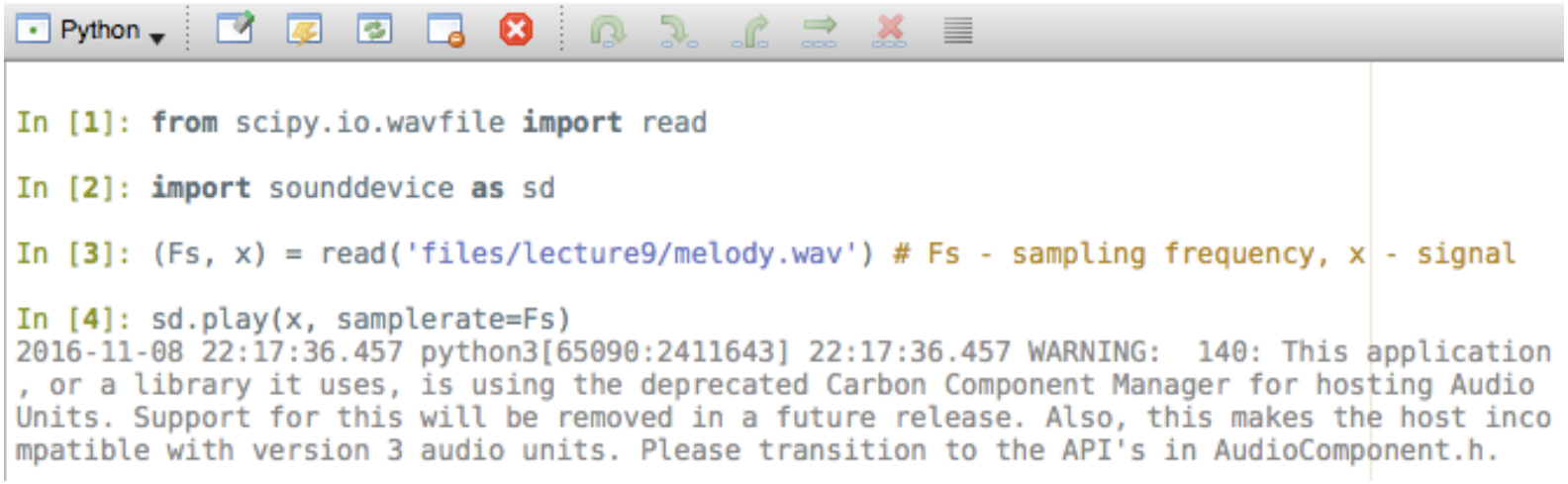
Step 4: write your routine to play the audio and save it as a .py module

More info for other systems: http://people.csail.mit.edu/hubert/pyaudio/#binaries

Working with files

- Working with files – playing sounds with sounddevice:

  sounddevice:
  - is much easier to install and use as there is no need for you to create your play module
  - … however depending on your setup it might give you a warning message of depricated components so it may not run



Working with files

- Working with files – recording sounds with sounddevice:

sounddevice:

```
178  ## Working with files – recording sounds with sounddevice 1/2:
179  from scipy.io.wavfile import read
180  import sounddevice as sd
181
182  # Set the sampling frequency:
183  Fs = 8000 # kHz
184
185  # Duration of our recoring in 'sec' will be:
186  duration = 3
187
188  # To record audio data from your sound device into a NumPy array, use:
189  recorded_audio_1 = sd.rec(int(duration * Fs), samplerate=Fs, channels=2, blocking=True)
190
191  # We can now listen to our recording:
192  sd.play(recorded_audio_1, Fs)
```

Note: make sure you use 'blocking=True', to make sure recording has finished

Working with files

- Working with files – recording sounds with sounddevice:

sounddevice:

```
193  ## Working with files – recording sounds with sounddevice 2/2:
194  from scipy.io.wavfile import read
195  import sounddevice as sd
196
197  # Set the sampling frequency:
198  Fs = 8000 # kHz
199
200  # Duration of our recoring in 'sec' will be:
201  duration = 3
202
203  # Let set our default parameters:
204  sd.default.samplerate = Fs
205  sd.default.channels = 2     # for stereo signal
206
207  # By default, the recorded array has the data type 'float32', but we can change that:
208  recorded_audio_2 = sd.rec(duration * Fs, dtype='float64', blocking=True)
209
210  # We can now listen to our recording:
211  sd.play(recorded_audio_2, Fs)
```

Algebraic operations

- Algebraic operations
  - standard linear algebra operations in Scipy are provided through the linalg module
  - there is a rich list of prebuilt functions
  - the input and output of all linear algebra routines are 2-dimensional array objects
  - scipy.linalg contains more advanced functions on top of all the functions available in numpy.linalg
  - scipy.linalg is always compiled with the ATLAS LAPACK and BLAS libraries, unlike numpy.linalg, hence it has very fast linear algebra capabilities
  - scipy.linalg is therefore always better to use than numpy.linalg

Algebraic operations

- Algebraic operations
  - Difference between a numpy.matrix and 2D numpy.ndarray
    - basic operations such as multiplications and transpose are included in NumPy for both matrix and ndarray types
    - numpy.matrix has a proper interface for matrix operations unlike numpy.ndarray
    - however, the numpy.matrix class does not add anything that cannot be achieved by using 2D numpy.ndarray objects
    - the implementation of this class resembles the one in Matlab
    - scipy.linalg operations can be used just as good to 2D numpy.ndarray objects as well as to numpy.matrix

Algebraic operations

- Algebraic operations
  - Difference between a numpy.matrix and 2D numpy.ndarray

the I and T class members serve as shortcuts for inverse and transpose respectively

the numpy.matrix class does not add anything that cannot be achieved by using 2D numpy.ndarray objects

```
Python
In [45]: # Matrix vs Array:

In [46]: # 1. Matrix:

In [47]: import numpy as npy

In [48]: a = npy.mat('[12 34 41;52 64 72]') # create matrix 'a'

In [49]: a
Out[49]:
matrix([[12, 34, 41],
        [52, 64, 72]])

In [50]: type(a)
Out[50]: numpy.matrixlib.defmatrix.matrix

In [51]: a.I # inverse of matrix 'a'
Out[51]:
matrix([[-0.05885099,  0.03258602],
        [ 0.01490374, -0.00181295],
        [ 0.02925572, -0.00803395]])

In [52]: b = npy.mat('[84 92]') # create matrix 'b'

In [53]: b
Out[53]: matrix([[84, 92]])

In [54]: type(b)
Out[54]: numpy.matrixlib.defmatrix.matrix

In [55]: b.T # transpose of matrix 'b'
Out[55]:
matrix([[84],
        [92]])

In [56]: a.T*b.T # multiplication of two matrices
Out[56]:
matrix([[ 5792],
        [ 8744],
        [10068]])
```

Operations

- Operations
  - Difference between a numpy.matrix and 2D numpy.ndarray

scipy.linalg operations can be used just as good to 2D numpy.ndarray objects as well as to numpy.matrix

Note:
npy.mat and npy.matrix
Have the same meaning, but are different objects.
Try, using 'id' to see the difference

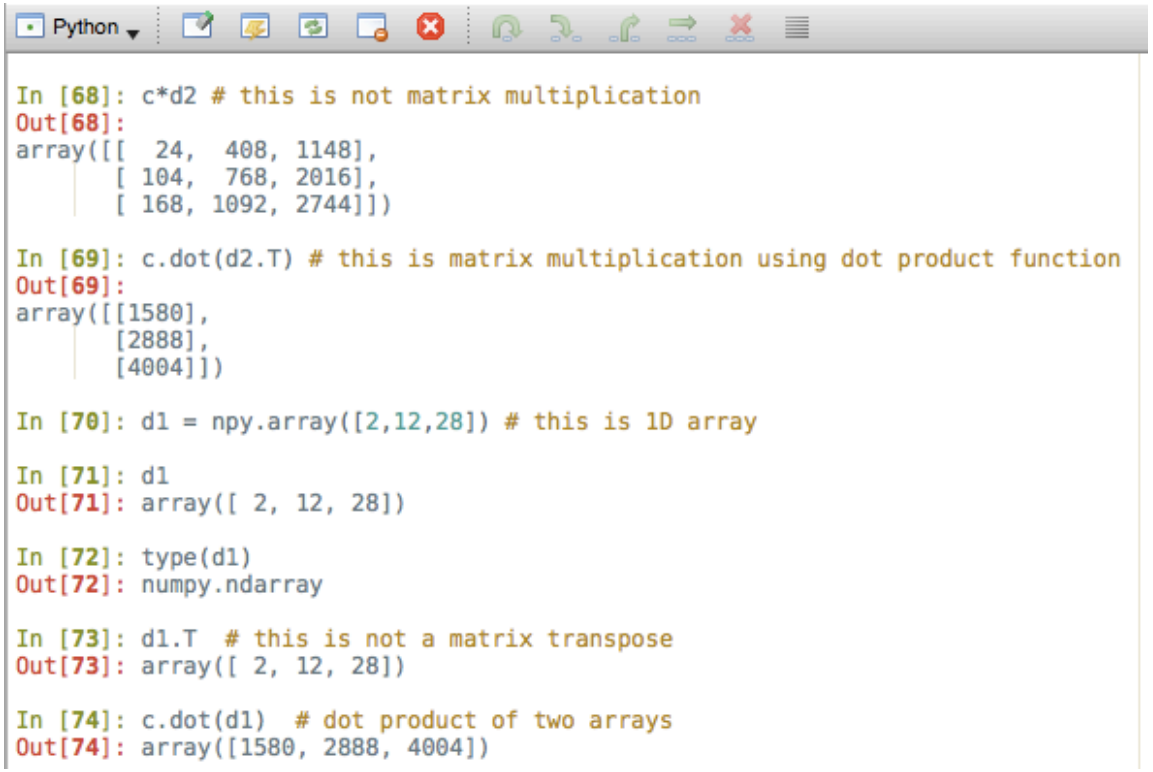
Python

```
In [57]: # Matrix vs Array:

In [58]: # 2. Array:

In [59]: import numpy as npy

In [60]: from scipy import linalg

In [61]: c = npy.array([[12,34,41],[52,64,72],[84,91,98]]) # create array 'c'

In [62]: c
Out[62]:
array([[12, 34, 41],
       [52, 64, 72],
       [84, 91, 98]])

In [63]: type(c)
Out[63]: numpy.ndarray

In [64]: linalg.inv(c)   # calculate the inverse of a matrix
Out[64]:
array([[-0.10752688,  0.15322581, -0.06758833],
       [ 0.3655914 , -0.87096774,  0.48694316],
       [-0.24731183,  0.67741935, -0.38402458]])

In [65]: d2 = npy.array([[2,12,28]]) # create 2D array

In [66]: d2
Out[66]: array([[ 2, 12, 28]])

In [67]: type(d2)
Out[67]: numpy.ndarray
```

