

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

- **Scipy 2/2**
 - Working with databases
 - Interpolation
 - Statistics
 - Optimization
 - Multithreading and Multiprocessing

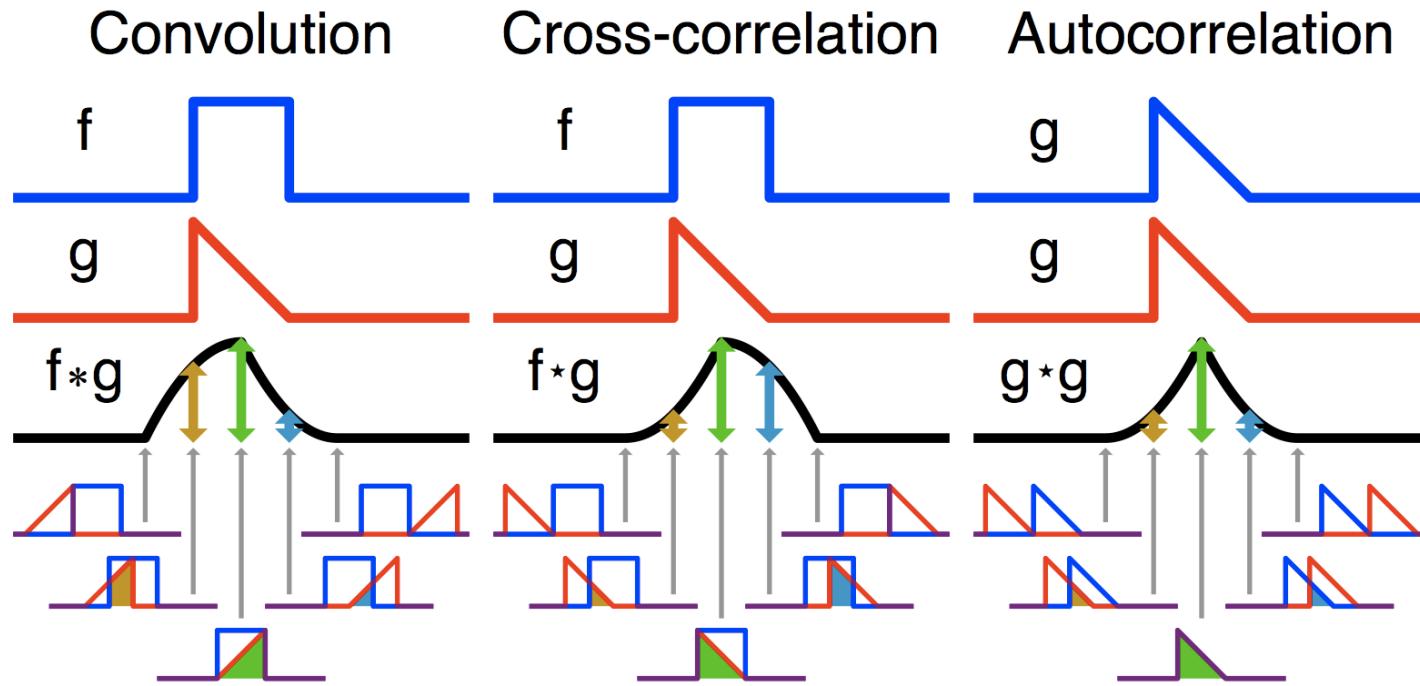
- **Project**
 - Project Presentation

Final Project

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.

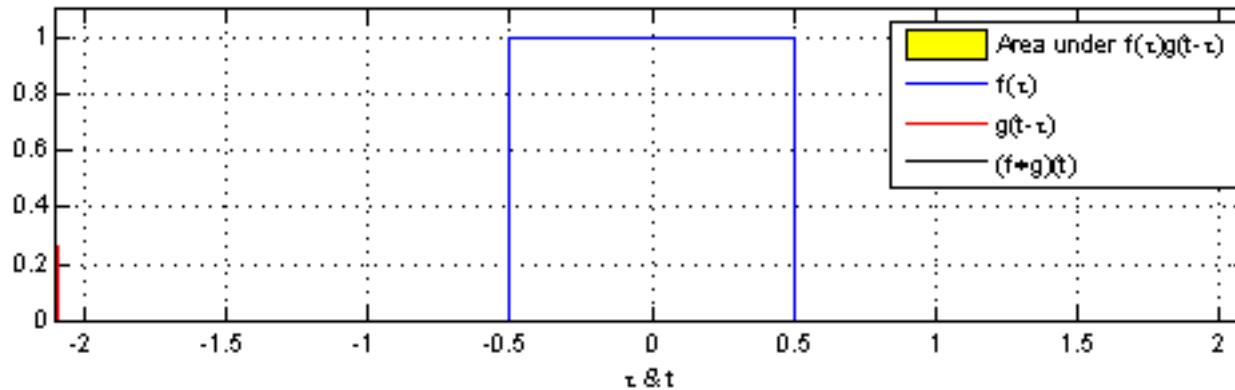
Signal Processing



Source Wikipedia

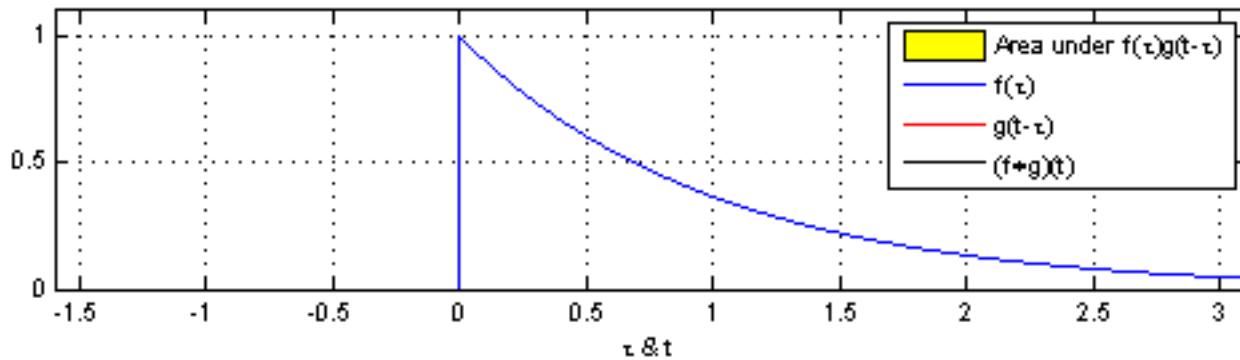
Signal Processing

- Convolving two signals:



box vs box

spiky vs box



Source Wikipedia

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:

```
217 ## 3.6 Image processing:  
218 from numpy import flipud # there is also 'fliplr'  
219 from scipy import ndimage, misc  
220 from PIL import Image  
221 from pylab import imshow, figure, subplot, axis, title, pause  
222 import sys  
223  
224 # Load the file:  
225 img = Image.open('files/lecture9/lena.bmp')  
226 img2 = misc.imread('files/lecture9/lena.bmp')  
227 height, width, channels = img2.shape # we save the image dimentions as parameters  
228  
229 # Notice the difference in image object types:  
230 type(img)  
231 type(img2)  
232  
233 # Show either one of the images the same way:  
234 imshow(img2)  
235 pause(1)
```

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)
```



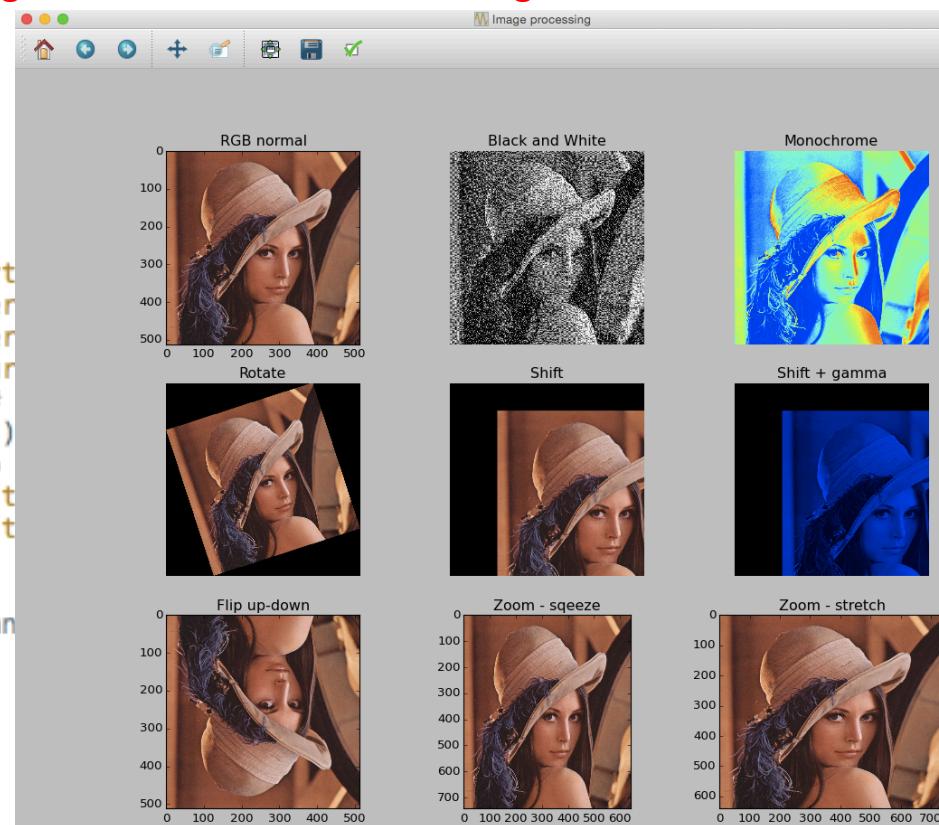
see the saved image

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:

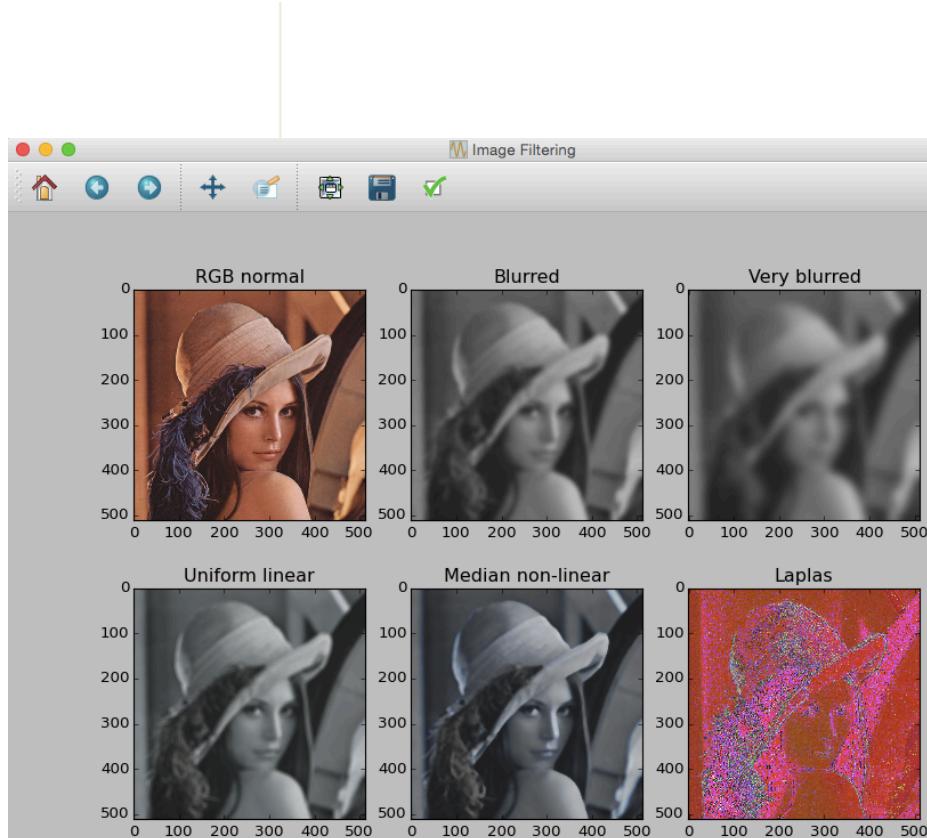
```
233 # Image transformation:  
234 img = img.convert('RGB') # this line will convert  
235 img_bw = img.convert('1') # this line will convert  
236 img_mn = img.convert('L') # this line will convert  
237 img_rotate = ndimage.rotate(img, 18) # rotate our  
238 img_shft_pos = ndimage.shift(img2, (75,125,0)) #  
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

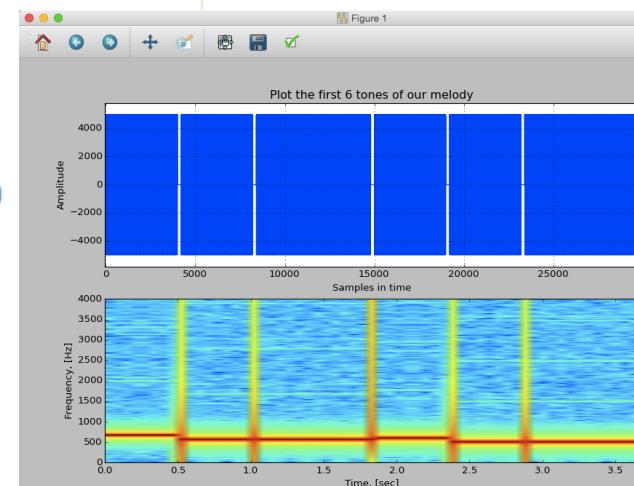
- Signal Processing – sound processing: spectrogram
 - here are some of the parameters users have control over:
 - **Fs** – the **sampling frequency** calculating the Fourier frequencies, in cycles per time
 - **NFFT** – the amount of **frequency bins** represented in each FFT window
 - **window** type 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 $(10 * \log_{10})$, otherwise it is $(20 * \log_{10})$
 - **Fc** – the center frequency can be controlled
 - **cmap** – is the colormap chosen

Signal Processing

- Signal Processing – sound processing: spectrogram

Example:

```
3 # Signal Processing:  
4 ## 1. Spectrogram:  
5  
6 from pylab import specgram, plot, subplot, title, xlabel, ylabel, grid, axis, xlim, ylim, pause  
7 from scipy.io.wavfile import read  
8  
9 (Fs, x) = read('files/lecture9/melody.wav') # Fs - sampling frequency, x - signal  
10  
11 # Lets plot signal 'x':  
12 subplot(2,1,1)  
13 plot(x)  
14 xlim([-50,29750]); ylim([-5800,5800]) # limit it to the first 6 tones only  
15 title('Plot the first 6 tones of our melody')  
16 xlabel('Samples in time'); ylabel('Amplitude')  
17 grid(True)  
18  
19 # Now we plot the spectrogram of the first 6 tones:  
20 subplot(2,1,2)  
21 xlabel('Time, [sec]'); ylabel('Frequency, [Hz]')  
22 specgram(x[0:29750], NFFT=512, Fs=Fs, noverlap=64, mode='magnitude')  
23 pause(1)
```