Operations

- Operations
  - Difference between a numpy.matrix and 2D numpy.ndarray

scipy.linalg operations can be used just as good to 2D numpy.ndarray objects as well as to numpy.matrix



Operations

- Operations
  - Finding the inverse of an array (matrix)

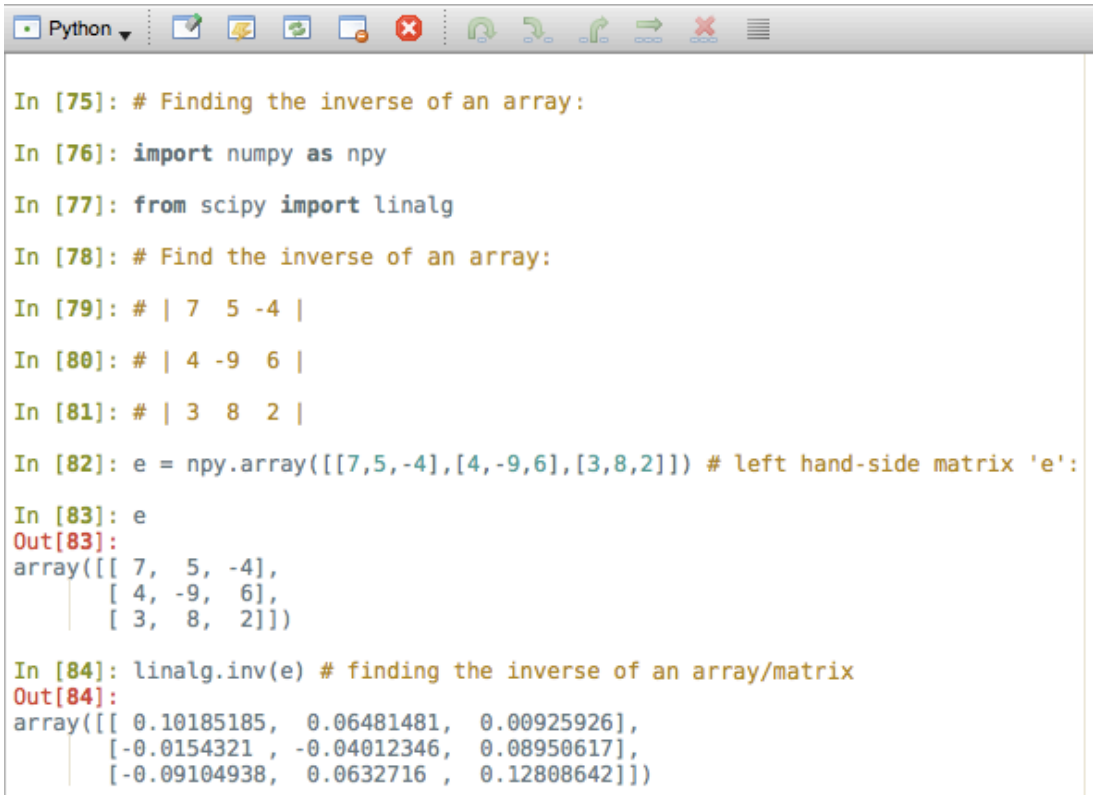
matrix ‘e’ has its inverse matrix ‘f’ such that e*f=I -> I is the identity matrix that has main diagonal with ones

we can then say that: f=e−1

the matrix inverse of the NumPy array ‘e’ is obtained in two ways:

a) using the Scipy linalg.inv, or

b) using e.I when ‘e’ is a matrix so cast it like this: npy.mat(e).I



Linear Systems

- Linear systems
  - Solving linear systems
    - to compute the solution vector of a linear system with any number of equations and unknowns, all we have to do is construct the right hand-side vector and left hand-side matrix with coefficients
    - lets consider the system:

$$7x + 5y - 4z = 15$$
$$4x - 9y + 6z = 12$$
$$3x + 8y + 2z = 10$$

    - the solution vector can be found by using the matrix inverse:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 7 & 5 & -4 \\ 4 & -9 & 6 \\ 3 & 8 & 2 \end{bmatrix}^{-1} \begin{bmatrix} 15 \\ 12 \\ 10 \end{bmatrix} = \begin{bmatrix} 2.39 \\ 0.18 \\ 0.67 \end{bmatrix}$$

    - now lets find the solution using Python
    - we will use scipy.linalg as it is faster than numpy.ndarray

Linear Systems

- Linear systems
  - Solving linear systems

dot product:

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{n} a_i b_i$$

we will use linalg.solve as it is faster than inv.dot

```
Python
In [85]: # Solving linear systems:

In [86]: import numpy as npy

In [87]: from scipy import linalg

In [88]: # Solving the linear system:

In [89]: # 7x + 5y - 4z = 15

In [90]: # 4x - 9y + 6z = 12

In [91]: # 3x + 8y + 2z = 10

In [92]: e = npy.array([[7,5,-4],[4,-9,6],[3,8,2]]) # left hand-side matrix 'e':

In [93]: e
Out[93]:
array([[ 7,  5, -4],
       [ 4, -9,  6],
       [ 3,  8,  2]])

In [94]: f = npy.array([[15],[12],[10]]) # we create the right hand-side vector 'f'

In [95]: f
Out[95]:
array([[15],
       [12],
       [10]])

In [96]: linalg.inv(e).dot(f) # we solve the system using 'linalg.inv'
Out[96]:
array([[ 2.39814815],
       [ 0.18209877],
       [ 0.67438272]])

In [97]: %timeit linalg.inv(e).dot(f) # we calculate how fast it is (it is slower)
10000 loops, best of 3: 72 µs per loop

In [98]: npy.linalg.solve(e,f) # we solve the system using 'linalg.solve'
Out[98]:
array([[ 2.39814815],
       [ 0.18209877],
       [ 0.67438272]])

In [99]: %timeit npy.linalg.solve(e,f) # we calculate how fast it is (it is faster)
10000 loops, best of 3: 38.5 µs per loop
```

Singular Value Decomposition

- SVD
  - Singular Value Decomposition svd
  - It is a great tool for dimensionality reduction

Can be used in:
- Image compression
- Computer vision (using PCA - SVD)
- Signal processing
- Fitting solutions
- Statistics

```
Python
In [100]: # More advanced features:

In [101]: # Using svd to obtain a Singular Value Decomposition:

In [102]: e
Out[102]:
array([[ 7,  5, -4],
       [ 4, -9,  6],
       [ 3,  8,  2]])

In [103]: a = e + npy.diag([1, 1, 1]) # an array of singular real, non-negative values

In [104]: a
Out[104]:
array([[ 8,  5, -4],
       [ 4, -8,  6],
       [ 3,  8,  3]])

In [105]: u, spectrum, v = linalg.svd(a) # Singular Value Decomposition (a == u*spectrum*v)

In [106]: spectrum # is the resulting spectrum array
Out[106]: array([ 13.42424973,   9.1261769 ,   6.28509461])

In [107]: s = npy.diag(spectrum)  # 's' is a matrix with non-zeros in its main diagonal

In [108]: s
Out[108]:
array([[ 13.42424973,   0.        ,   0.        ],
       [  0.        ,   9.1261769 ,   0.        ],
       [  0.        ,   0.        ,   6.28509461]])

In [109]: u   # ndarray: unitary matrix having left singular vectors as columns
Out[109]:
array([[-0.61451975, -0.5313394 , -0.58313285],
       [ 0.58595351, -0.80234418,  0.1135883 ],
       [-0.52822719, -0.27188648,  0.80439653]])

In [110]: v  # ndarray: unitary matrix having right singular vectors as rows
Out[110]:
array([[-0.30966539, -0.89286512,  0.32695447],
       [-0.9068147 ,  0.1738915 , -0.38399069],
       [-0.28599729,  0.41539575,  0.86351139]])

In [111]: svd_matrix = u.dot(s).dot(v) # to recompose the original matrix

In [112]: svd_matrix
Out[112]:
array([[ 8.,  5., -4.],
       [ 4., -8.,  6.],
       [ 3.,  8.,  3.]])
```

Linear algebra operations

- Linear algebra operations
  - there is so much more you can do:
    - determinant of a matrix can be found by using linalg.det()
    - computing norms
    - solving linear least-squares problems
    - finding eigenvalues and eigenvectors
    - singular value decomposition SVD (among other decompositions)
    - exponential and logarithm functions
    - trigonometric functions
    - matrix functions and many other special functions

more info: http://docs.scipy.org/

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:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i 2\pi k \frac{n}{N}}$$