Signal Processing

- Signal Processing – sound processing: spectrogram

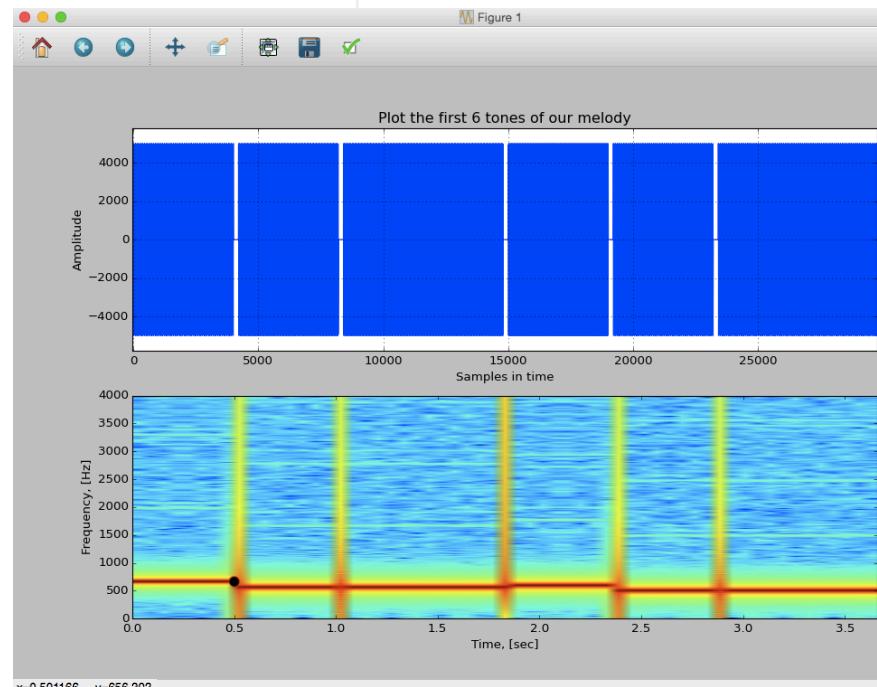
Example:

lets recall the code snipped from the melody:

```
97 # Creating each tone with Fs samples per second and length 0.5 or 0.8 seconds:  
98 Es = note(659,Fs,0.5,amplitude=5000)  
99 El = note(659,Fs,0.8,amplitude=5000)  
100 Css = note(554,Fs,0.5,amplitude=5000)  
101 Csl = note(554,Fs,0.8,amplitude=5000)  
102 D = note(587,Fs,0.5,amplitude=5000)  
103 Bs = note(494,Fs,0.5,amplitude=5000)  
104 Bl = note(494,Fs,0.8,amplitude=5000)  
105 As = note(440,Fs,0.5,amplitude=5000)  
106 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

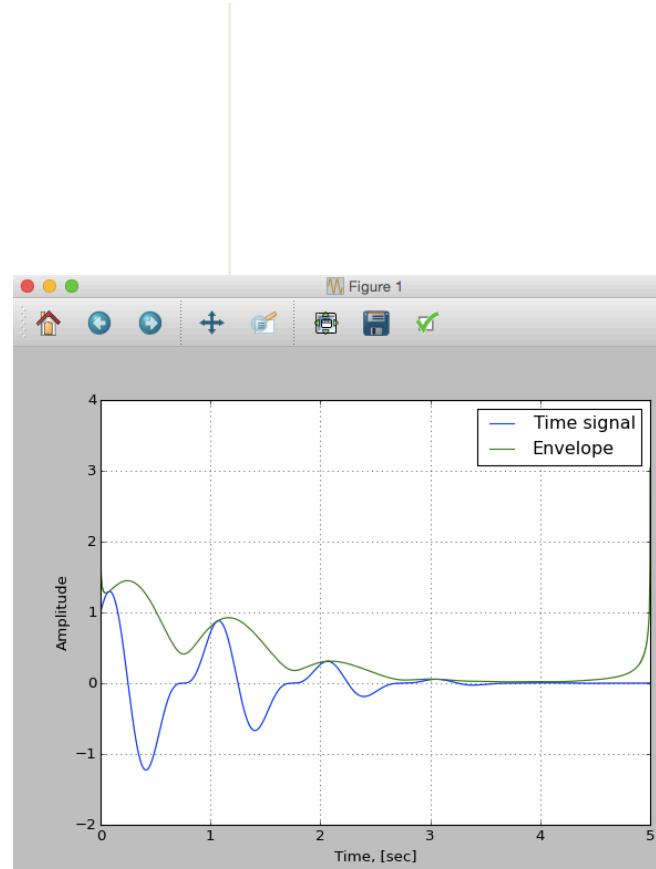
Signal Processing

- 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

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

Words ^②	Ladefoged (2006) ^③	Roach (2009) ^④	Words ^②	Ladefoged (2006) ^③	Roach (2009) ^④
feet ★ ^⑤	/i/ ^⑥	/ɪ/ ^⑥	bird★ ^⑤	/ɜ:/, /ə:/ ^⑥	/ə/ ^⑥
hard★ ^⑤	/ɑ/ ^⑥	/a:/ ^⑥	bed★ ^⑤	/ɛ/ ^⑥	/e/ ^⑥
food★ ^⑤	/u/ ^⑥	/ʊ/ ^⑥	attend ^⑦	/ə/ ^⑥	/ə/ ^⑥
ord★ ^⑤	/ɔ/ ^⑥	/ɔ:/ ^⑥	book ^⑦	/ʊ/ ^⑥	/ʊ/ ^⑥
hot★ ^⑤	/a/(GA), ^⑧ /ɒ/ (RP) ^⑨	/ɒ/ ^⑩	go★ ^⑤	/oʊ/(GA), ^⑧ /əʊ/(RP), ^⑨	/əʊ/ ^⑩
bus, ^⑩	/ʌ/ ^⑪	/ʌ/ ^⑪	boy ^⑩	/ɔɪ/ ^⑫	/ɔɪ/ ^⑪
book ^⑩	/ʊ/ ^⑪	/ʊ/ ^⑪	big ^⑩	/ɪ/ ^⑫	/ɪ/ ^⑪
dear, ^⑩	/ɪə/ ^⑪	/ɪə/ ^⑪	care★ ^⑤	/ɛə/ ^⑫	/eə/ ^⑪
bike ^⑩	/aɪ/ ^⑪	/aɪ/ ^⑪	how ^⑩	/au/ ^⑫	/au/ ^⑪
cake ^⑩	/eɪ/ ^⑪	/eɪ/ ^⑪	tour ^⑩	/uə/ ^⑫	/uə/ ^⑪

★ =There is a difference between two systems^⑪

Signal Processing

- Signal Processing – sound processing: speech

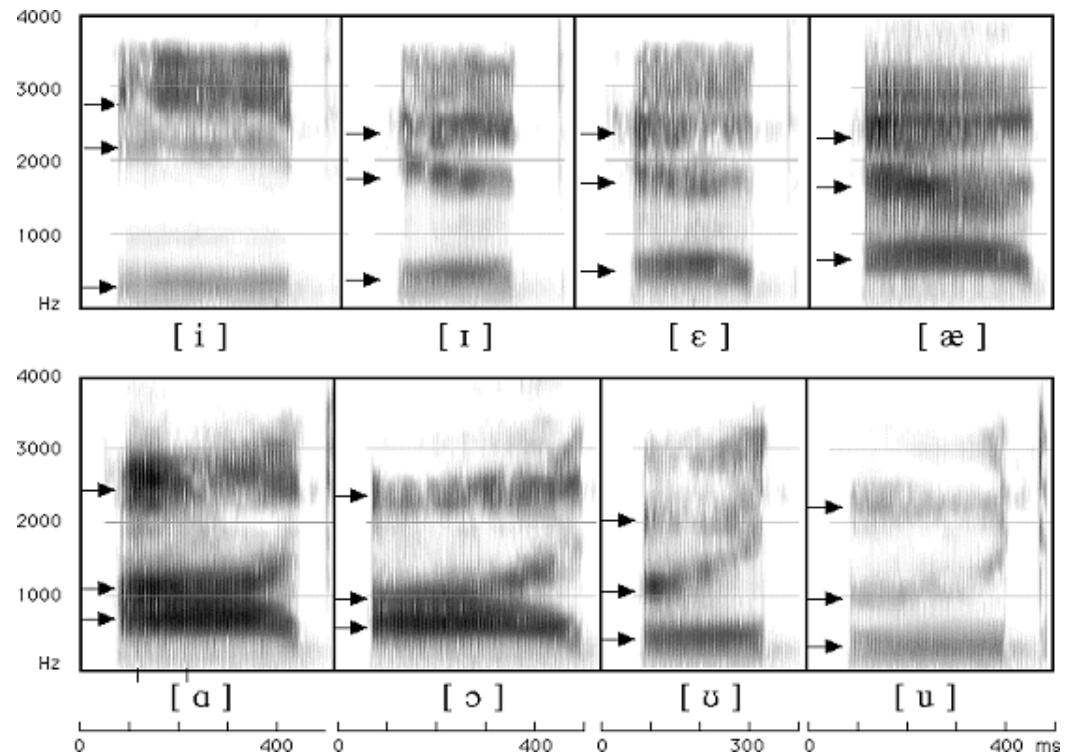
Formants are resonances in the vocal tract

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

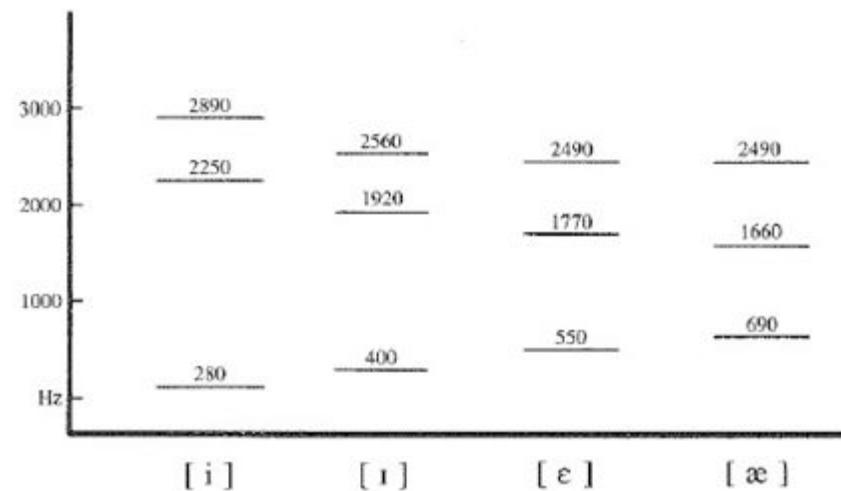
Formants are resonances
in the vocal tract

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

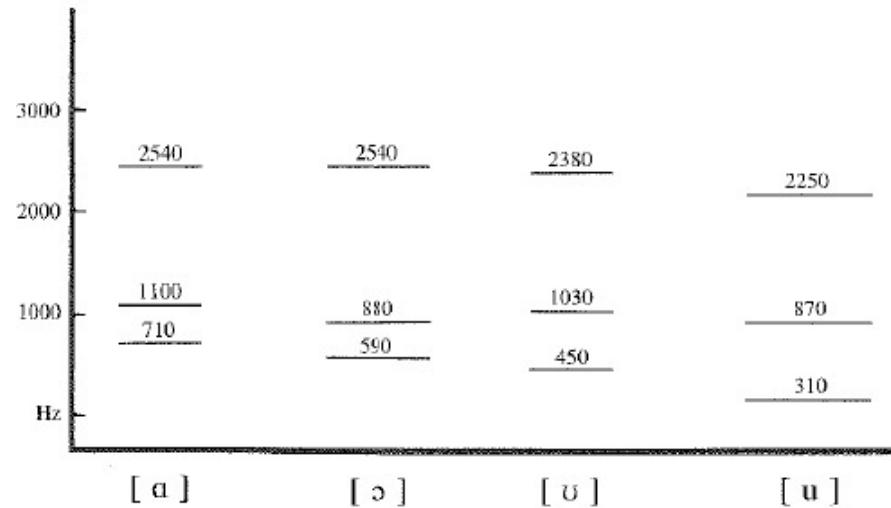
Formants are resonances
in the vocal tract

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)
i:	280	2620	3380
I	360	2220	2960
e	600	2060	2840
æ	800	1760	2500
ʌ	760	1320	2500
a:	740	1180	2640
ɒ	560	920	2560
ɔ:	480	760	2620
ʊ	380	940	2300
u:	320	920	2200
ɜ:	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

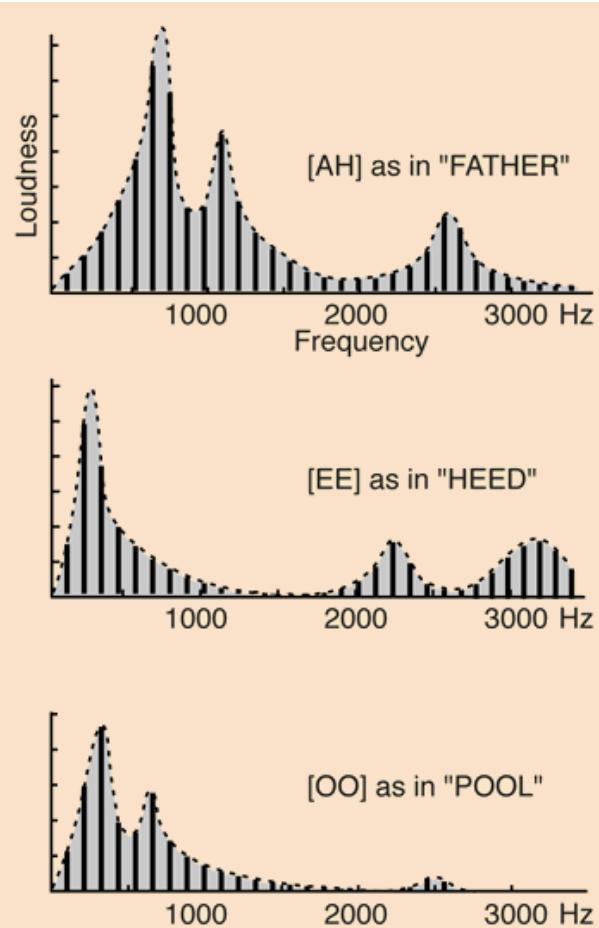
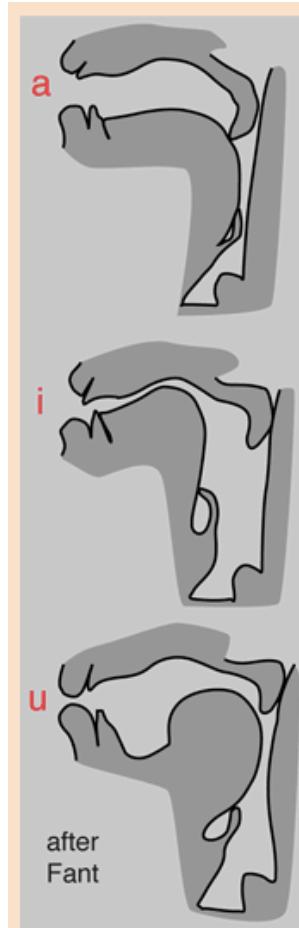
- Signal Processing – sound processing: speech

See levels for each formant

Each vowel is formed by formants

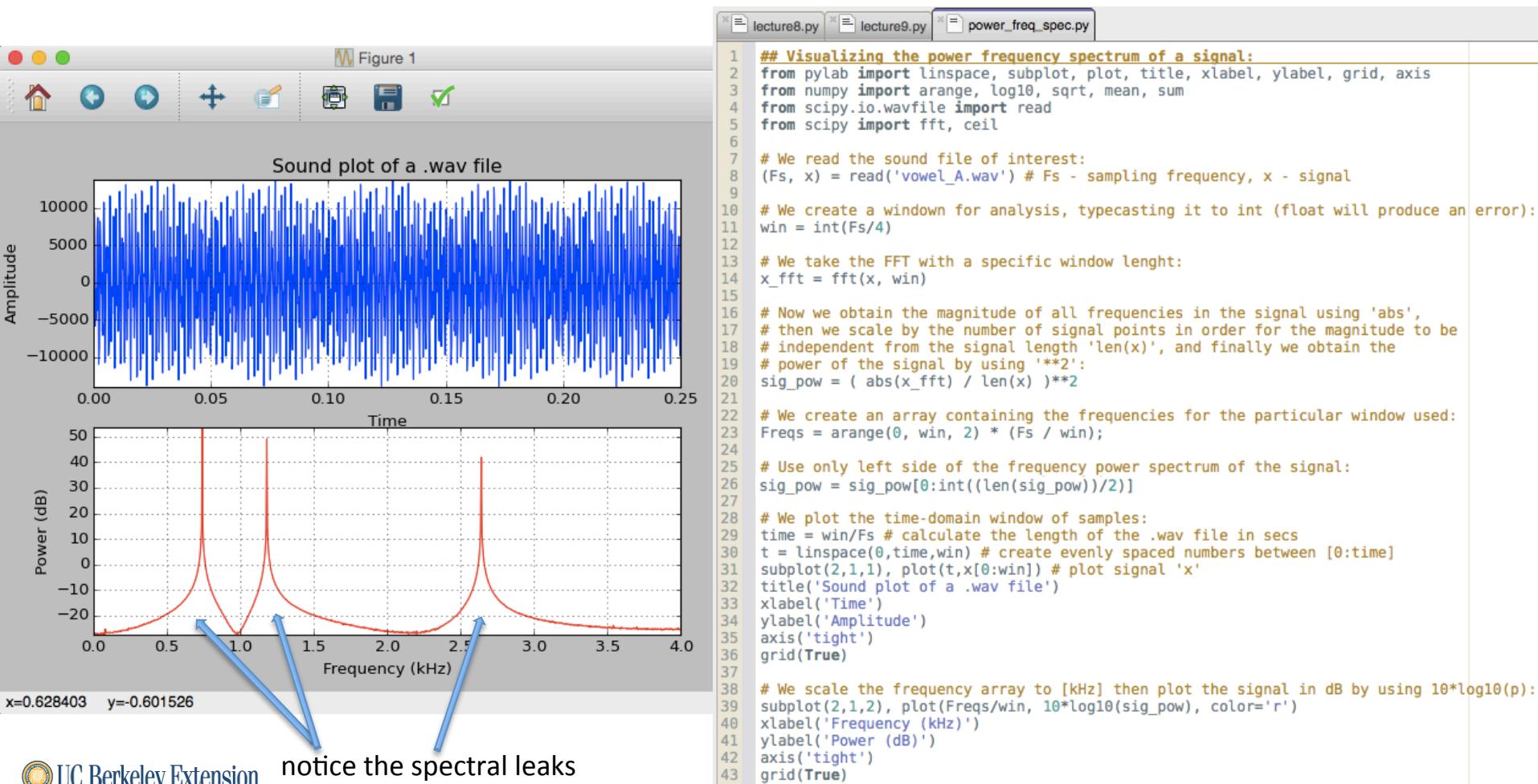
Each formant has a different frequency and amplitude

Formants for different genders or kids vary



Signal Processing

- Signal Processing – sound processing: power frequency spectrum



Signal Processing

- Signal Processing – sound processing: resampling
 - digital audio, just like anything digital is a representation of finite number of digits in order to be stored in a computer
 - in the audio world sound is represented by samples and bits per sample, that define the sound resolution
 - it follows that, sound quality can vary when we change these two parameters
 - resampling or changing the bit resolution per sample is performed for different reasons depending on the case
 - when we do resampling we either up-sample or down-sample

Signal Processing

- Signal Processing – sound processing: **resampling**
 - samples:
 - samples represent the frequency resolution of an audio
 - **down-sampling**:
 - » it is used in order to **reduce** the amount of information in the audio
 - » it follows that when we down-sample we **decrease frequency resolution**
 - **up-sampling**:
 - » up-sampling is usually **used for sampling rate matching** in preparation for another process involving multiple signals
 - » **adds samples between** the ones of the original signal (**no new information**)
 - » no new information added means, no new frequencies will be present, hence **no sound quality improvement**
 - » up-sampling is actually an **interpolation**, and can also be thought of as **curve fitting**
 - bits:
 - they are the **building parts of a sample** and define the sample resolution
 - **less bits** per sample decreases resolution and makes the sound **more noisy**
 - **increasing sample resolution** is used for calculations so the **rounding (quantization) error** is lower

Signal Processing

- Signal Processing – sound processing: resampling

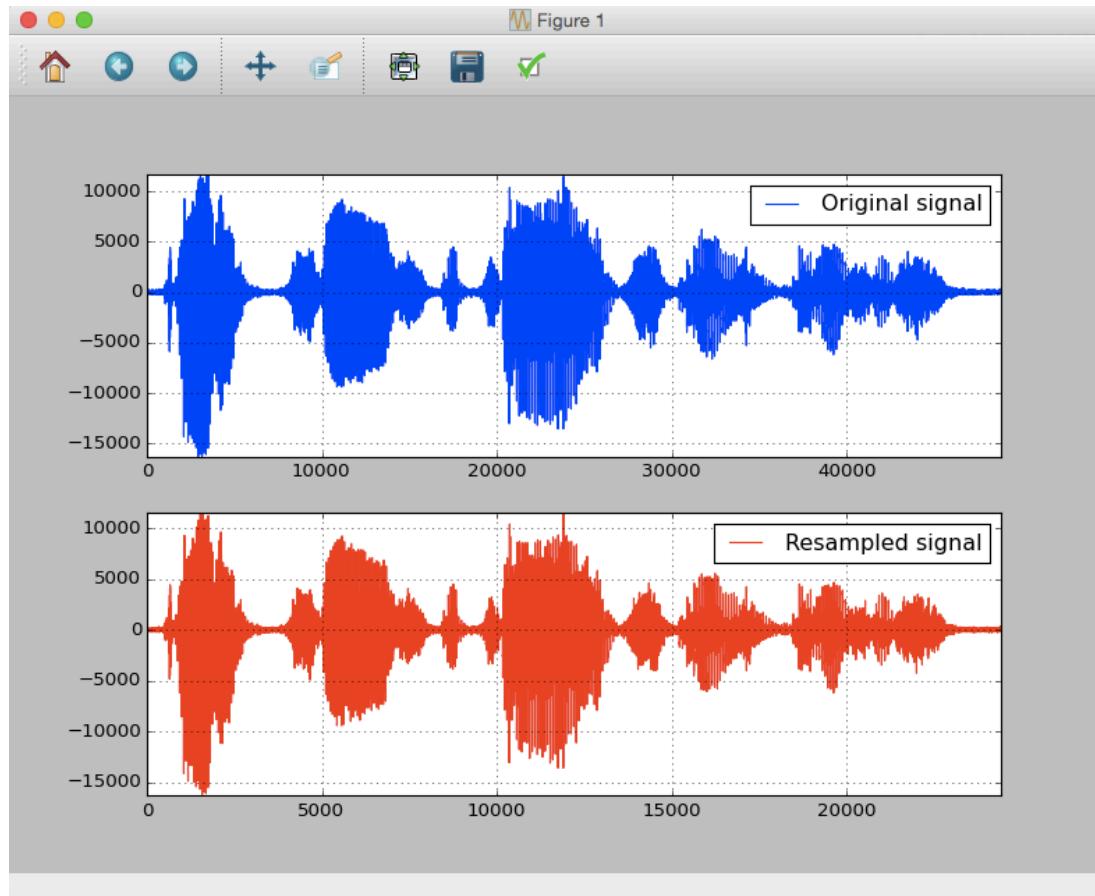
Example:

```
43 ## 3. Modifying sound:
44 from pylab import specgram, figure, plot, subplot, grid, axis, xlabel, ylabel, title, legend, pause
45 from scipy.io.wavfile import read, write
46 import scipy.signal.signaltools as sigtool
47 from numpy import linspace, pi, fft, abs, exp, hanning, zeros, angle, arange, sin, cos
48 import sounddevice as sd
49
50 # 3.1 Resampling:
51 (Fs, x) = read('files/lecture9/cheese.wav')
52 x2=x[0:len(x):2]      # we decrease the original audio samples in half
53 F = Fs/2              # we now have to decrease Fs by the same amount
54 F, s = F.as_integer_ratio() # a quick conversion from 'float' to 'int'
55 write('files/lecture9/cheese_resampled.wav',F,x2)
56
57 # Let's listen:
58 sd.play((x2),F), pause(3)
59 sd.play((x),Fs), pause(3)
60
61 """ Lets plot the two signals: """
62 figure(1)
63 subplot(2,1,1)
64 plot(x,label='Original signal')
65 axis('tight'); legend(loc='best')
66 grid(True)
67 subplot(2,1,2)
68 plot(x2,'r', label='Resampled signal')
69 axis('tight'); legend(loc='best')
70 grid(True)
71 pause(1)
72
73 """ Lets plot the spectrograms of two signals: """
74 figure(2)
75 subplot(2,1,1)
76 specgram(x), axis('tight'), title('Original signal');
77 ylabel('Frequency')
78 subplot(2,1,2)
79 specgram(x2), axis('tight'), title('Resampled (down-sampled) signal');
80 xlabel('Time'), ylabel('Frequency')
81 pause(1)
```

Signal Processing

- Signal Processing – sound processing: resampling

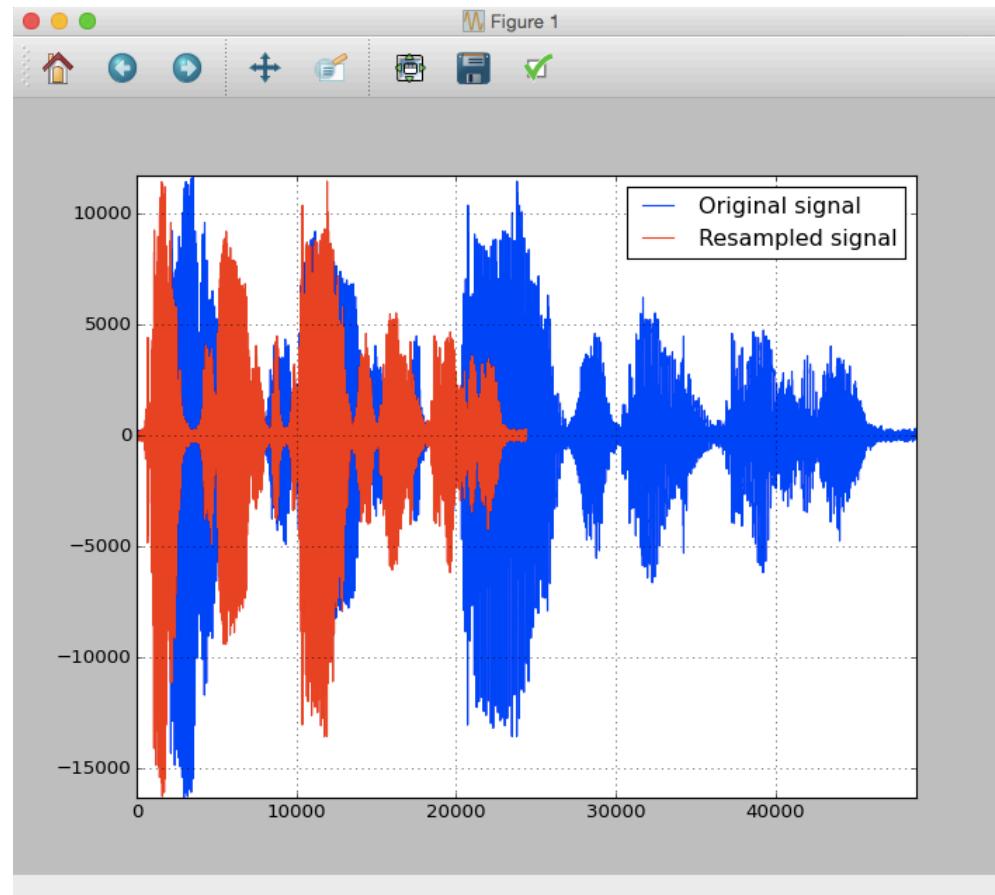
Example:



Signal Processing

- Signal Processing – sound processing: resampling

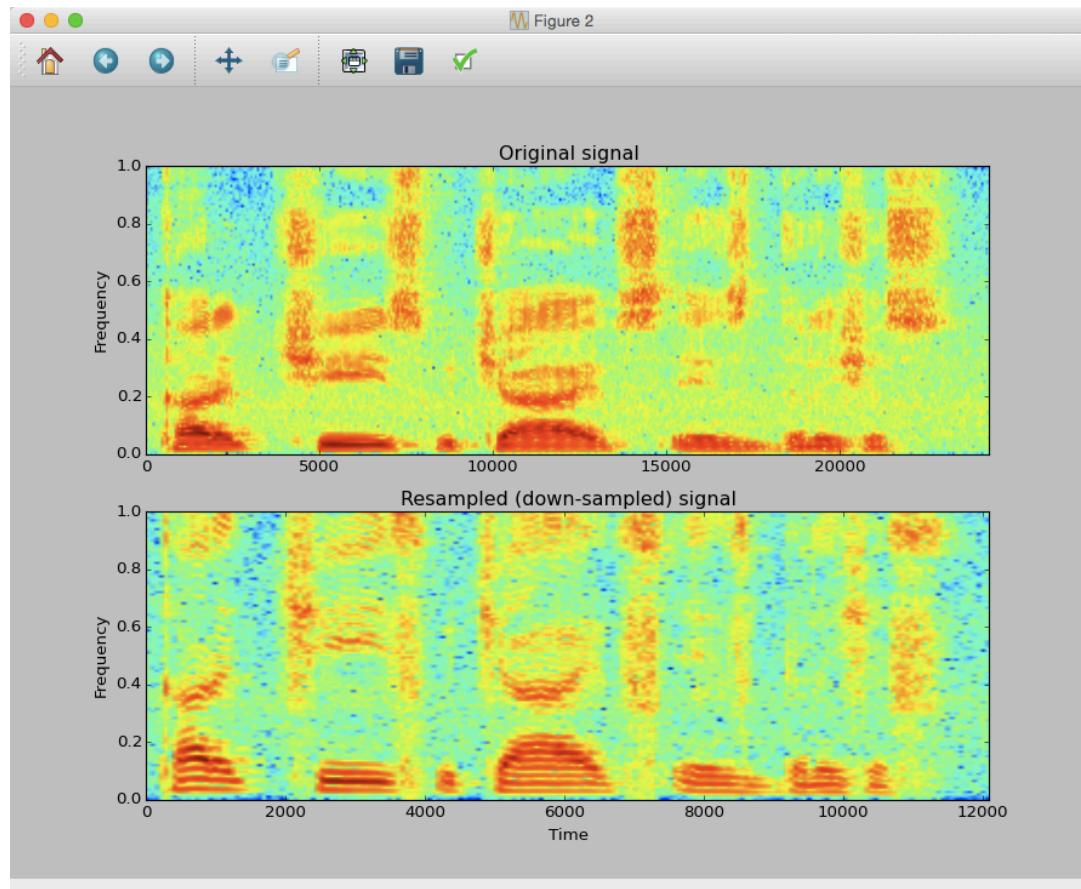
Example:



Signal Processing

- Signal Processing – sound processing: resampling

Example:



Signal Processing

- Signal Processing – sound processing: time domain manipulations

- discussion on speed playback change:

- what would happen if we **lower the number of samples**, but **don't change the sampling rate?**
 - what about when we **lower the sampling rate without changing the number of samples?**

```
76 # 3.2 Time domain manipulations:  
77 (Fs, x) = read('files/lecture9/cheese.wav')  
78 x2 = x[0:len(x):2] # we lower the original audio samples in half  
79 F = Fs/2           # we now lower Fs by the same amount  
80 F, s = F.as_integer_ratio() # conversion from 'float' to 'int'  
81  
82 # We use original Fs and shorter signal:  
83 write('files/lecture9/cheese_faster_pitch_up.wav',Fs,x2)  
84  
85 # We use lower Fs and original signal:  
86 write('files/lecture9/cheese_slower_pitch_down.wav',F,x)  
87  
88 # Now we read back the resampled file with lowered pitch:  
89 (F, x4) = read('files/lecture9/cheese_slower_pitch_down.wav')
```

Signal Processing

- Signal Processing – sound processing: frequency domain manipulations
 - Phase Vocoder (PV) – also known as **sound stretching**, is a method in which any given sound can be stretched or shrunk in time while **preserving its pitch**
 - defined in these terms, it follows that the **time stretching** needs a **manipulation** of the sound **in frequency domain**
 - Here is the procedure in a few simple steps:
 - we first need to split the main audio into **overlapping time blocks** (chunks)



- we rearrange these blocks so that:
 - if we want a shorter (**faster**) sound we **overlap** these blocks **more**



- we rearrange these blocks so that:
 - if we want to stretch the sound (**slower**) we **overlap** these blocks **less** (fig.b)



Signal Processing

- Signal Processing – sound processing: frequency domain manipulations

Example:

```
98 ## 3.3 Frequency domain manipulations (phase vocoder):
99 from pylab import figure, plot, subplot, grid, axis, xlabel, ylabel, title, legend, pause
100 from scipy.io.wavfile import read, write
101 from numpy import pi, fft, exp, hanning, zeros, arange, angle, sin, cos
102
103 (Fs, x) = read('files/lecture9/cheese.wav')
104
105 def modify(sound, c, win_size, d):
106
107     """ lets define the basic parameters: """
108     hann_win = hanning(win_size)
109     phase = zeros(win_size)
110     final_signal = zeros(int( len(sound) / c + win_size ))
111     out = zeros(win_size, dtype=complex)
112
113     for m in arange(0, len(sound)-(win_size + d), d*c):
114
115         """ we define the two neighbouring and overlapping audio parts: """
116         aud1 = sound[int(m):int(m)+win_size]
117         aud2 = sound[int(m)+d:int(m)+win_size+d]
118
119         """ we need to synchronize the second to the first window signal: """
120         sig1 = fft.fft(hann_win * aud1)
121         sig2 = fft.fft(hann_win * aud2)
122
123         phase += (angle(sig2) - angle(sig1))
124
125         """ wrap the phase back between pi and -pi: """
126         for k in phase:
127             if phase[int(k)] < -pi:
128                 phase[int(k)] = phase[int(k)] + 2*pi
129             elif phase[int(k)] >= pi:
130                 phase[int(k)] = phase[int(k)] - 2*pi
131
132         out.real, out.imag = cos(phase), sin(phase)
133
134         """ increment: """
135         win = int(m/c)
136
137         """ combine the newly calculated amplitude and phase parts: """
138         final_signal[win : win + win_size] += (fft.ifft(sig2.real*exp(1j*out.imag))).real
139
140     return final_signal.astype('int16')
```

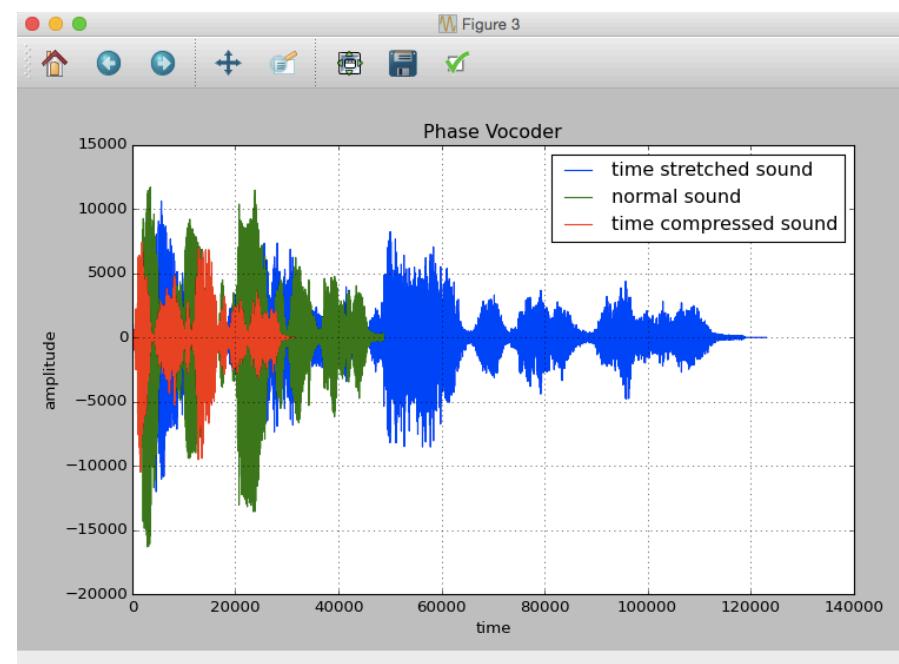
lets discuss
for a minute...

Signal Processing

- Signal Processing – sound processing: frequency domain manipulations

Example:

```
142 """ Let's test what we have done: """
143 win_size = 1024; overlap = int(win_size/2)
144 stretch_coef = 0.4; shrink_coef = 1.6
145
146 y = modify(x,stretch_coef,win_size,overlap)
147 write('files/lecture9/cheese_slower.wav',Fs,y)
148
149 y2 = modify(x,shrink_coef,win_size,overlap)
150 write('files/lecture9/cheese_faster.wav',Fs,y2)
151
152 # Now plot the results and compare:
153 figure(3)
154 plot(y, label='time stretched sound')
155 plot(x, label='normal sound')
156 plot(y2, label='time compressed sound')
157 legend(loc='best'); title('Phase Vocoder')
158 xlabel('time'); ylabel('amplitude'); grid(True)
159 pause(3)
```



Signal Processing

- Signal Processing – sound processing: pitch shift
 - recall the talk about changing the sampling rate without changing the number of samples and vs. versa
 - it resulted in time stretch with a pitch shift
 - but what if we want to have the original length of the speech with a pitch shift?
 - .. we can do it by:
 - pulling the sound as we just did, while not changing the pitch
 - then factoring the sampling rate by the amount of pull or stretch
 - then finally we resample with the new rate

Example:

- in theory when doubling the frequency of any sound the pitch of that sound increases by one octave (or 12 semitones)
- so to increase the pitch by s semitones we multiply the frequency by a factor of $2^{\frac{s}{12}}$

Signal Processing

- Signal Processing – sound processing: pitch shift

Example:

- to simplify the problem we first stretch the sound by 100%
(changes in Frequency domain)
- then we decrease the number of samples by the same amount and sample with same Fs
(changes in Time domain)
- Here is the code:

```
161 ## 3.4 Time-Frequeincy changes / pitch shift:  
162 (Fs, x1) = read('files/lecture9/cheese.wav')  
163 win_size = 1024; overlap = int(win_size/2)  
164 stretch_coef = 0.5  
165  
166 # Time stretch in frequency domain:  
167 y = modify(x,stretch_coef,win_size,overlap)  
168  
169 # Pitch change in time domain:  
170 """ we lower the original audio samples by the 'stretch_coef' """  
171 x2 = y[0:len(y):int(1/stretch_coef)]  
172 write('files/lecture9/cheese_normalspeed_pitch_up.wav',Fs,x2)
```

Signal Processing

- Signal Processing – sound processing: **filter design**
 - Using Scipy we can easily create **digital or analog filters** of types: **LPF, HPF, BPF or BSF**.
 - In Scipy there are many categories one can choose a filter method from
 - For example in the IIR section, a Matlab-like available filters are: **Chebyshev, Butterworth, Elliptic, Bessel**, etc.
 - **Butterworth** filter has a very **flat frequency response** in the passband so we choose it for our next example
 - Here is the syntax for a Butterworth filter:

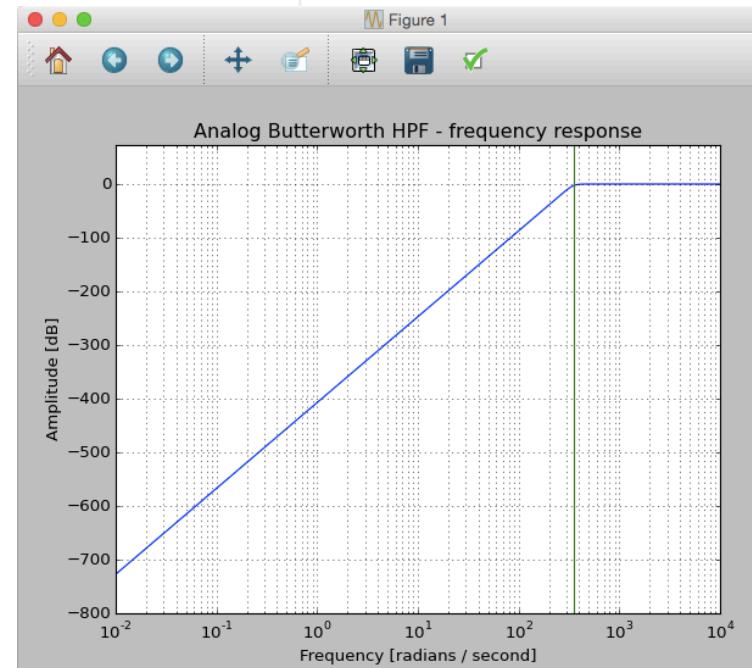
```
scipy.signal.butter(N, Wn, btype='low', analog=False, output='ba')
```

, where:
 - N** – is the filter's order (**int**)
 - Wn** – is giving the critical frequencies, or the point at which the gain drops to $1/\sqrt{2}$ that of the passband (the “-3 dB point”). For digital filters, **Wn** is normalized from 0 to 1, where 1 is the Nyquist frequency, pi radians/sample. (**scalar**)
 - btype** – is the type of filter to be designed. Choices are: {‘lowpass’, ‘highpass’, ‘bandpass’, ‘bandstop’}, the default is ‘lowpass’
 - analog** – (**bool**), optional parameter. True = analog filter, False = digital filter design
 - output** – {‘ba’, ‘zpk’, ‘sos’}, optional parameter. The default is ‘ba’. Options are:
 1. (‘ba’) – numerator (**b**) and denominator (**a**) polynomials of the IIR filter
 2. (‘zpk’) – zero-pole and system gain of the IIR filter transfer function
 3. (‘sos’) – second-order sections representation of the IIR filter

Signal Processing

- Signal Processing – sound processing: filter design
 - Example using an Analog Butterworth HP filter:

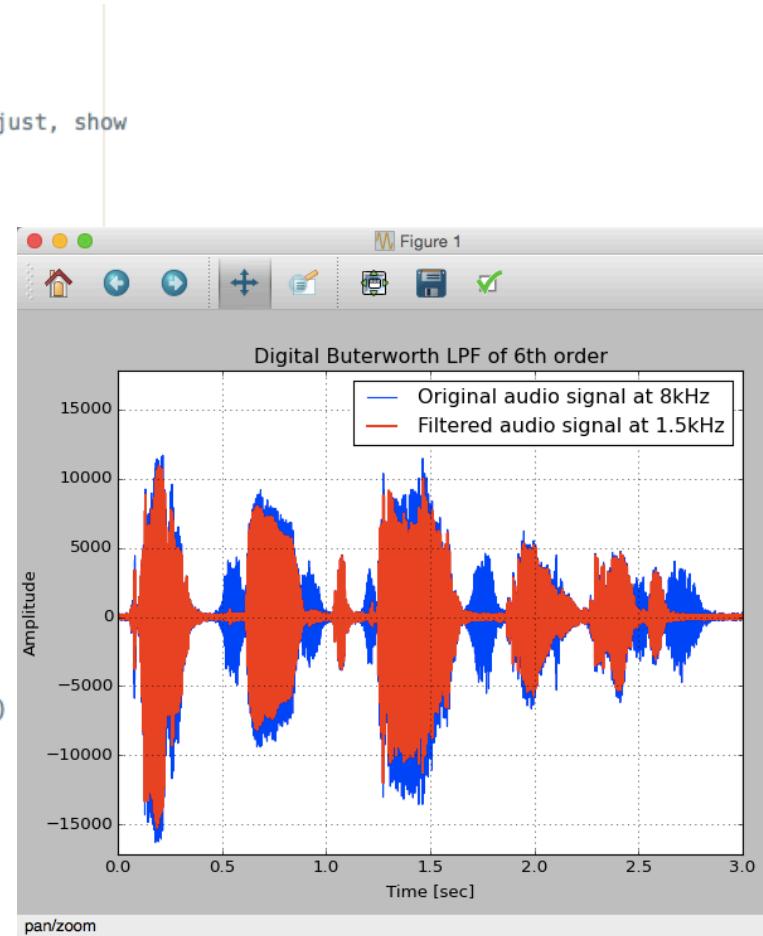
```
174 ## 3.5 Filtering:  
175 # Butterworth example of Analog HPF:  
176 from scipy import signal, log10  
177 from pylab import semilogx, title, xlabel, ylabel, margins, grid, axvline, pause  
178  
179 b, a = signal.butter(8, 350, 'high', analog=True, output='ba')  
180 w, h = signal.freqs(b, a) # Compute frequency response of analog filter with 'b'-num, 'a' -den  
181 semilogx(w, 20 * log10(abs(h)))  
182 title('Analog Butterworth HPF - frequency response')  
183 xlabel('Frequency [radians / second]')  
184 ylabel('Amplitude [dB]')  
185 margins(0, 0.1)  
186 grid(which='both', axis='both')  
187 axvline(350, color='green') # Show a cutoff frequency line  
188 pause(1)
```



Signal Processing

- Signal Processing – sound processing: filter design
 - Example using a Digital Butterworth LP filter:

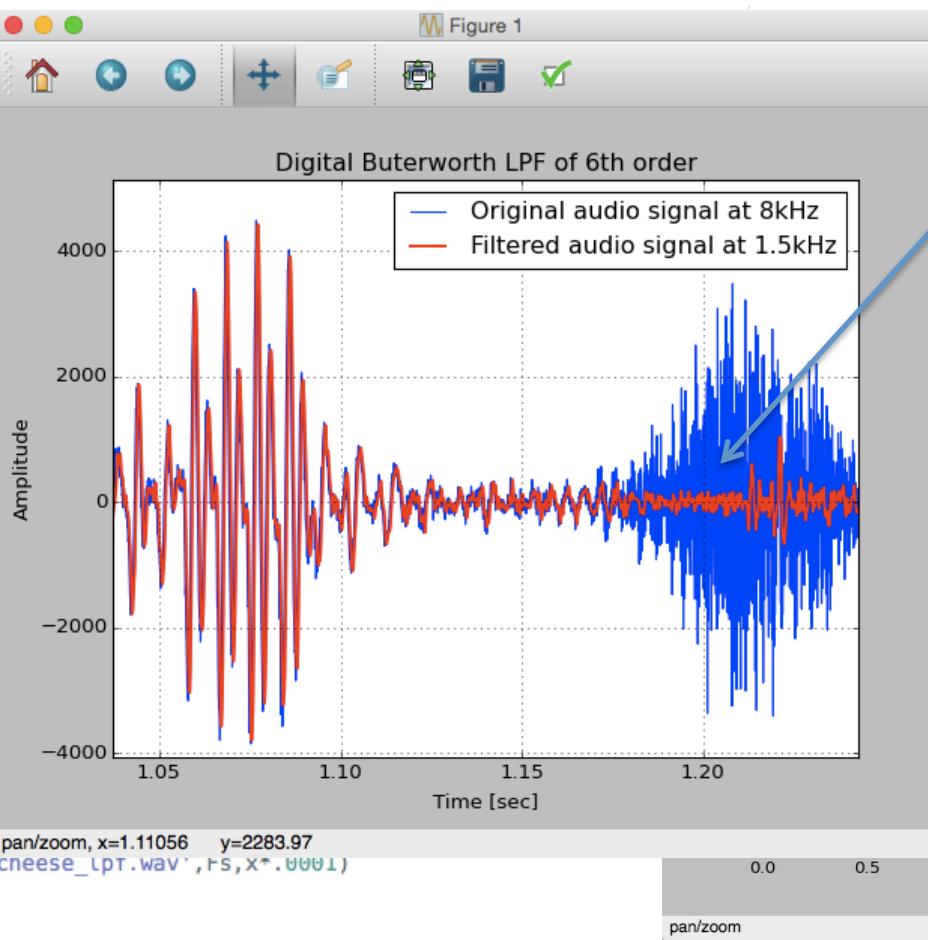
```
182 # Real example of a Digital LP filter:
183 from scipy.io.wavfile import read, write
184 from numpy import int, linspace
185 from scipy.signal import butter, lfilter, freqz
186 from pylab import plot, title, xlabel, ylabel, grid, legend, subplots_adjust, show
187
188 def butterworth_lpf(data, cutoff, fs, order):
189     nyquist = fs / 2
190     cutoff_freq = cutoff / nyquist
191     b, a = butter(order, cutoff_freq, btype='low', analog=False)
192     x = lfilter(b, a, data)
193     return x
194
195 # We load an audio file to be filtered:
196 (Fs, data) = read('files/lecture10/cheese.wav')
197 T = int(len(data)) / Fs # the audio length in seconds
198 t = linspace(0, T, len(data), endpoint=False)
199
200 # We specify filter requirements:
201 order = 6
202 cutoff = 1500 # we set the cutoff frequency of the filter, [Hz]
203
204 # Filter the audio signal:
205 x = butterworth_lpf(data, cutoff, Fs, order)
206
207 # Plot both the original and filtered signals:
208 plot(t, data, 'b-', label='Original audio signal at 8kHz')
209 plot(t, x, 'r-', linewidth=1.75, label='Filtered audio signal at 1.5kHz')
210 title('Digital Butterworth LPF of 6th order')
211 xlabel('Time [sec]'), ylabel('Amplitude'), grid(), legend()
212 show()
213
214 # Save the filtered and normalized audio:
215 write('files/lecture10/cheese_lpf.wav', Fs, x*.0001)
```



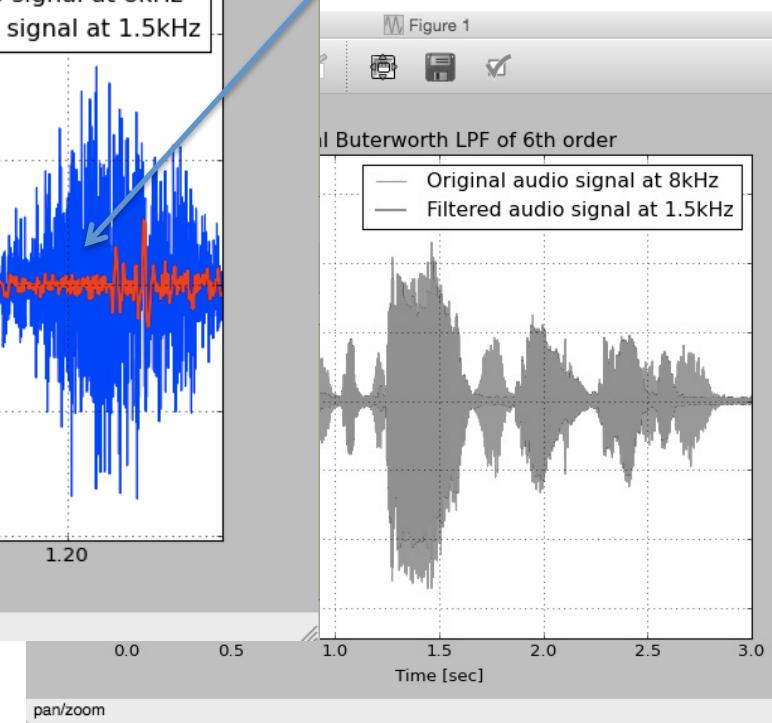
Signal Processing

- Signal Processing – sound processing: filter design
 - Example using a Digital Butterworth LP filter:

```
182 # Real example of a Dig
183 from scipy.io.wavfile i
184 from numpy import int,
185 from scipy.signal import
186 from pylab import plot,
187
188 def butterworth_lpf(data
189     nyquist = fs / 2
190     cutoff_freq = cutoff
191     b, a = butter(order,
192     x = lfilter(b, a, d
193     return x
194
195 # We load an audio file
196 (Fs, data) = read('file
197 T = int(len(data)) / Fs
198 t = linspace(0, T, len(
199
200 # We specify filter req
201 order = 6
202 cutoff = 1500 # we set
203
204 # Filter the audio sign
205 x = butterworth_lpf(dat
206
207 # Plot both the original
208 plot(t, data, 'b-', lab
209 plot(t, x, 'r-', linewidth
210 title('Digital Butterwo
211 xlabel('Time [sec]'), ylabel('Amplitude')
212 show()
213
214 # Save the filtered and
215 write('files/lecture10/cneese_lpt.wav', fs, x+.0001)
```



Observe all frequencies above 1.5kHz are filtered out – red signal is lower at the noisy part to the right



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

- **Scipy 2/2**
 - Working with databases
 - Interpolation
 - Statistics
 - Optimization
 - Multithreading and Multiprocessing

- **Project**
 - Project Presentation

Final Project

Python and MySQL Database

- the Python standard for database interfaces is the Python DB-API
- Python Database API supports a wide range of database servers such as:
 - GadFly
 - mSQL
 - MySQL
 - PostgreSQL
 - Microsoft SQL Server 2000
 - Informix
 - Interbase
 - Oracle
 - Sybase
- if you need to access an Oracle database as well as a MySQL database, you must download both the Oracle and the MySQL database modules

Python and MySQL Database

- the DB API provides a minimal standard for working with databases using Python structures and syntax
- This API includes the following:
 - Importing the API module
 - Acquiring a connection with the database
 - Issuing SQL statements and stored procedures
 - Closing the connection
- install MySQL connector using homebrew: `brew install mysql-connector-c`
- PyMySQL and MySQLdb are **both database connectors for Python**

Python and MySQL Database

- PyMySQL and MySQLdb provide the same functionality
- MySQLdb, is a C extension module that may be difficult to compile (MAC)
- The difference is in the implementation where MySQLdb is a C extension and PyMySQL is pure Python
- PyMySQL might be easier to get running on some systems
- PyMySQL is pure python version of MySQLdb (MySQL-Python) package:
 - pymysql can be installed on any system without needing a C compiler
 - Installing mysqldb may need a compiler and in windows can produce error ('Unable to find vcvarsall.bat')

Python and MySQL Database

- MySQLdb example:
- To continue you need to first:
 - You have **created a database TESTDB**
 - You have **created a table EMPLOYEE** in TESTDB
 - This **table has fields FIRST_NAME, LAST_NAME, AGE, SEX and INCOME**
 - **User ID "testuser" and password "test123"** are set to access TESTDB
 - Python module **MySQLdb is installed properly** on your machine

Python and MySQL Database

- MySQLdb example – connecting:

```
1 #!/usr/bin/python
2
3 import pymysql
4
5 ## Following is the example of connecting with MySQL database TESTDB:
6
7 # Open database connection
8 db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )
9
10 # prepare a cursor object using cursor() method
11 cursor = db.cursor()
12
13 # execute SQL query using execute() method.
14 cursor.execute("SELECT VERSION()")
15
16 # Fetch a single row using fetchone() method.
17 data = cursor.fetchone()
18
19 print "Database version : %s" % data          >>> Database version : 5.0.45
20
21 # disconnect from server
22 db.close()
```

Python and MySQL Database

- MySQLdb example – creating table:

```
24 ## We create the database table EMPLOYEE:  
25  
26 # Open database connection  
27 db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )  
28  
29 # prepare a cursor object using cursor() method  
30 cursor = db.cursor()  
31  
32 # Drop table if it already exist using execute() method.  
33 cursor.execute("DROP TABLE IF EXISTS EMPLOYEE")  
34  
35 # Create table as per requirement  
36 sql = """CREATE TABLE EMPLOYEE (  
37     FIRST_NAME  CHAR(20) NOT NULL,  
38     LAST_NAME   CHAR(20),  
39     AGE INT,  
40     SEX CHAR(1),  
41     INCOME FLOAT )"""  
42  
43 cursor.execute(sql)  
44  
45 # disconnect from server  
46 db.close()
```

Python and MySQL Database

- MySQLdb example – insert into table:

```
48 ## Executes SQL INSERT statement to create a record into EMPLOYEE table:
49
50 # Open database connection
51 db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )
52
53 # prepare a cursor object using cursor() method
54 cursor = db.cursor()
55
56 # Prepare SQL query to INSERT a record into the database.
57 sql = """INSERT INTO EMPLOYEE(FIRST_NAME,
58         LAST_NAME, AGE, SEX, INCOME)
59         VALUES ('Mac', 'Mohan', 20, 'M', 2000)"""
60 try:
61     # Execute the SQL command
62     cursor.execute(sql)
63     # Commit your changes in the database
64     db.commit()
65 except:
66     # Rollback in case there is any error
67     db.rollback()
68
69 # disconnect from server
70 db.close()
```

Python and MySQL Database

- MySQLdb example – insert into table:

```
72 ## The previous example can be written dynamically:  
73  
74 # Open database connection  
75 db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )  
76  
77 # prepare a cursor object using cursor() method  
78 cursor = db.cursor()  
79  
80 # Prepare SQL query to INSERT a record into the database.  
81 sql = "INSERT INTO EMPLOYEE(FIRST_NAME, ...\n        LAST_NAME, AGE, SEX, INCOME) \\\n        VALUES ('%s', '%s', '%d', '%c', '%d' )...%s...\n        ('Mac', 'Mohan', 20, 'M', 2000)  
82  
83 try:  
84     # Execute the SQL command  
85     cursor.execute(sql)  
86     # Commit your changes in the database  
87     db.commit()  
88 except:  
89     # Rollback in case there is any error  
90     db.rollback()  
91  
92 # disconnect from server  
93  
94 db.close()  
95  
96 # Result to screen:  
97 # .....  
98 # user_id = "test123"  
99 # password = "password"  
100 # .....  
101 # con.execute('insert into Login values("%s", "%s")' % \  
102 #           (user_id, password))  
103 # .....
```

Python and MySQL Database

- MySQLdb example – read from table:
 - READ Operation on any database means to fetch some useful information from the database.
 - Once our database connection is established, you are ready to make a query into this database. You can use either **fetchone()** method to fetch single record or **fetchall()** method to fetch multiple values from a database table.
 - **fetchone()**: It **fetches the next row** of a query result set. A result set is an object that is returned when a cursor object is used to query a table.
 - **fetchall()**: It **fetches all the rows** in a result set. If some rows have already been extracted from the result set, then it retrieves the remaining rows from the result set.
 - **rowcount**: This is a read-only attribute and **returns the number of rows** that were affected by an execute() method.

Python and MySQL Database

- MySQLdb example – read from table:

```
106 ## Using the READ operation:  
107 # Open database connection  
108 db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )  
109  
110 # prepare a cursor object using cursor() method  
111 cursor = db.cursor()  
112  
113 sql = "SELECT * FROM EMPLOYEE \\\n114      WHERE INCOME > '%d'" % (1000)  
115 try:  
116     # Execute the SQL command  
117     cursor.execute(sql)  
118     # Fetch all the rows in a list of lists.  
119     results = cursor.fetchall()  
120     for row in results:  
121         fname = row[0]          135 # This will produce the following result :  
122         lname = row[1]          136 # fname=Mac, lname=Mohan, age=20, sex=M, income=2000  
123         age = row[2]  
124         sex = row[3]  
125         income = row[4]  
126         # Now print fetched result  
127         print "fname=%s,lname=%s,age=%d,sex=%s,income=%d" % \  
128             (fname, lname, age, sex, income )  
129 except:  
130     print "Error: unable to fetch data"  
131  
132 # disconnect from server  
133 db.close()
```

Python and MySQL Database

- MySQLdb example – update a record in table:

```
138 ## UPDATE Operation for one or more records, which are already available in the database:  
139  
140 # Open database connection  
141 db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )  
142  
143 # prepare a cursor object using cursor() method  
144 cursor = db.cursor()  
145  
146 # Prepare SQL query to UPDATE required records  
147 sql = "UPDATE EMPLOYEE SET AGE = AGE + 1  
148             WHERE SEX = '%c'...%_.('M').)  
149 try:  
150     # Execute the SQL command  
151     cursor.execute(sql)  
152     # Commit your changes in the database  
153     db.commit()  
154 except:  
155     # Rollback in case there is any error  
156     db.rollback()  
157  
158 # disconnect from server  
159 db.close()
```

Python and MySQL Database

- MySQLdb example – delete a record in table:

```
161 ## DELETE some records from your database:  
162  
163 # Open database connection  
164 db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )  
165  
166 # prepare a cursor object using cursor() method  
167 cursor = db.cursor()  
168  
169 # Prepare SQL query to DELETE required records  
170 sql = "DELETE FROM EMPLOYEE WHERE AGE > '%d'" % (20)  
171 try:  
172     # Execute the SQL command  
173     cursor.execute(sql)  
174     # Commit your changes in the database  
175     db.commit()  
176 except:  
177     # Rollback in case there is any error  
178     db.rollback()  
179  
180 # disconnect from server  
181 db.close()
```

Python and MySQL Database

- MySQLdb example – some operations:
 - `db.commit()` – commit is the operation, which gives a green signal to database to **finalize the changes**, and after this operation, no change can be reverted back
 - `db.rollback()` – If you are not satisfied with one or more of the changes and you want to **revert back those changes** completely, then use **rollback()** method
 - `db.close()` – to **disconnect** Database connection

Python and MySQL Database

- PyMySQL example – install and create database:

- Install your PyMySQL package:

- >>> pip install pymysql

- Lets create our database:

```
185 # Lets create our database:  
186 mysql -u user -pPassword -e "CREATE DATABASE test"
```

Python and MySQL Database

- PyMySQL example - create a table:

```
188 #!/usr/bin/env python3
189
190 #Importing needed modules of PyMySQL
191 from pymysql import connect, err, sys, cursors
192
193 #Doing our connection
194 conn = connect( host = 'localhost',
195                  port = 3306,
196                  user = 'user',
197                  passwd = 'Password',
198                  db = 'test' );
199 #Setting cursors and defining type of returns we need from Database, here it's gonna be returning as dictionary data
200 cursor = conn.cursor( cursors.DictCursor );
201
202 #Running a query easily
203 cursor.execute( """CREATE TABLE IF NOT EXISTS `test_user`
204 (
205     `ID` INT AUTO_INCREMENT NOT NULL,
206     `user_name` VARCHAR( 200 ) NOT NULL,
207     `user_email` VARCHAR( 200 ) NOT NULL,
208     PRIMARY KEY ( `ID` )
209     )Engine = MyISAM""")
210 cursor.execute( "INSERT INTO `test_user` ( `user_name`, `user_email` ) ...")
```

Python and MySQL Database

- PyMySQL example - return data from table:

```
212 ## Return data from table:  
213 cursor.execute( "SELECT * FROM `test_user`" )  
214 data = cursor.fetchall()  
215 print( data )  
  