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

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

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

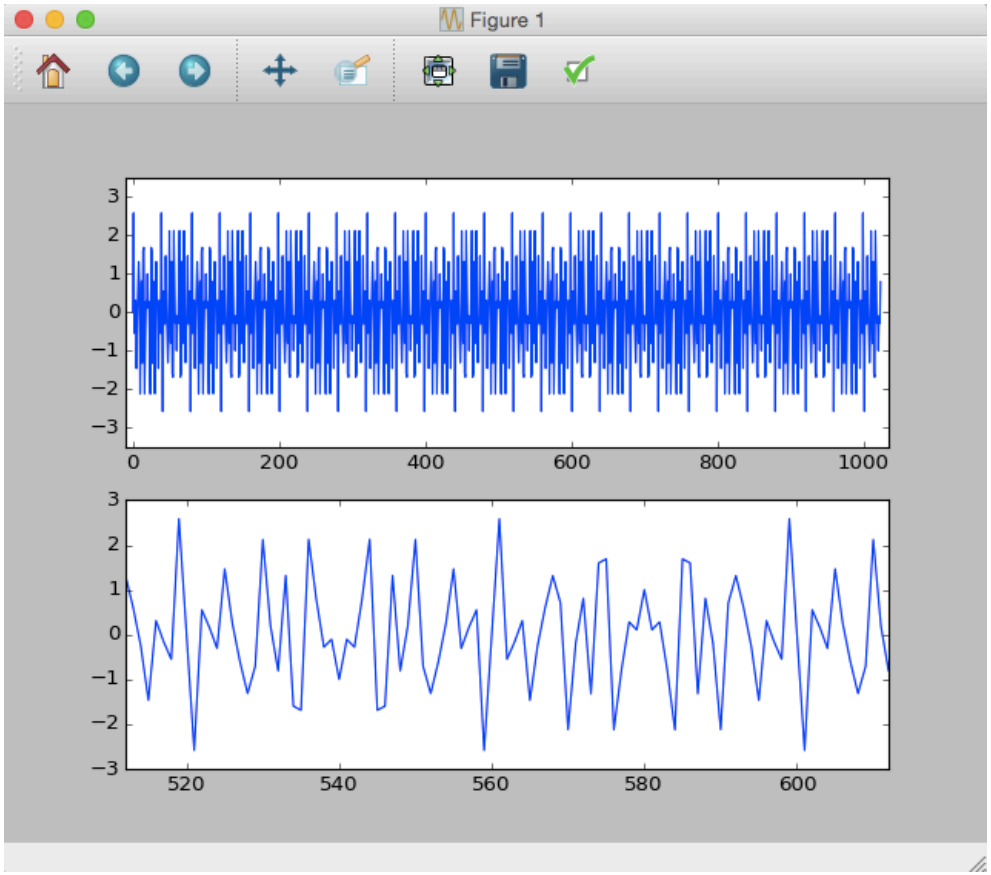
```
243  # FFT example:
244  from numpy.fft import rfft
245  from scipy import arange, sin, pi, fft, real ,imag, log10
246  from scipy.io.wavfile import write
247  from matplotlib.pyplot import figure, plot, subplot, axis, grid
248  from pylab import xticks, yticks, xlim, ylim, xlabel, ylabel, title
249
250  Fs=4000 # sampling frequency
251  a = arange(1024) # create a vector holding the number of bins
252  signal1 = sin(2*pi*a*(650/Fs))   # create a tone with frequency = 650Hz
253  signal2 = sin(2*pi*a*(1150/Fs))   # create a tone with frequency = 1.15kHz
254  signal3 = sin(2*pi*a*(1450/Fs))   # create a tone with frequency = 1.425kHz
255  signal4 = sin(2*pi*a*(1250/Fs))   # create a tone with frequency = 1.25kHz
256
257  # Create a complex tone:
258  signal = signal1 + signal2 + signal3
259
260  # Take the FFT of the complex signal:
261  freq_domain_npy = rfft(signal)  # using npy
262  freq_domain_cpy = fft(signal)   # using cpy
263  bin_val = Fs/len(a)    # calculate the value of each frequeny bin in [Hz]
264  bin_val # each bin in [Hz]
265  t = arange(1,Fs/2+2,bin_val)  # create a vector of frequency bins in [Hz]
266
267  # Save into a wav file:
268  write('files/lecture8/fft_file_example.wav',Fs,signal) # save to file
```

The Fast Fourier Transform

- The Fast Fourier Transform

Example:

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

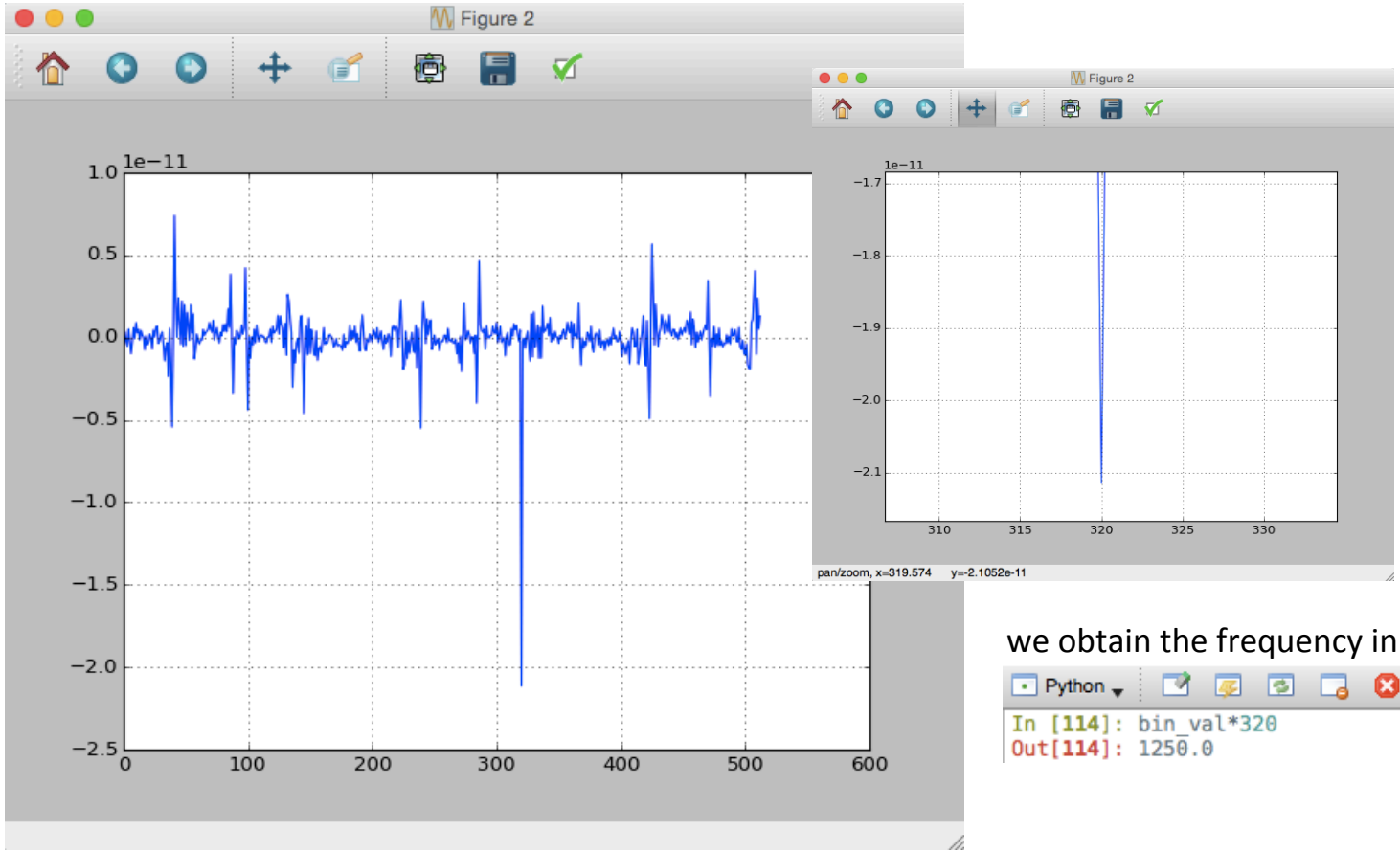


The Fast Fourier Transform

- The Fast Fourier Transform

Example:

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



- observe the quantization noise due to rounding errors

we obtain the frequency in [Hz]:

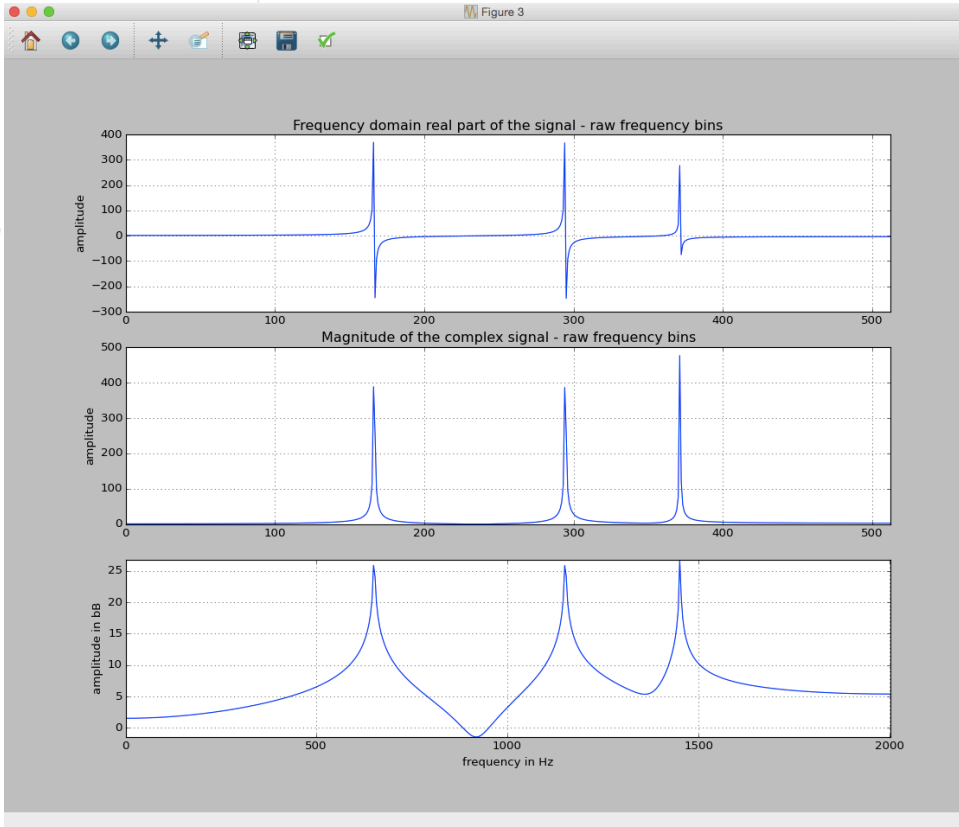
```
Python
In [114]: bin_val*320
Out[114]: 1250.0
```

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)
```



- notice the difference in the x scales

The Fast Fourier Transform

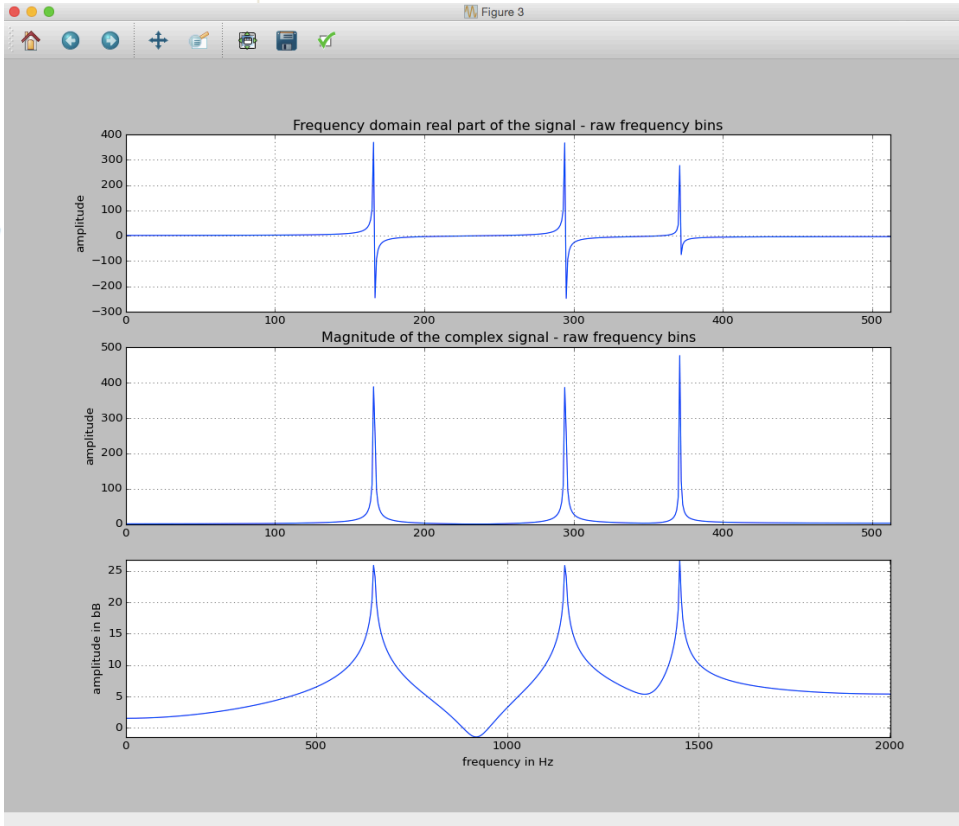
- The Fast Fourier Transform

Example:

```
278  # Plot the Frequency domain signal using numpy:
279  figure(3)
280  subplot(3,1,1),
281  title('Frequency domain real part of the signal - raw frequency bins'),
     plot(freq_domain_npy),
282  xlim(0,len(a)/2),
283  ylabel('amplitude'),
284  grid(True)
285
286  # Plot the magnitude of the complex signal output:
287  subplot(3,1,2),
288  plot(abs(freq_domain_cpy)),
289  title('Magnitude of the complex signal - raw frequency bins'),
290  xlim(0,len(a)/2),
291  ylabel('amplitude'),
292  grid(True)
293
294  # Plot in dB scale:
295  subplot(3,1,3),
296  plot(t,10*log10(freq_domain_npy)),
297  xlabel('frequency in Hz'),
298  ylabel('amplitude in bB'),
299  axis('tight'),
300  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))
```



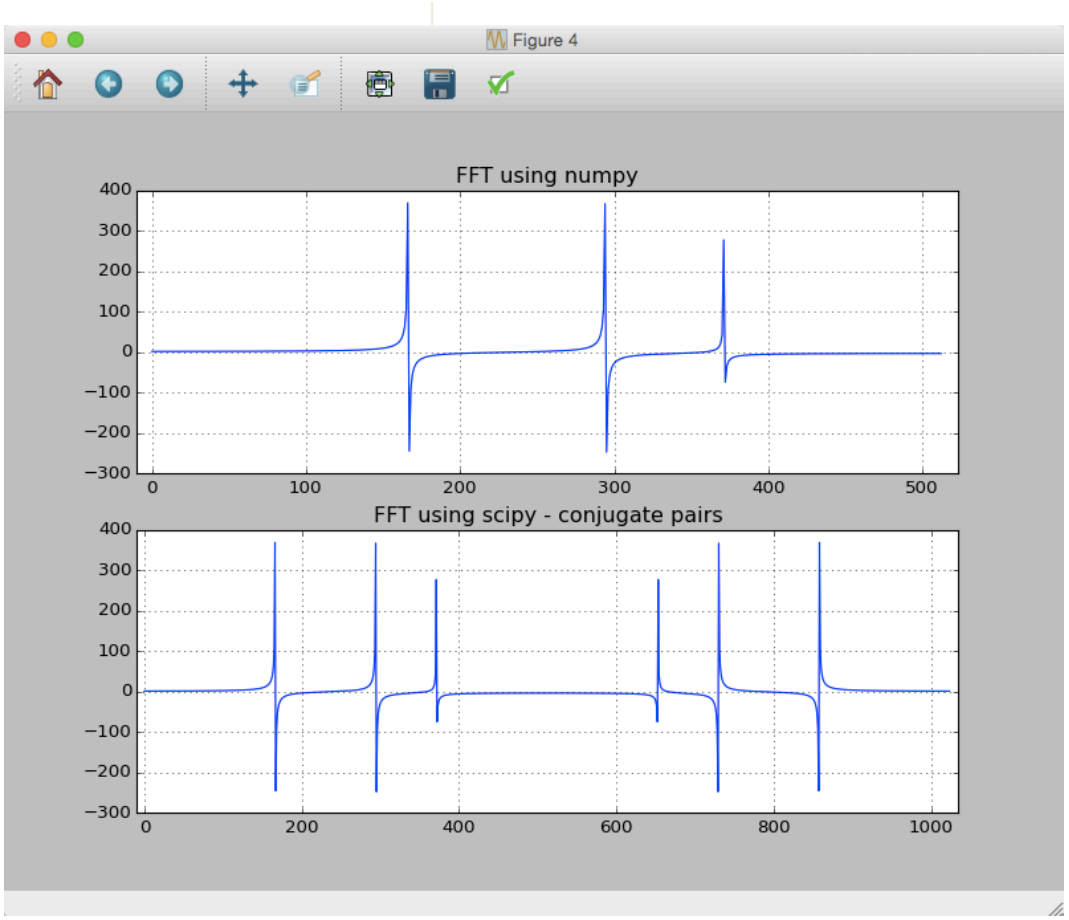
- notice the difference in the x scales

The Fast Fourier Transform

- The Fast Fourier Transform

Example:

```
302 # Plot the Frequency domain signal using numpy:
303 figure(4)
304 subplot(2,1,1),
305 plot(freq_domain_npy),
306 xlim(-10,len(t)+10),
307 title('FFT using numpy'),
308 grid(True)
309
310 # Plot the Frequency domain signal using scipy:
311 subplot(2,1,2),
312 plot(freq_domain_cpy),
313 xlim(-10,len(freq_domain_cpy)+10),
314 title('FFT using scipy - conjugate pairs'),
315 grid(True)
```



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

The Fast Fourier Transform

- The Fast Fourier Transform

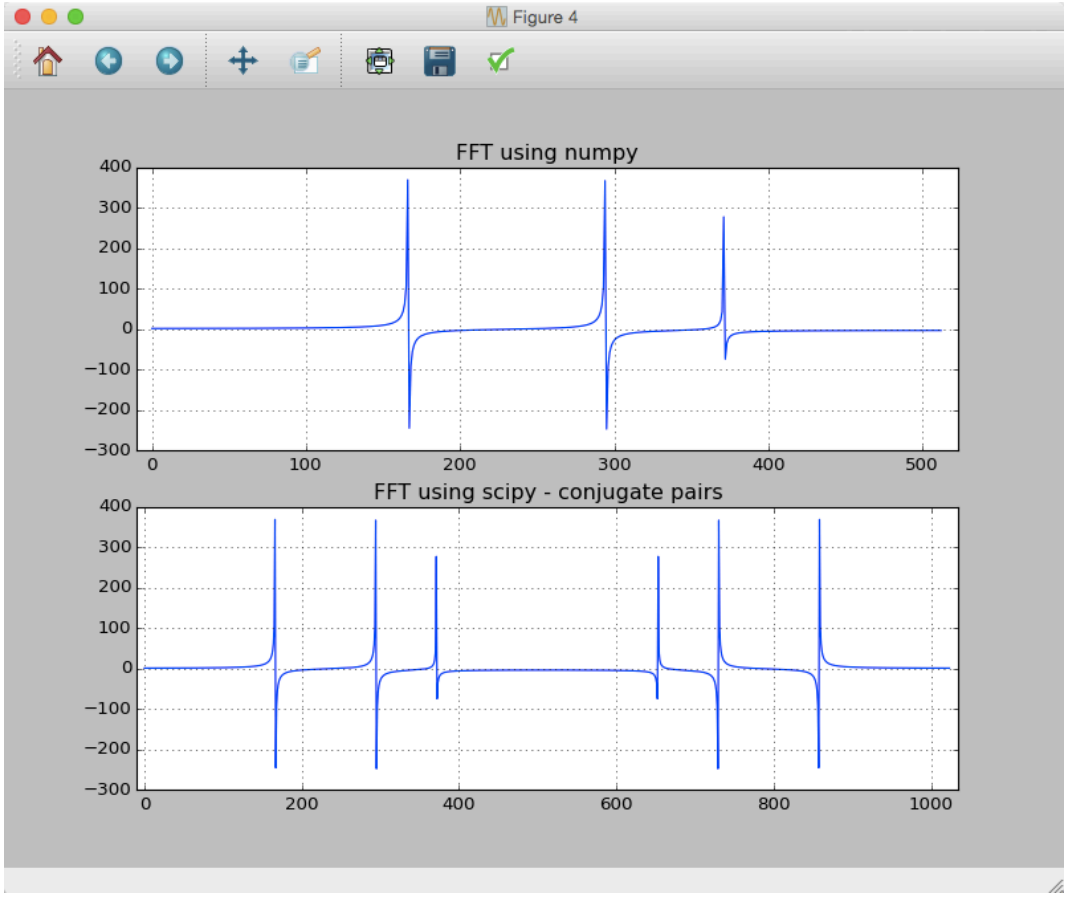
Example:

- the FFT of a real-valued input signal produces a conjugate symmetric result with positive frequencies and their negative counterparts

- so the negative frequencies are just mirrored duplicates of the positive frequencies

- therefore they can just be ignored when analyzing the result

- they are needed when we want to reconstruct the signal back to time domain using the inverse FFT the IFFT



The Fast Fourier Transform

- The Fast Fourier Transform

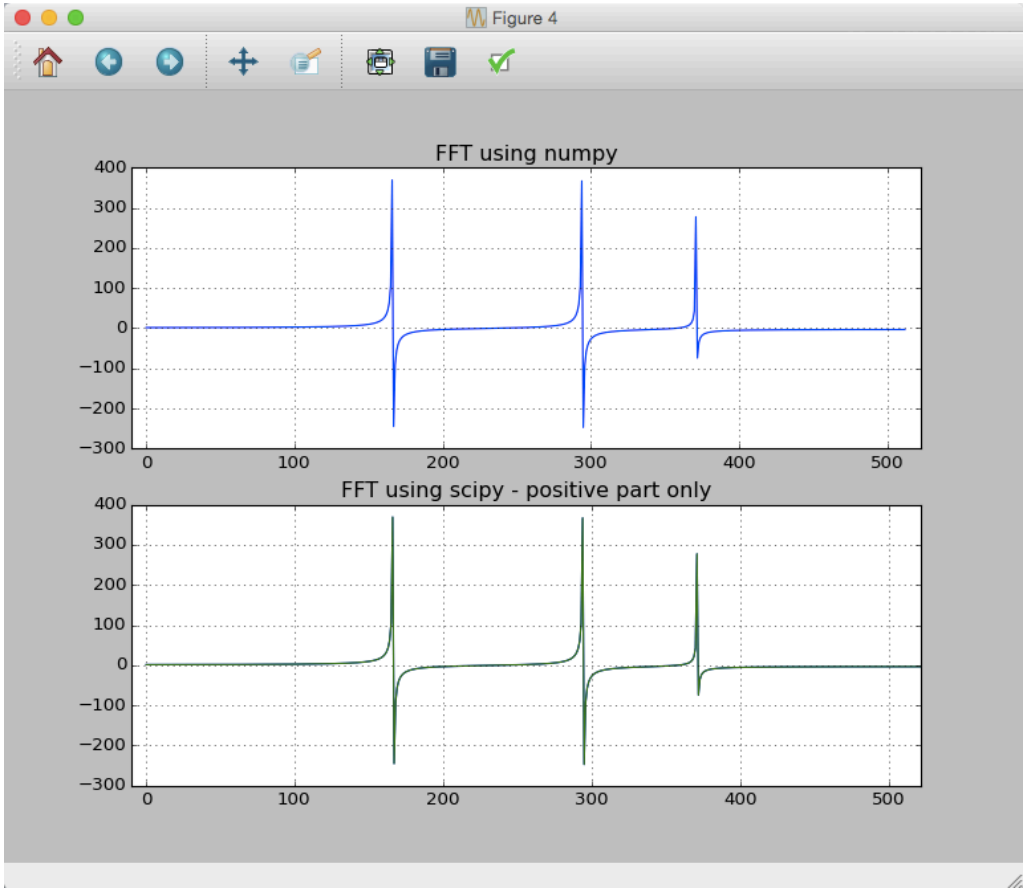
Example:

- the FFT of a real-valued input signal produces a conjugate symmetric result with positive frequencies and their negative counterparts

- so the negative frequencies are just mirrored duplicates of the positive frequencies

- therefore they can just be ignored when analyzing the result

- they are needed when we want to reconstruct the signal back to time domain using the inverse FFT the IFFT



Signal Processing

- Signal Processing
  - there are many Digital Signal Processing (DSP) techniques we can use and develop using Scipy that can be applied to one-dimensional and multi-dimensional arrays on signals such as music, speech and images
  - many of the signal processing tools are located under the scipy.signal package
  - some of the DSP techniques available in Python are:
    - spectrum analysis using different windows – Hamming, Hanning, Bratlett, Blackman, Kaiser
    - recreating the time-domain signal's envelope – using the Hilbert transform
    - performing autocorrelation – correlation of a signal with itself
    - applying convolution – convolving two signals (ex: reverb)
    - digital filter design – Finite Impulse Response (FIR) filter design
    - using transforms – FFT, DCT, DST, Hilbert, etc.

Signal Processing

- Signal Processing

  Below are few of the things that we can do:

  - image processing
    - filtering
    - geometrical transformation of images
    - morphology
    - measurements
  - sound processing (music and speech)
    - spectrogram
    - filter design
    - time & frequency domain manipulations
  - speech processing
    - pitch detection
    - pitch change

Signal Processing

- Signal Processing – image processing:
  - Scipy has several prebuilt functions for image processing
  - The module within Scipy that contains these functions is called ndimage
  - We are not restricted to use image processing tools from one image package only such as Scipy, so we can use misc image methods (Scipy)as well as Pillow (PIL) methods

Example:

```
## 3.6 Image processing:
from numpy import flipud # there is also 'fliplr'
from scipy import ndimage, misc
from PIL import Image
from pylab import imshow, figure, subplot, axis, title, pause
import sys

# Load the file:
img = Image.open('files/lecture9/lena.bmp')
img2 = misc.imread('files/lecture9/lena.bmp', flatten=False)
height, width, channels = img2.shape # we save the image dimentions as parameters

# Notice the difference in image object types:
type(img)
type(img2)

# Show either one of the images the same way:
imshow(img2)
pause(1)
```

when "flatten=True" is the same as monochrome image since 3rd channel is gone

```
In [1]: type(img)
Out[1]: PIL.Image.Image

In [2]: type(img2)
Out[2]: numpy.ndarray
```

Notice how the two objects are of different types

Signal Processing

- Signal Processing – image processing: image transformation
  - Once loaded the image packages, the user has access to a number of basic image manipulation functionality, such as image transformation and filtering

Example:

```
233 # Image transformation:
234 img = img.convert('RGB') # this line will convert our image to RGB
235 img_bw = img.convert('1') # this line will convert our image to black and white
236 img_mn = img.convert('L') # this line will convert our image to monochrome
237 img_rotate = ndimage.rotate(img, 18) # rotate our image at any angle
238 img_shft_pos = ndimage.shift(img2, (75,125,0)) # print image with [x,y] offset
239 img_shft_gam = ndimage.shift(img2, (75,125,1.85)) # change the gamma of the image
240 img_flip = flipud(img2) # this command will flip (not a 180 deg. turn) our image
241 img_zoom1 = ndimage.zoom(img2, (1.45,1.25,1)) # this command will zoom our image
242 img_zoom2 = ndimage.zoom(img2, (1.25,1.45,1)) # this command will zoom our image
243
244 # Saving the image:
245 misc.imsave('files/lecture10/lena_mn.bmp', img_mn)
```

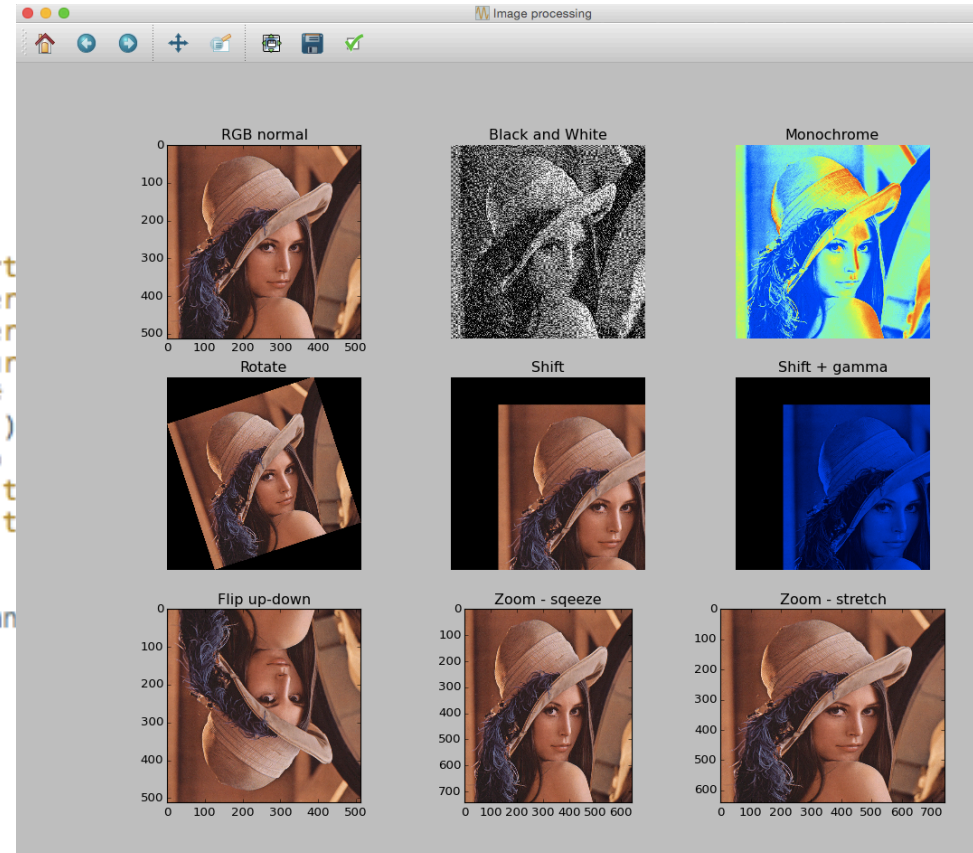
see the saved image

Signal Processing

- Signal Processing – image processing: image transformation
  - Once loaded the image packages, the user has access to a number of basic image manipulation functionality, such as image transformation and filtering

Example:

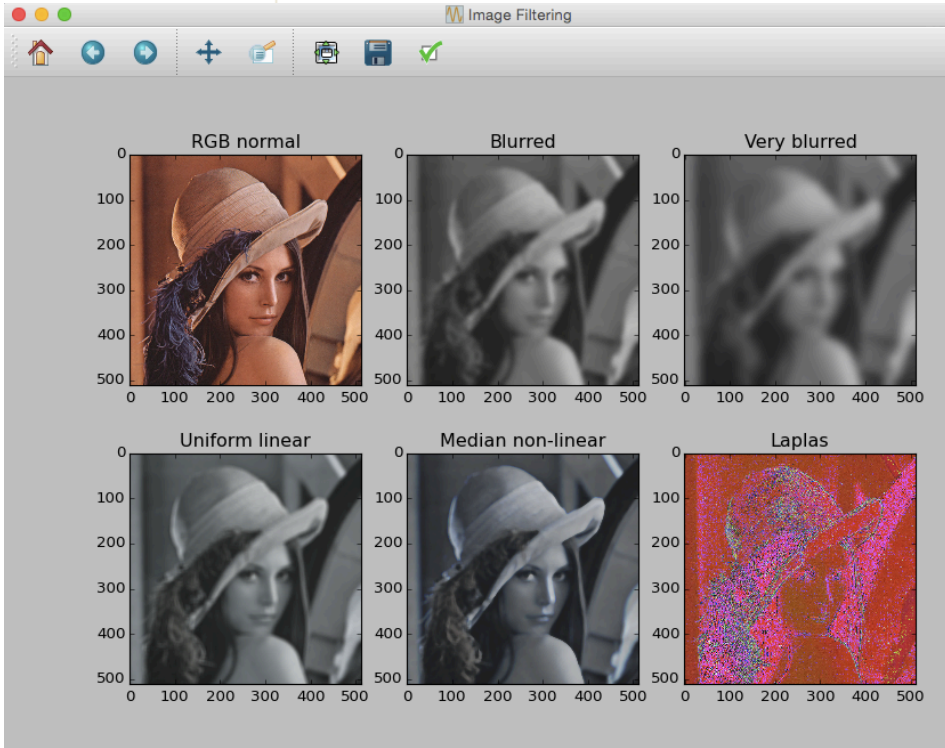
```
233  # Image transformation:
234  img = img.convert('RGB') # this line will convert
235  img_bw = img.convert('1') # this line will conver
236  img_mn = img.convert('L') # this line will conver
237  img_rotate = ndimage.rotate(img, 18) # rotate our
238  img_shft_pos = ndimage.shift(img2, (75,125,0)) #
239  img_shft_gam = ndimage.shift(img2, (75,125,1.85))
240  img_flip = flipud(img2) # this command will flip
241  img_zoom1 = ndimage.zoom(img2, (1.45,1.25,1)) # t
242  img_zoom2 = ndimage.zoom(img2, (1.25,1.45,1)) # t
243
244  # Saving the image:
245  misc.imsave('files/lecture10/lena_mn.bmp', img_mn
```



Signal Processing

- Signal Processing – image processing: image filtering
  - image

```
259 # Filtering:
260 from numpy import copy, random, float
261 from scipy import misc, ndimage, signal
262 from pylab import imshow, figure, subplot, axis, title
263
264 # Load the file:
265 img = misc.imread('files/lecture10/lena.bmp', flatten=False)
266
267 # Gaussian filter:
268 blurring_1 = ndimage.gaussian_filter(img, sigma=5)
269 blurring_2 = ndimage.gaussian_filter(img, sigma=10)
270
271 # Uniform (linear) filter:
272 uniform_img = ndimage.uniform_filter(img, size=11)
273
274 # Median (non-linear) filter:
275 median_img = ndimage.median_filter(img, size=9)
276
277 # Laplas filter:
278 laplas_img = ndimage.laplace(img)
```



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

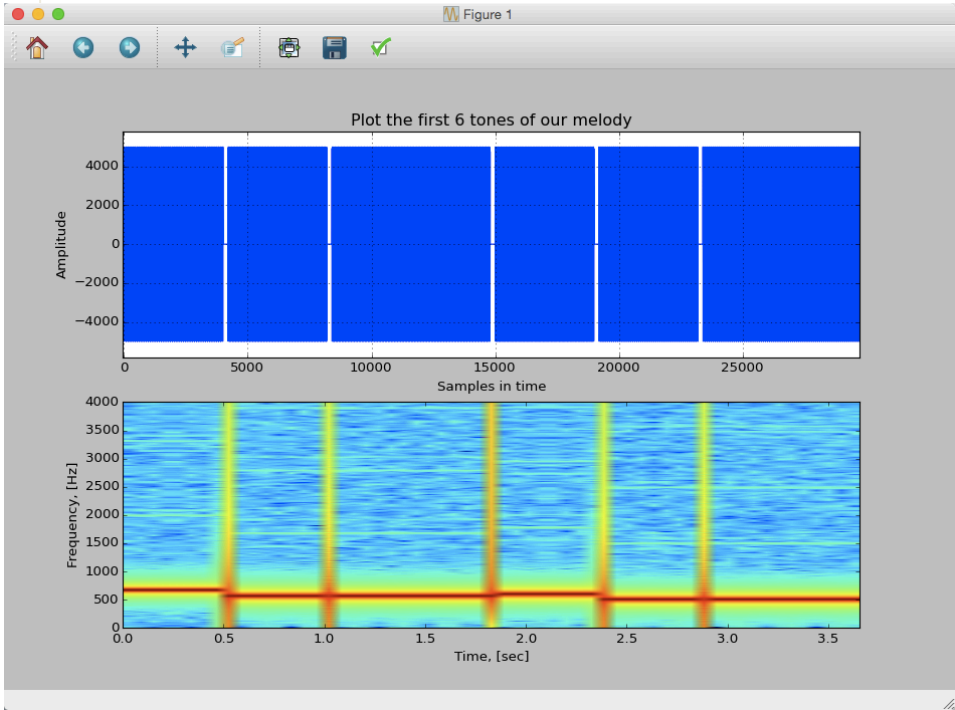
- Signal Processing – sound processing: spectrogram
  - here are some of the parameters users have control over:
    - Fs – the sampling frequency calculating the Fourier frequencies, in cycles per time
    - NFFT – the amount of frequency bins represented in each FFT window
    - window type used – 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:

```
## Scipy 3/3

# Signal Processing:
# 1. Spectrogram:

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

(Fs, x) = read('files/lecture10/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, Fc=0, noverlap=12, mode='magnitude')
axis('tight')
```



Signal Processing

- Signal Processing – sound processing: spectrogram

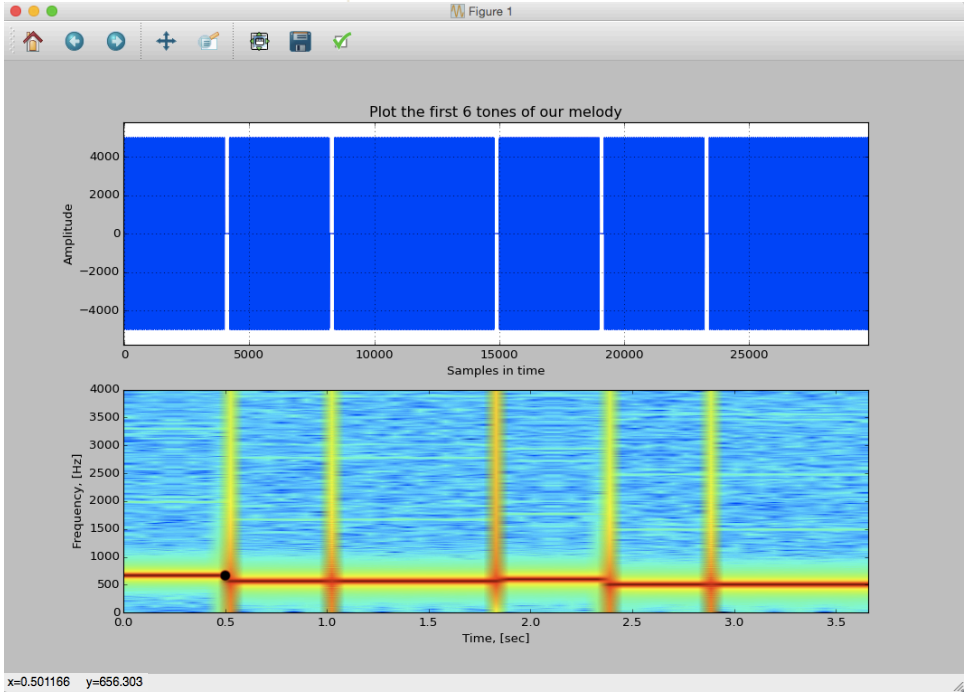
  Example:

  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

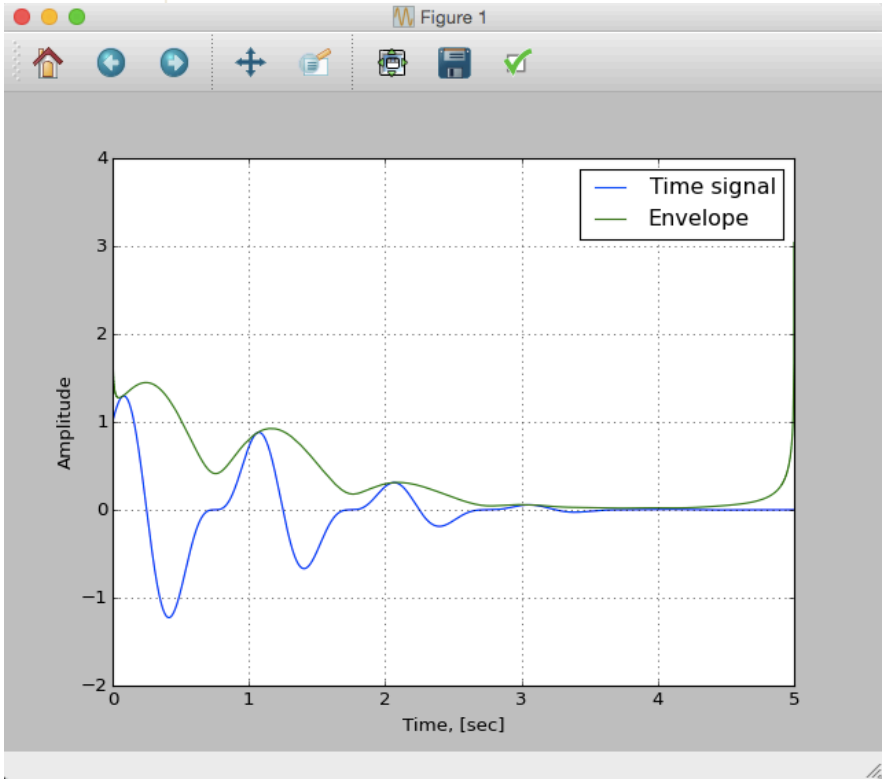
- Signal Processing – sound processing: envelope
  - the envelope of a time-domain signal is a smooth curve around the extreme parts of that signal so it connects all the peaks in a signal and is the estimate of the total signal level of the audio input
  - the envelope detection methodology of squaring and lowpass filtering a signal works like this:
    - it squares the input signal, due to which half of its energy is shifted up to higher frequencies and half is going down toward DC
    - then sending it through a minimum-phase lowpass filter (LPF), to eliminate any high frequency energy thus avoid aliasing (FIR decimation)
    - the signal is downsampled
  - simpler way of calculating the envelope of a signal is to just use the Hilbert transform
  - the Hilbert transform is used to calculate the analytic signal – one without negative components. They can be discarded at no loss
  - the analytic signal is a complex signal, with real part represented by the original signal and imaginary part by the Hilbert transform of that same signal
  - the envelope signal can be used to observe how the signal changed after some frequency components were suppressed or reinforced by filtering of the original signal

Signal Processing

- Signal Processing – sound processing: envelope

Example:

```
25  # 2. Envelope:
26  import scipy.signal.signaltools as sigtool
27  from numpy import linspace, sin, cos, pi, exp, abs
28  from pylab import plot, grid, legend, show, xlabel, ylabel
29
30  # Lets create a simple signal using the 'sin' and 'cos' functions:
31  a = linspace(0,5,1e5)
32  b = (1+sin(2*pi*a))*exp(-a**2/3)*(cos(2*pi*a))
33
34  # Now we take the envelope using 'Hilbert's' transform:
35  envelope = abs(sigtool.hilbert(b))
36
37  # Plotting:
38  plot(a,b,label='Time signal')
39  plot(a,envelope,label='Envelope')
40  xlabel('Time, [sec]'), ylabel('Amplitude'), legend(); grid(True)
41  show()
```



- the shape of the envelope matches the actual signal
- the amplitude of the envelope does not match the original signal, because the Hilbert transform was implemented using a non-ideal FIR filter, so the magnitude response is not one for all frequencies

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

Words	Ladefoged (2006)	Roach (2009)	Words	Ladefoged (2006)	Roach (2009)
feet ★	/i/	/iː/	bird★	/ɜ/,/ɜˠ/	/ɜɪ/
hard★	/ɑ/	/ɑː/	bed★	/ɛ/	/e/
food★	/u/	/uː/	attend	/ə/	/ə/
lord★	/ɔ/	/ɔɪ/	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					

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

Signal Processing

- Signal Processing – sound processing: speech

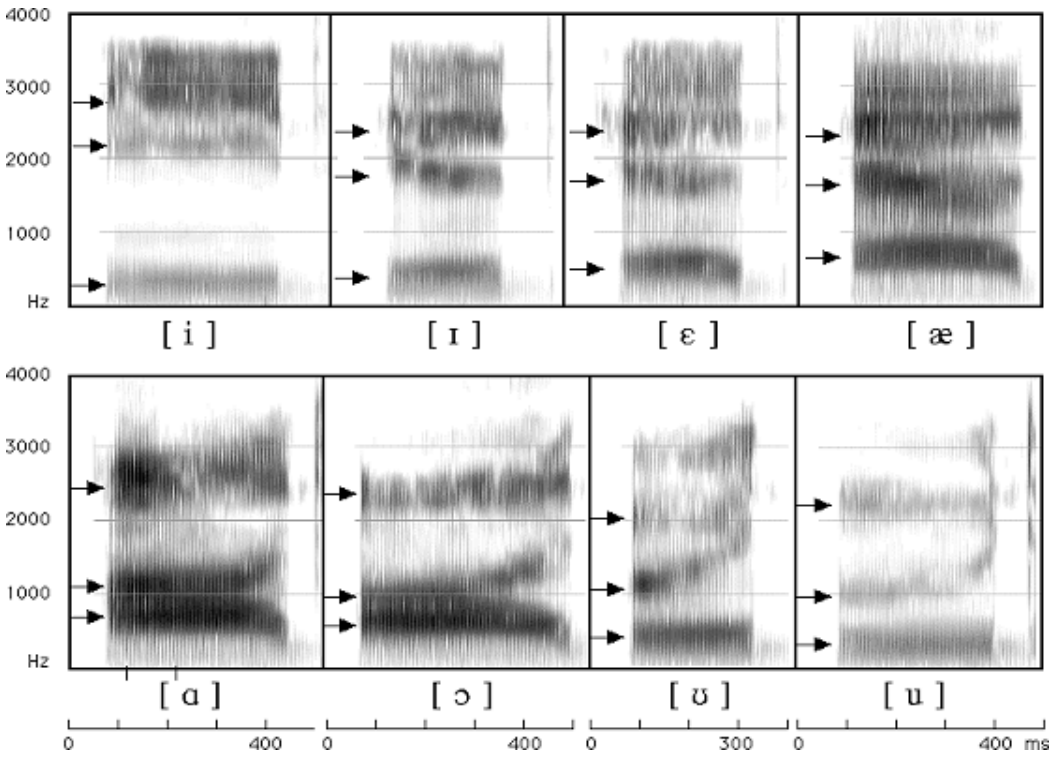
Formants are resonances in the vocal tract

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

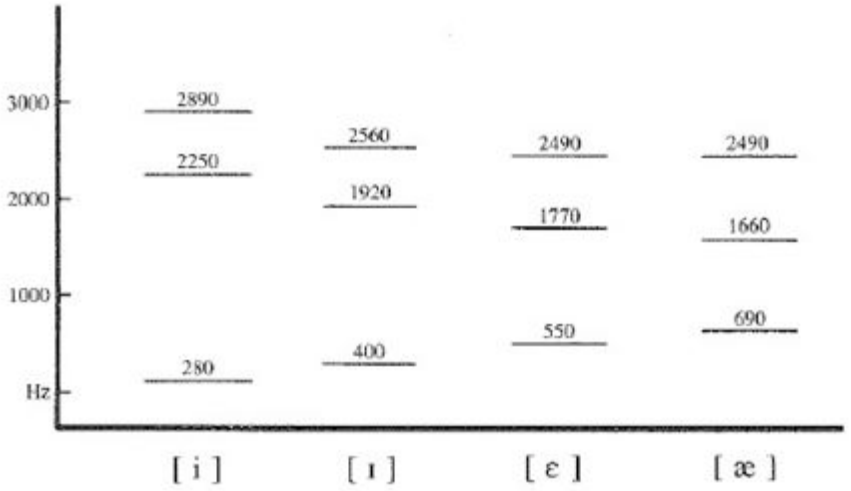
Formants are resonances in the vocal tract

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

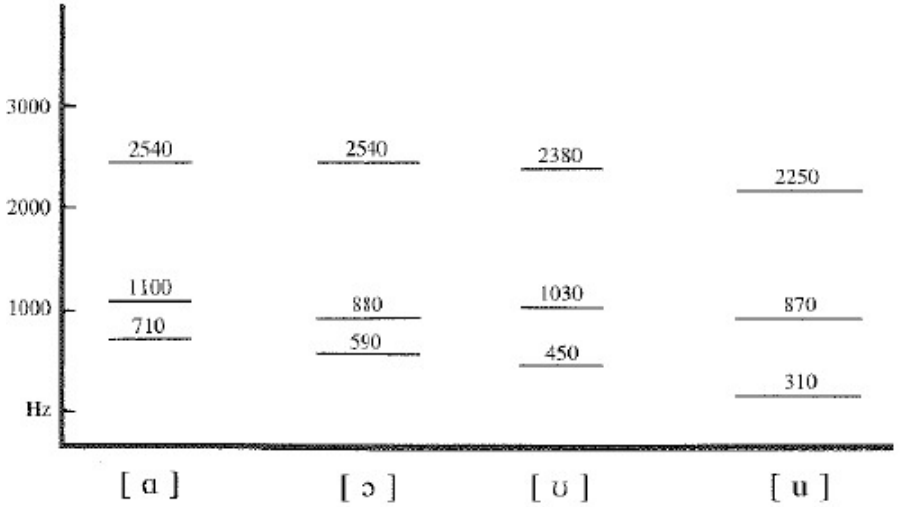
Formants are resonances in the vocal tract

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

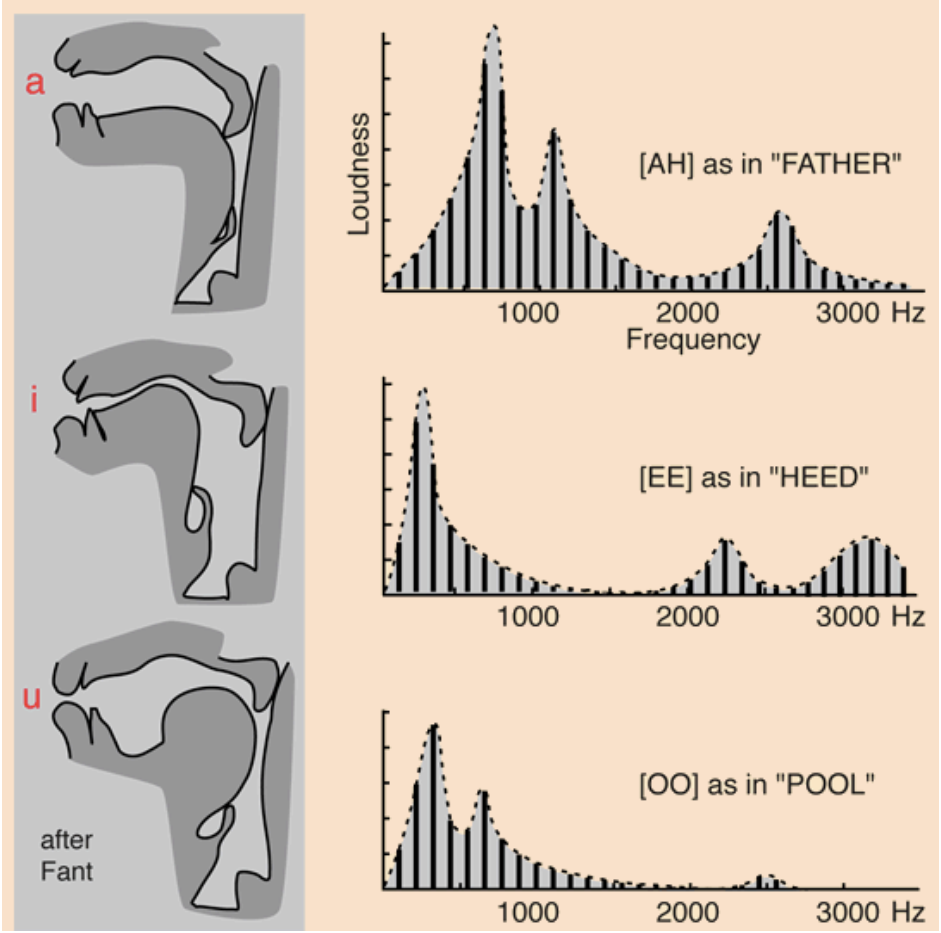
- Signal Processing – sound processing: speech

See levels for each formant

Each vowel is formed by formants

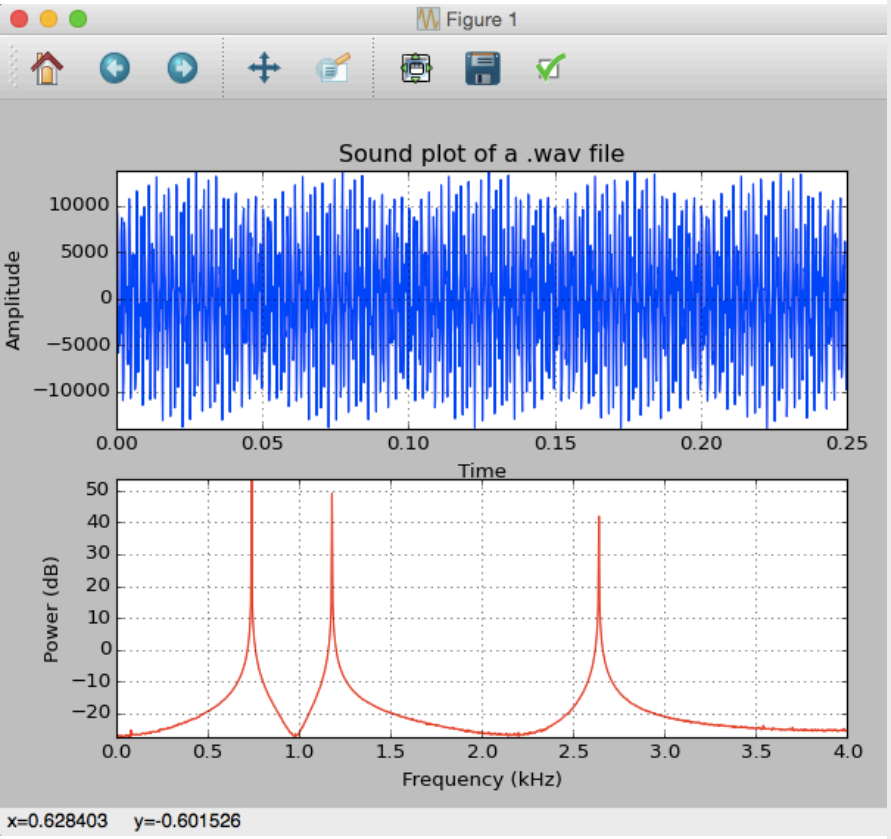
Each formant has a different frequency and amplitude

Formants for different genders or kids vary



Signal Processing

- Signal Processing – sound processing: power frequency spectrum



lecture8.py | lecture9.py | power_freq_spec.py

```
## Visualizing the power frequency spectrum of a signal:
from pylab import linspace, subplot, plot, title, xlabel, ylabel, grid, axis
from numpy import arange, log10, sqrt, mean, sum
from scipy.io.wavfile import read
from scipy import fft, ceil

# We read the sound file of interest:
(Fs, x) = read('vowel_A.wav') # Fs - sampling frequency, x - signal

# We create a windown for analysis, typecasting it to int (float will produce an error):
win = int(Fs/4)

# We take the FFT with a specific window lenght:
x_fft = fft(x, win)

# Now we obtain the magnitude of all frequencies in the signal using 'abs',
# then we scale by the number of signal points in order for the magnitude to be
# independent from the signal length 'len(x)', and finally we obtain the
# power of the signal by using '**2':
sig_pow = ( abs(x_fft) / len(x) )**2

# We create an array containing the frequencies for the particular window used:
Freqs = arange(0, win, 2) * (Fs / win);

# Use only left side of the frequency power spectrum of the signal:
sig_pow = sig_pow[0:(len(sig_pow))/2]

# We plot the time-domain window of samples:
time = win/Fs # calculate the length of the .wav file in secs
t = linspace(0,time,win) # create evenly spaced numbers between [0:time]
subplot(2,1,1), plot(t,x[0:win]) # plot signal 'x'
title('Sound plot of a .wav file')
xlabel('Time')
ylabel('Amplitude')
axis('tight')
grid(True)

# We scale the frequency array to [kHz] then plot the signal in dB by using 10*log10(p):
subplot(2,1,2), plot(Freqs/win, 10*log10(sig_pow), color='r')
xlabel('Frequency (kHz)')
ylabel('Power (dB)')
axis('tight')
grid(True)
```

Homework #4

Generate the English phonemes corresponding to the vowels: a, e, i, o, u

1. Synthesize each formant tone in preparation for creating each vowel

   We will simplify the synthesis of each vowel: a, e, i, o, u accepting the following constrains:

   1. you will use pure tones only (serving as center frequencies) using the sin function
   2. you will manually adjust the amplitude of each formant frequency you generate
   3. you will use the first three formants for the average male speaker

   Note: in the exercise we will not use: glottal pulse train simulation, vocal tract estimation, female/child speaker formants, formant bandwidth reproduction, so it will be a simplified model.
   Only addition of simple tones will be used to represent each vowel.

2. Create a function that generates each simple formant tone with sampling frequency Fs=8kHz, duration 2 seconds, and different amplitudes based on the spectrograms of the American English Vowels provided by Ladeforged 2006:185-187. Refer to the spectrogram provided in the lectures and use your judgment. Obtain best results by testing and repetition!
3. Use the formant frequencies given by J.C. Wells - refer to the table provided in the lectures
4. Simply add all three formants for each vowel and save the files in .wav format
5. Extra point (optional): you can incorporate the code I provided in the previous slide as a module.py to plot each vowel sound. In this case send me both files.
6. Organize your code: use each line from this HW as a comment line before coding each step
7. Save these steps in a .py file and email it to me before next class. I will run it!