217 # This will be returned:  
218 #     [{ID': 1, 'user_email': 'donjajo@example.com', 'user_name': 'donjajo'}]
```

Interpolation

- Interpolation
 - `scipy.interpolate` is the Scipy's package that contains interpolation functionality
 - it is useful for fitting a function as close as possible to a given set of points obtained from experiments
 - this gives us the tools to recreate given signal between specific evaluating points where no measurement samples are available
 - the `scipy.interpolate` module is based on the FITPACK Fortran subroutines from the netlib project
 - there are few main `scipy.interpolate` sub-packages the contain:
 - `Univariate` interpolation – one-dimensional interpolation
 - `Multivariate` interpolation – multi-dimensional interpolation
 - `1D Splines` – one-dimensional interpolating splines for given set of data points
 - `2D Splines` – bivariate spline approximations over given set of data points
 - `Lagrange` and `Taylor` polynomial interpolators

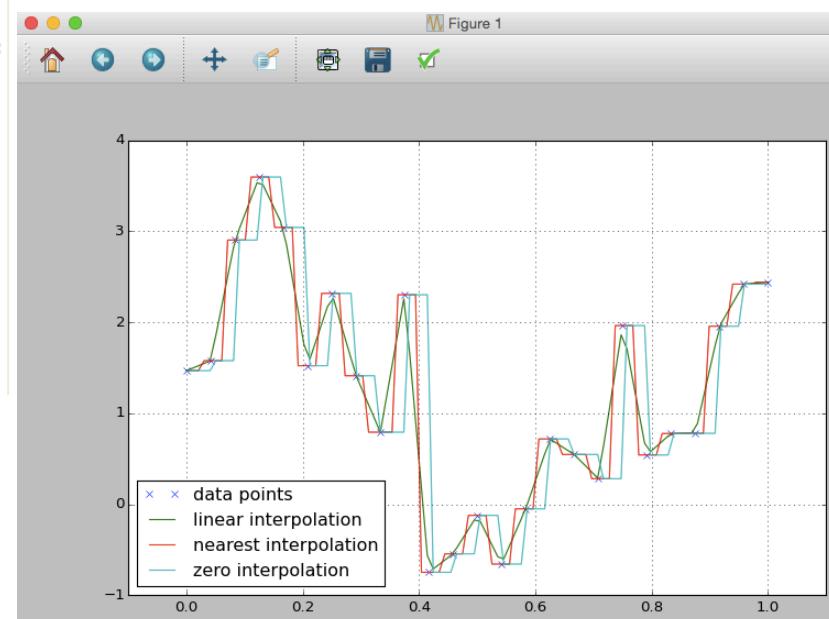
Interpolation

- Interpolation
 - we can examine several interpolation functions specified in the `kind` property of `scipy.interpolate.interp1d`:
 - `linear` – interpolation of first order
 - `nearest` – returns the value closest to the point of interpolation
 - `zero` – levels after given data point, similar to quantization (square-like)
 - `slinear` – spline interpolation of first order
 - `quadratic` – spline interpolation of second order
 - `cubic` – spline interpolation of third order
 - `linear` is always set by default if nothing is specified
 - the difference between `scipy.interpolate.interp1d` and `scipy.interpolate.interp2d` is that the later is used for `2-D arrays` and the former `for 1-D arrays`
 - we should note that the `computed interpolation time must stay within the measured time range`

Interpolation

- Interpolation – Example using `scipy.interpolate.interp1d`:

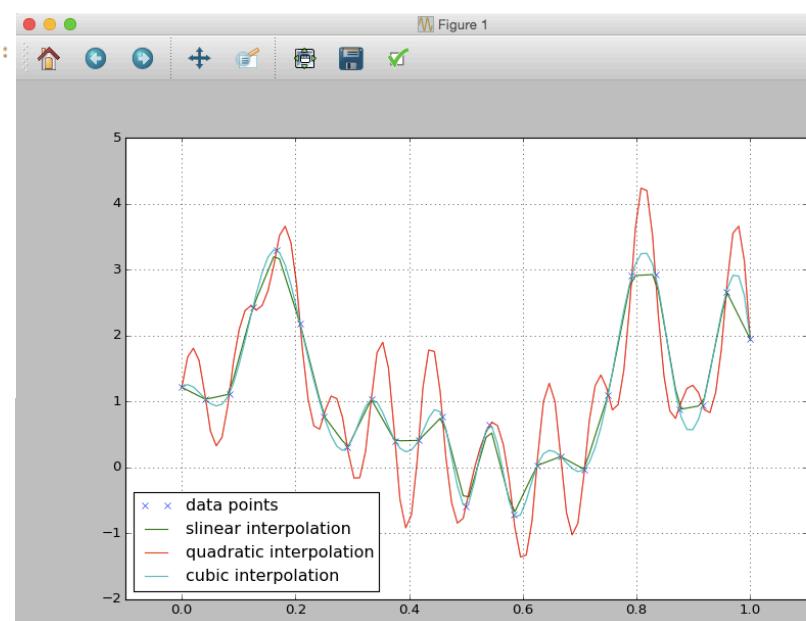
```
1 ## Scipy 2/3
2
3 # Interpolation example:
4 from numpy import linspace, random, cos, pi
5 from scipy.interpolate import interp1d
6 import pylab as plt
7
8 # We first create the set of data points over which to interpolate:
9 data_span = linspace(0, 1, 25)
10 noise = (random.random(25)*3 + 0.5) / 2
11 data_points = cos(2 * pi * data_span) + noise**2
12 t_calculated = linspace(0, 1, 100)
13
14 # We build the 'linear' interpolation function like this:
15 linear_i = interp1d(data_span, data_points) # no 'kind' needed as 'linear' is default
16 linear = linear_i(t_calculated)
17
18 # This is how a 'nearest' interpolation is obtained - using the 'kind' property:
19 near_i = interp1d(data_span, data_points, kind='nearest')
20 near = near_i(t_calculated)
21
22 # This is how a 'zero' interpolation is obtained - using the 'kind' property:
23 zero_i = interp1d(data_span, data_points, kind='zero')
24 zero = zero_i(t_calculated)
25
26 # Now lets plot them all:
27 plt.plot(data_span, data_points, 'x', ms=5, label='data points')
28 plt.plot(t_calculated, linear, label='linear interpolation')
29 plt.plot(t_calculated, near, label='nearest interpolation')
30 plt.plot(t_calculated, zero, label='zero interpolation')
31
32 plt.legend(loc='lower left')
33 plt.grid(True), plt.xlim(-0.1,1.1)
```



Interpolation

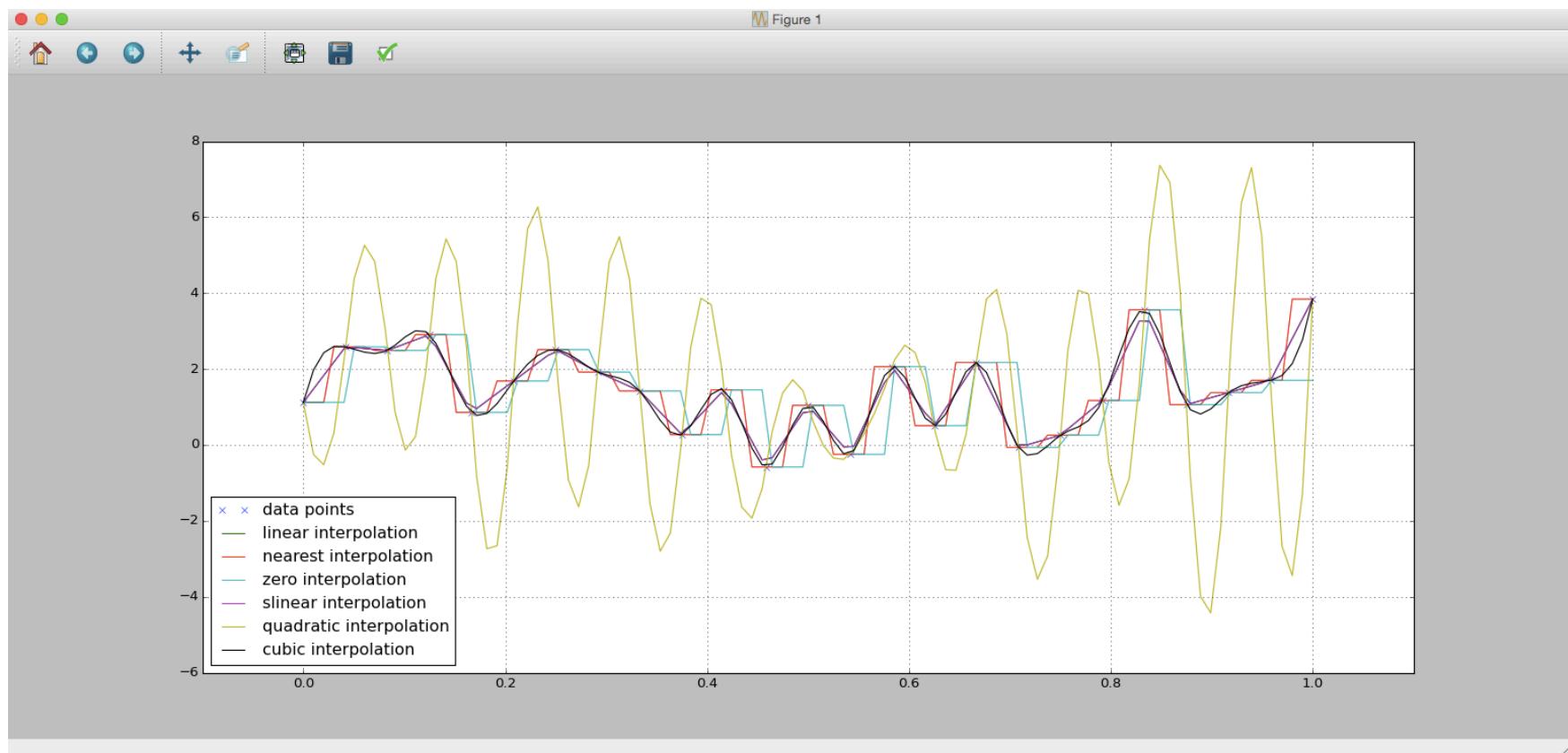
- Interpolation – Example using `scipy.interpolate.interp1d`:

```
1 ## Scipy 2/3
2
3 # Interpolation example:
4 from numpy import linspace, random, cos, pi
5 from scipy.interpolate import interp1d
6 import pylab as plt
7
8 # We first create the set of data points over which to interpolate:
9 data_span = linspace(0, 1, 25)
10 noise = (random.random(25)*3 + 0.5) / 2
11 data_points = cos(2 * pi * data_span) + noise**2
12 t_calculated = linspace(0, 1, 100)
13
14 # This is how a 'slinear' interpolation is obtained - using the 'kind' property:
15 slinear_i = interp1d(data_span, data_points, kind='slinear')
16 slinear = slinear_i(t_calculated)
17
18 # This is how a 'quadratic' interpolation is obtained - using the 'kind' property:
19 quadratic_i = interp1d(data_span, data_points, kind='quadratic')
20 quadratic = quadratic_i(t_calculated)
21
22 # This is how a 'cubic' interpolation is obtained - using the 'kind' property:
23 cubic_i = interp1d(data_span, data_points, kind='cubic')
24 cubic = cubic_i(t_calculated)
25
26 # Now lets plot them all:
27 plt.plot(data_span, data_points, 'x', ms=5, label='data points')
28 plt.plot(t_calculated, slinear, label='slinear interpolation')
29 plt.plot(t_calculated, quadratic, label='quadratic interpolation')
30 plt.plot(t_calculated, cubic, label='cubic interpolation')
31
32 plt.legend(loc='lower left')
33 plt.grid(True), plt.xlim(-0.1,1.1)
```



Interpolation

- Interpolation – Example using `scipy.interpolate.interp1d`:

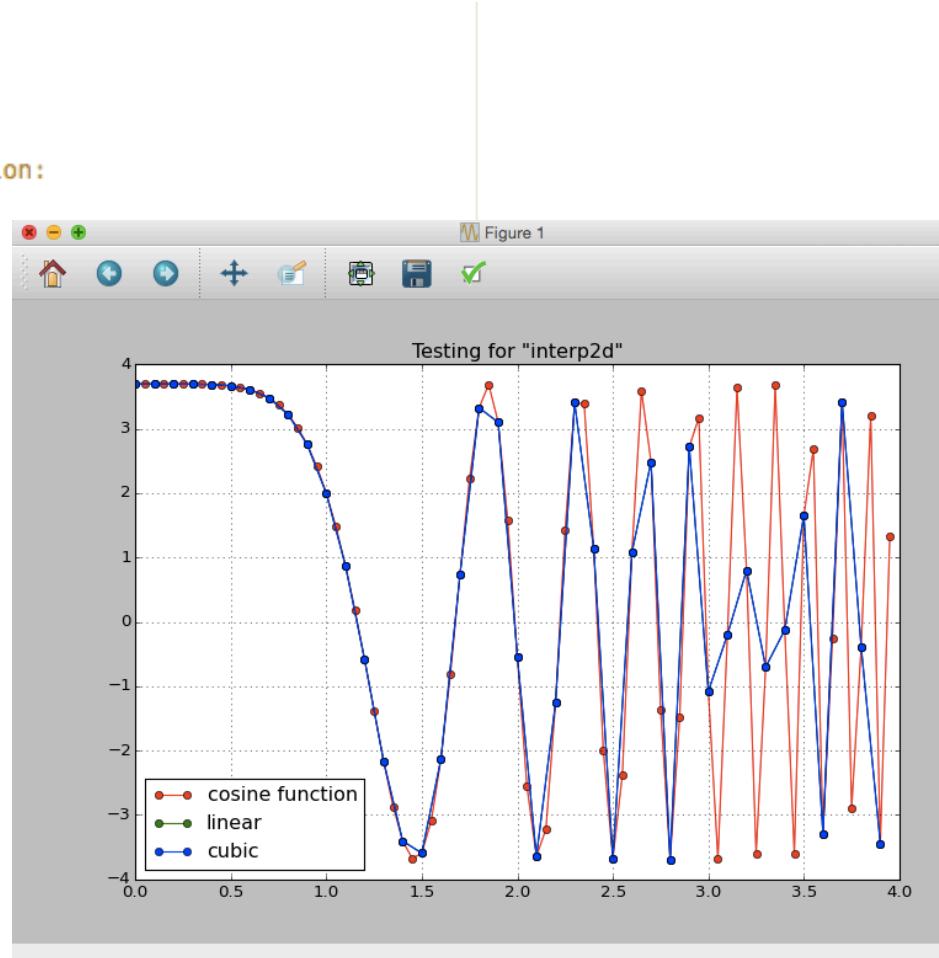


Interpolation

- Interpolation – Example using `scipy.interpolate.interp2d`:

```
99 # Testing the 'interp2d' interpolation:  
100 from numpy import arange, meshgrid, cos, exp  
101 from scipy.interpolate import interp2d  
102 from pylab import plot, grid, title, legend  
103  
104 # Construct the vectors with data for interpolation:  
105 a,b = arange(0, 4, 5e-2), arange(0, 4, 5e-2)  
106 at,bt = meshgrid(a, b)  
107 c = cos(at**3+bt**3)/2*exp(2)  
108  
109 # Lets test for the only two kinds accepted:  
110 m = interp2d(a, b, c, kind='linear')  
111 n = interp2d(a, b, c, kind='cubic')  
112  
113 # Plot the results and compare:  
114 an,bn = arange(0, 4, 1e-1), arange(0, 4, 1e-1)  
115 cn = n(an, bn); cm = m(an, bn)  
116 plot(an, c[0, :], 'ro-', label='cosine function')  
117 plot(an, cm[0, :], 'go-', label='linear')  
118 plot(an, cn[0, :], 'bo-', label='cubic')  
119 title('Testing for "interp2d"'), grid(True)  
120 legend(loc='lower left')
```

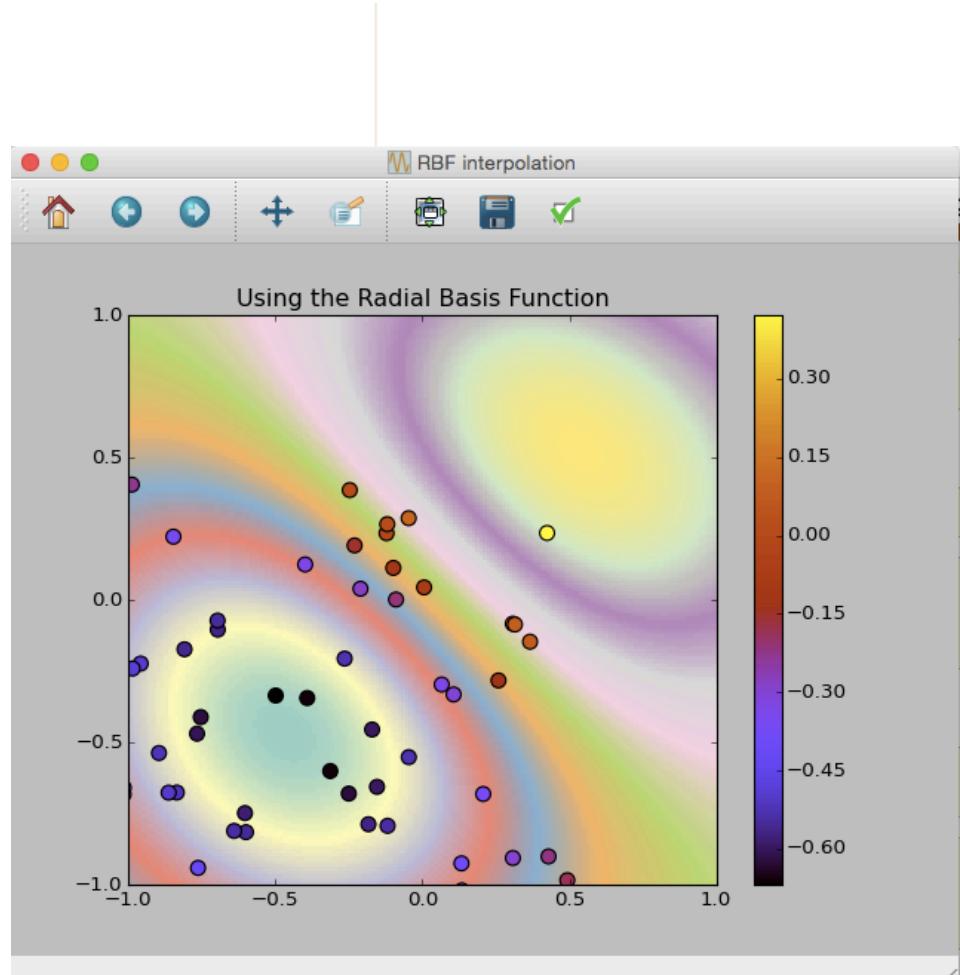
- note that the two kinds of interpolations allowed in `interp2d` overlap since they perform in a similar fashion in this example



Interpolation

- Interpolation – Example using `scipy.interpolate.Rbf`:

```
51 # 2-D interpolation example:  
52 from numpy import random, exp, linspace, meshgrid  
53 from scipy.interpolate import Rbf  
54 import pylab as plt  
55  
56 # Create the scattered data:  
57 pts = 80  
58 a,b = random.rand(pts)*2-1.5, random.rand(pts)*2-1.5  
59 c = (a+b-.125)*exp(-1*(a**2+b**2))  
60 t = linspace(-1.0, 1.0, pts*2)  
61 ai, bi = meshgrid(t, t)  
62  
63 # Using the RBF function:  
64 r = Rbf(a, b, c, epsilon=3)  
65 ci = r(ai, bi)  
66  
67 # Now we plot the result:  
68 plt.figure('RBF interpolation')  
69 plt.title('Using the Radial Basis Function')  
70 plt.pcolor(ai, bi, ci, cmap='Set3')  
71 plt.scatter(a, b, pts, c, cmap='gnuplot')  
72 plt.xlim(-1, 1), plt.ylim(-1, 1)  
73 plt.colorbar()
```



Statistics

- Statistics
 - Scipy is rich in [statistical functions](#) and [probability distributions](#) of random processes, provided under the [scipy.stats](#) package
 - [a rich and very lengthy description of this package](#) can be found using [help\(stats\)](#)
 - there are many methods that need to be examined thoroughly depending on your needs
 - some of the common [scipy.stats](#) methods are:
 - Random Variables (RV)
 - Shifting and Scaling
 - [Fitting Distributions](#)
 - Shape Parameters
 - Broadcasting
 - T-test
 - Kolmogorov-Smirnov test
 - Descriptive Statistics
 - Tails of a distribution
 - Comparing means
 - Density estimation – Univariate and Multivariate

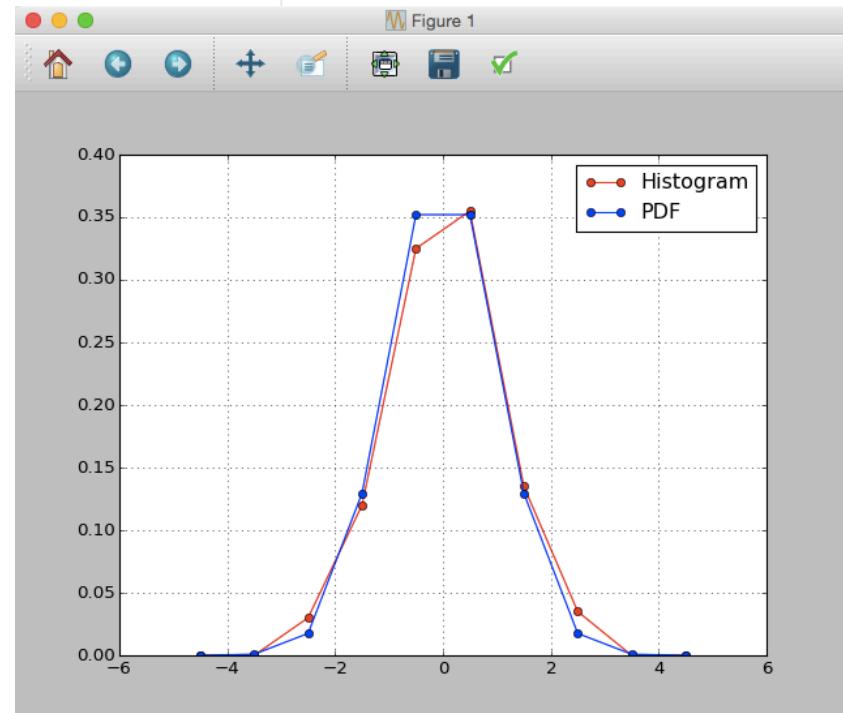
Statistics

- Statistics – Random Variables (RV)
 - Random Variables are represented in two main classes:
 - continuous random variables
 - discrete random variables
 - there are over 80 continuous random variables and 10 discrete random variables
 - common Random Variables methods are:
 - pdf: [Probability Density Function \(PDF\)](#) the likelihood of a random variable to take a given value
 - cdf: [Cumulative Distribution Function \(CDF\)](#) the probability that a real-valued rand. fun. $X \leq x$
(distribution function X is evaluated at x)
 - rvs: [Random Variates](#) are frequently used as the input to simulation models
 - sf: Survival Function (1-CDF)
 - ppf: Percent Point Function (Inverse of CDF)
 - isf: Inverse Survival Function (Inverse of SF)
 - stats: Return mean, variance, (Fisher's) skew, or (Fisher's) kurtosis
 - moment: non-central moments of the distribution

Statistics

- Statistics – Random Variables (RV) – PDF and histogram
 - we use [histograms](#) when we want to describe our samples
 - we use [PDF](#) when we want to describe the hypothesized underlying distribution
 - A PDF answers the question: "[How common are samples at exactly this value?](#)"
Example: the [histogram](#) of a random process is an estimation of the probability density function (PDF) of that random process

```
122 # Statistics:  
123 # 1. Random Variables:  
124 from numpy import array, arange, random, histogram  
125 from scipy import stats  
126 from pylab import plot, grid, legend  
127  
128 # Lets create a NumPy array first:  
129 a = random.normal(size=200)  
130 x = arange(-5, 6)  
131 y = 0.5*(x[1:] + x[:-1])  
132  
133 # 1.1 Take the Probability Density Function (PDF):  
134 b = stats.norm.pdf(y) # here 'norm' is the distribution  
135  
136 # 1.2 Create a histogram:  
137 hist = histogram(a, bins=x, normed=True)[0]  
138  
139 # Plot the results:  
140 plot(y, hist, 'ro-', label='Histogram');  
141 plot(y, b, 'bo-', label='PDF')  
142 legend(loc='upper right')  
143 grid(True)
```

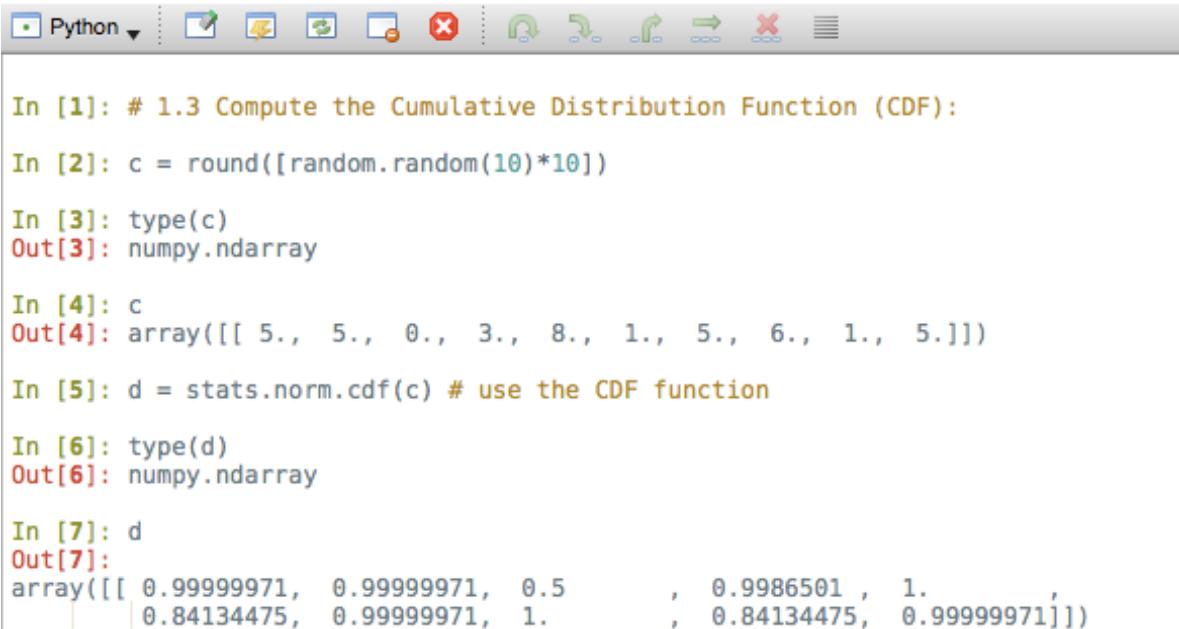


Statistics

- Statistics – Random Variables (RV) – CDF

- A CDF answers the question "How common are samples that are less than or equal to this value?"
- **CDF is the integral of PDF**, where PDF has to integrate to 1 and CDF is obtained by integrating the PDF from $(-\infty : z)$ and $z = \text{whatever point we want}$
- we take a **normal random variable** by firstly creating a NumPy array of numbers and then using the **Cumulative Distribution Function (CDF)** to calculate a vector of points

Example:



```
In [1]: # 1.3 Compute the Cumulative Distribution Function (CDF):  
  
In [2]: c = round([random.random(10)*10])  
  
In [3]: type(c)  
Out[3]: numpy.ndarray  
  
In [4]: c  
Out[4]: array([[ 5.,  5.,  0.,  3.,  8.,  1.,  5.,  6.,  1.,  5.]])  
  
In [5]: d = stats.norm.cdf(c) # use the CDF function  
  
In [6]: type(d)  
Out[6]: numpy.ndarray  
  
In [7]: d  
Out[7]: array([[ 0.99999971,  0.99999971,  0.5         ,  0.9986501  ,  1.  
           | 0.84134475,  0.99999971,  1.         ,  0.84134475,  0.99999971]])
```

Statistics

- Statistics – Random Variables (RV) – Random Variates

- the expression: `scipy.stats.norm.rvs(12,34)` generates a random sample from a random Gaussian (normal) distribution with **mean 12** and **standard deviation 34**
- the function call `scipy.stats.norm.rvs(34, 41, size = 12)` creates an array of **12 samples** from a Gaussian distribution with **mean 34** and **standard deviation 41**

Example:

```
153 from numpy import mean, std, median, var
154 from scipy import stats
155
156 # 1.4 Generate a random sample from a Gaussian distribution
157 #     with mean 12 and standard deviation 34 using 'rvs':
158 e = stats.norm.rvs(12,34)
159 e
160
161 # Create an array of 12 samples from a Gaussian distribution
162 # with mean 34 and standard deviation 41 using 'rvs':
163 f = stats.norm.rvs(34, 41, size = 12)
164 f
165 mean(f)      # find the mean in 'f'
166 median(f)    # find the median in 'f'
167 std(f)       # find the std in 'f'
168 var(f)       # find the variance in 'f'
169
170 # We can generate a vector with random variates like this:
171 g = stats.norm.rvs(34, size=12)
172 g
173
174 # However, this syntax will generate a different result:
175 h = stats.norm.rvs(34)
176 h
```

where:

In [8]: e
Out[8]: 6.146797153855011

In [9]: f
Out[9]: array([105.12910258, 21.81276355, 60.32536413, 83.38992546,
48.84749003, 79.76435341, 65.9802982 , 59.14567579,
89.21333067, -18.38905114, 55.71356883, 9.62385194])

In [10]: g
Out[10]: array([34.51800431, 35.32589712, 33.97931315, 34.61763683,
33.76703425, 33.56391954, 33.83717582, 33.64128155,
34.35934696, 33.74579288, 33.02609254, 35.29855233])

In [11]: h
Out[11]: 33.95149705772917

and:

In [12]: mean(f) # find the mean in 'f'
Out[12]: 55.046389456433197

In [13]: median(f) # find the median in 'f'
Out[13]: 59.735519959912388

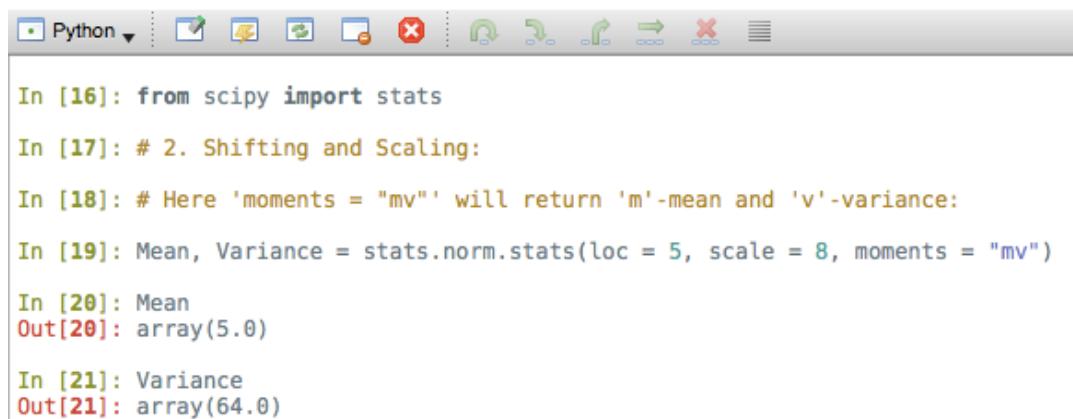
In [14]: std(f) # find the std in 'f'
Out[14]: 33.97735778260459

In [15]: var(f) # find the variance in 'f'
Out[15]: 1154.4608422035087

Statistics

- Statistics – Shifting and Scaling

- all continuous distributions take the two parameters: `loc` and `scale` as keyword parameters
- they adjust the `location` and `scale` of a given distribution
- so for the standard normal distribution the `location` = `mean` and the `scale` = `std`



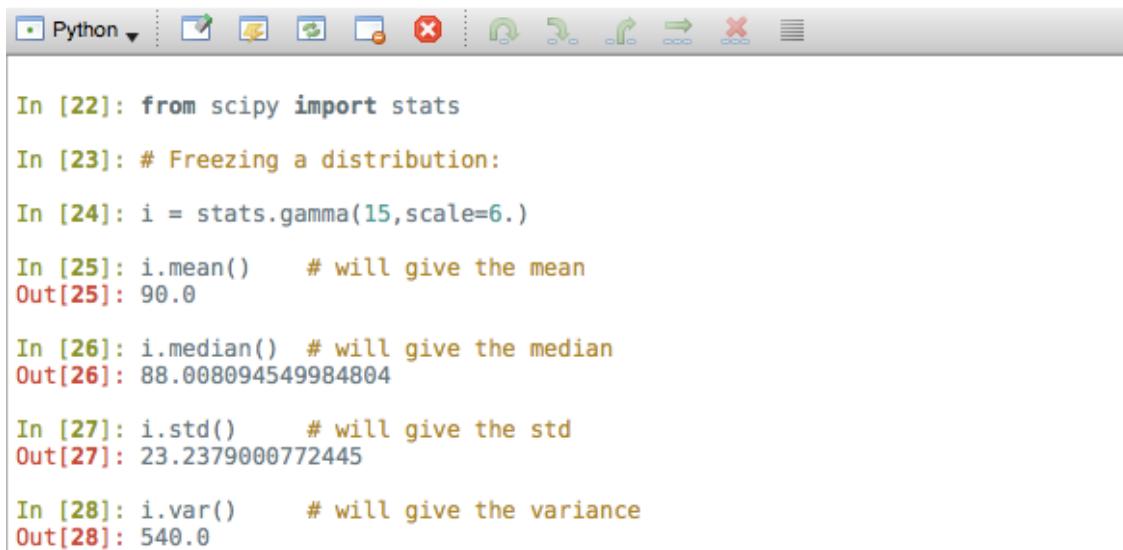
```
In [16]: from scipy import stats
In [17]: # 2. Shifting and Scaling:
In [18]: # Here 'moments = "mv"' will return 'm'-mean and 'v'-variance:
In [19]: Mean, Variance = stats.norm.stats(loc = 5, scale = 8, moments = "mv")
In [20]: Mean
Out[20]: array(5.0)
In [21]: Variance
Out[21]: array(64.0)
```

- the standardized distribution for a random variable X equals $(X - \text{loc}) / \text{scale}$, hence the default values for `loc` and `scale` are respectively `0` and `1`
- so the output of variable `h` in the `RV` example of the previous slide `stats.norm.rvs(34)` generates a normally distributed random variate with mean `loc = 34`, hence the difference between `g` and `h`

Statistics

- Statistics – Fitting Distributions – freezing loc and scale
 - passing the `loc` and `scale` parameters can become annoying, hence we can **freeze them** (fix them)
 - frozen RV object is one using the `same methods` but **holding a given shape, location, and scale fixed**
 - we use `stats.gamma` in order not to include the `scale` or the `shape` parameters anymore and to effectively freeze them

Example:



```
In [22]: from scipy import stats
In [23]: # Freezing a distribution:
In [24]: i = stats.gamma(15,scale=6.)
In [25]: i.mean()      # will give the mean
Out[25]: 90.0
In [26]: i.median()   # will give the median
Out[26]: 88.008094549984804
In [27]: i.std()       # will give the std
Out[27]: 23.2379000772445
In [28]: i.var()        # will give the variance
Out[28]: 540.0
```

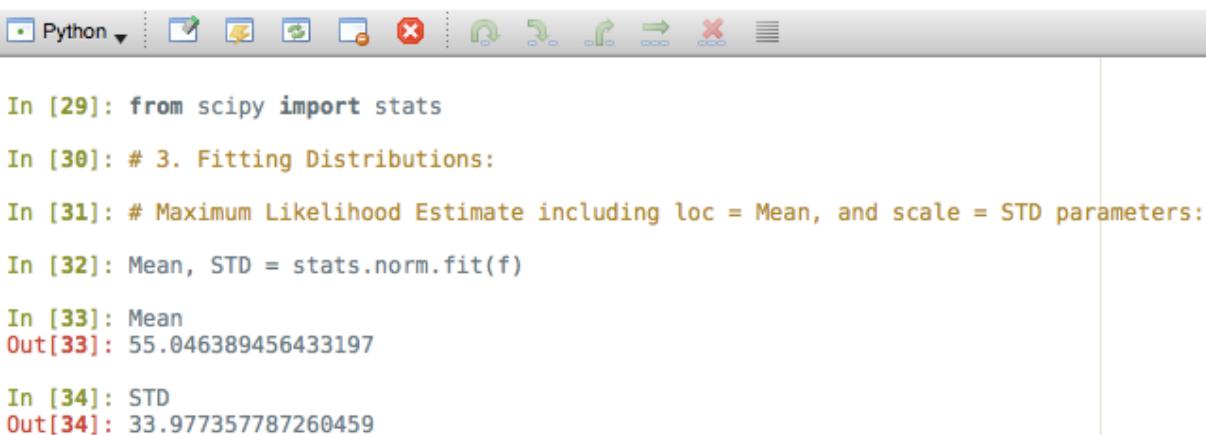
Statistics

- Statistics – Fitting Distributions

- the main methods of a not frozen distribution are related to the estimation of distribution parameters:
 - `fit`: Maximum Likelihood Estimation of distribution parameters, including `location` and `scale`
 - `fit_loc_scale`: when shape parameters are provided, it estimates `location` and `scale`
 - `expect`: computes the `expectation` of a function `against the PDF`
 - `nllf`: calculates the negative log likelihood function
- when we have a normal processes, we can use a **Maximum-Likelihood Fit (MLF)** of the observations to estimate the parameters of the given distribution

Example:

note: the **Mean** and **STD** values for variable **f** in the **RV** example of the previous slide are exactly the same



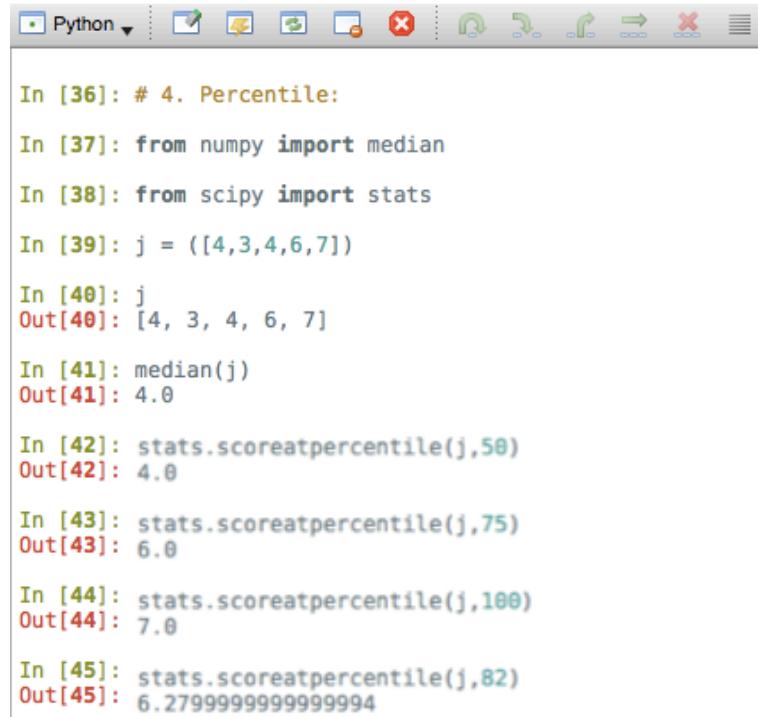
```
In [29]: from scipy import stats
In [30]: # 3. Fitting Distributions:
In [31]: # Maximum Likelihood Estimate including loc = Mean, and scale = STD parameters:
In [32]: Mean, STD = stats.norm.fit(f)
In [33]: Mean
Out[33]: 55.046389456433197
In [34]: STD
Out[34]: 33.977357787260459
```

Statistics

- Statistics – Percentile

- the **median** in a given vector is the value that is located at **equal distances** from the **first** and **last** element in that vector
- in case we would like to **select another value** from that vector, **including a value between elements** in the same vector we use **percentile**
- the **percentile** method gives the ability to select the **percentage** at which the **observation** should focus in the vector
- the **percentile** is a CDF estimator

Example:



```
In [36]: # 4. Percentile:  
In [37]: from numpy import median  
In [38]: from scipy import stats  
In [39]: j = ([4,3,4,6,7])  
In [40]: j  
Out[40]: [4, 3, 4, 6, 7]  
In [41]: median(j)  
Out[41]: 4.0  
In [42]: stats.scoreatpercentile(j,50)  
Out[42]: 4.0  
In [43]: stats.scoreatpercentile(j,.75)  
Out[43]: 6.0  
In [44]: stats.scoreatpercentile(j,100)  
Out[44]: 7.0  
In [45]: stats.scoreatpercentile(j,.82)  
Out[45]: 6.279999999999994
```

Statistics

- Statistics – Statistical Tests
 - statistical tests serve as decision indicators when we have two vectors with observations generated from Gaussian distribution process
 - a T-test can be used to decide if the two sets of observations are different and to what degree
 - the output of such test is as follows:
 - T value: represents the number that has:
 - sign proportional to the difference between the two distributions
 - magnitude that is related to the degree of the difference between the two vectors
 - P value: represents the probability of both distributions being identical
 - when P = 1, the two vectors are identical
 - when P is close to 0, the two distributions are very likely to have different meaning

Statistics

- Statistics – Statistical Tests

Example:

- observe how the T and P values change when we make l to be the same as a portion of k
- also, see how the sign of the T value changes when we reverse the order of k and l

```
In [121]: from numpy import random
In [122]: from scipy import stats
In [123]: # We create the two vectors using random normal distribution:
In [124]: k = random.normal(1, 5, size=25)
In [125]: l = random.normal(3, 6, size=10)

In [126]: k
Out[126]:
array([ 12.58238816, -1.6753272 , -2.74962851,  9.04077941,
       -0.82362539, -10.32995868,  0.94521675, -1.74637583,
       0.48603381, 10.99541606, -2.69298861, -2.83599334,
      -11.33668027,  7.86934694,  1.5608238 , -10.54091636,
       4.61734667,  3.67922731,  0.74947059, 11.18795862,
      3.33801024, -2.33968043, -1.21651045,  4.87192564,  3.26540041])

In [127]: l
Out[127]:
array([ 10.76506563, -6.12857422, -7.75417503,  9.82786799,
       -0.34576453, -7.95220394,  5.41105004, -0.36373088,
      3.28697705, -6.39391327])

In [128]: T, P = stats.ttest_ind(k, l)

In [129]: T
Out[129]: 0.42096326000412648

In [130]: P
Out[130]: 0.67651102044315836

In [131]: # Lets make them identical between [0:10]:
In [132]: l[0:10] = k[0:10]

In [133]: l
Out[133]:
array([ 12.58238816, -1.6753272 , -2.74962851,  9.04077941,
       -0.82362539, -10.32995868,  0.94521675, -1.74637583,
       0.48603381, 10.99541606])

In [134]: T, P = stats.ttest_ind(k, l)

In [135]: T
Out[135]: -0.24120361863694167

In [136]: P
Out[136]: 0.81088974280652271
```

Optimization

- Optimization
 - optimization is the process of **finding minimization or equality** with the goal of making a process as **fully functional and effective** as possible
 - **scipy.optimize** provides a rich functionality for optimization structures some of which are:
 - optimization:
 - local optimization:
 - » **minimize** – minimization of a scalar function containing one or more variables
 - » **minimize_scalar** – minimization of a scalar function with one variable
 - general-purpose multivariate methods:
 - » **fmin** – minimize a function using the **downhill simplex algorithm**
 - » **fmin_bfgs** – minimize a function using the BFGS algorithm for non-linear opt.
 - constrained multivariate methods:
 - » **fmin_l_bfgs_b** – minimize a given function by using the L-BFGS-B algorithm
 - univariate (scalar) minimization methods:
 - » **fminbound** – bounded minimization for scalar functions

Optimization

- Optimization
 - `scipy.optimize` provides a rich functionality for optimization structured some of which are:
 - optimization:
 - global optimization:
 - » `brute` – minimize a function over a specified range using brute force
 - fitting:
 - `curve_fit` – use non-linear least squares to fit a function f to data
 - root finding:
 - scalar:
 - `brentq` – find a root of a function in a given interval
 - `brenth` – find the root of a function in $[a,b]$ interval
 - multidimensional:
 - `root` – find a root of a vector function
 - `fsolve` – find the roots of a function
 - ... and much more

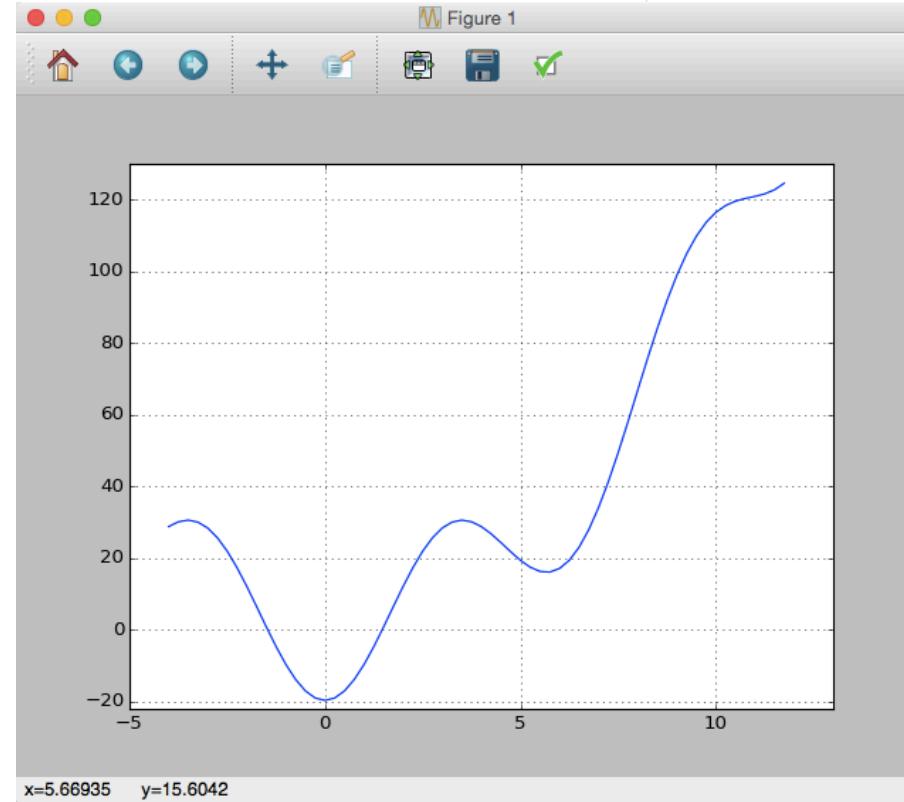
Optimization

- Optimization

Example:

```
232 # 6. Optimization:  
233 from numpy import cos, pi, arange  
234 from scipy import optimize  
235 from pylab import plot, grid, show, xlim, ylim  
236  
237 # Lets find the minimum of a function:  
238 def fun(m):  
239     n = m**2 - 6.25*cos(m)*pi  
240     return n  
241  
242 # Now plot it:  
243 m = arange(-4, 12, 0.25)  
244 plot(m, fun(m))  
245 xlim(-5,13); ylim(-22,130)  
246 grid(True)  
247 show()
```

- from the graph we see that there is a **minimum** at point **x = 0.00**
- we also observe that there is a **local minima** at point **x = 5.67**
- considering the code above, lets find the minimum of the specified scalar function

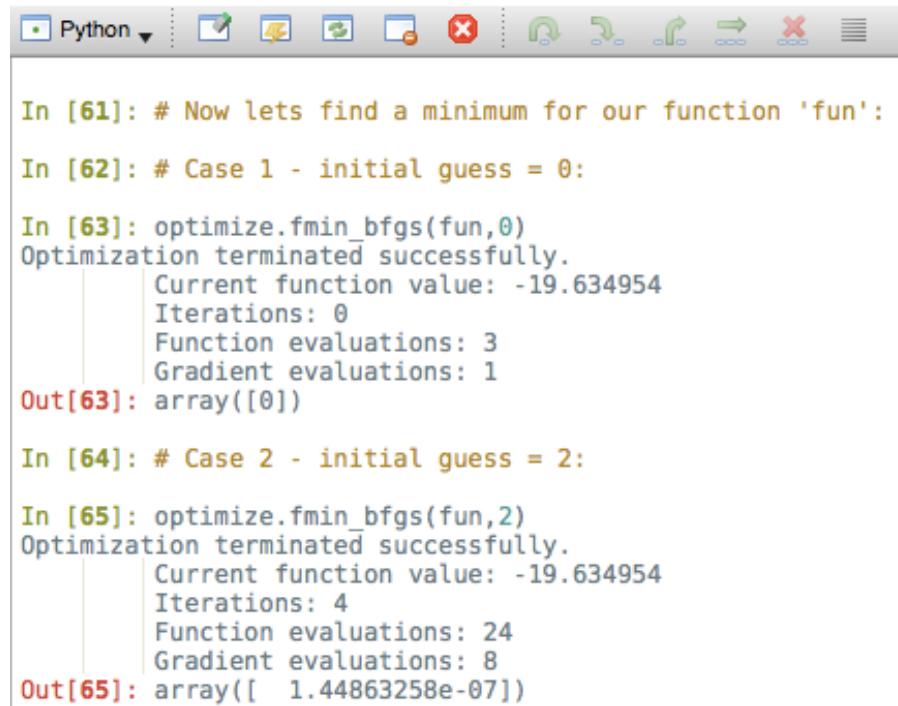


Optimization

- Optimization

Example:

- lets use the `optimize.fmin_bfgs` function to find a minimum for our function `fun`
- we will perform a **gradient descent** starting **from a given initial point** in order to decrease the computational cost:
 - Case 1: we start from 0
we see that the guess was correct hence no iterations were performed
 - Case 2: we start from 2
we see that 4 iterations were obtained to find a minima at about $1.4486e-07$ which is close enough to zero



The screenshot shows a Jupyter Notebook interface with a Python kernel. The code cell In [61] contains a comment about finding a minimum for a function 'fun'. The code cell In [62] sets an initial guess of 0. The code cell In [63] runs the optimization with `optimize.fmin_bfgs(fun, 0)`. The output shows the optimization terminated successfully with a current function value of -19.634954, 0 iterations, 3 function evaluations, and 1 gradient evaluation. The code cell Out[63] shows the result as an array([0]). The code cell In [64] runs the optimization with an initial guess of 2. The code cell In [65] runs the optimization with an initial guess of 2. The output shows the optimization terminated successfully with a current function value of -19.634954, 4 iterations, 24 function evaluations, and 8 gradient evaluations. The code cell Out[65] shows the result as an array([1.44863258e-07]).

```
In [61]: # Now lets find a minimum for our function 'fun':  
In [62]: # Case 1 - initial guess = 0:  
In [63]: optimize.fmin_bfgs(fun, 0)  
Optimization terminated successfully.  
    Current function value: -19.634954  
    Iterations: 0  
    Function evaluations: 3  
    Gradient evaluations: 1  
Out[63]: array([0])  
  
In [64]: # Case 2 - initial guess = 2:  
In [65]: optimize.fmin_bfgs(fun, 2)  
Optimization terminated successfully.  
    Current function value: -19.634954  
    Iterations: 4  
    Function evaluations: 24  
    Gradient evaluations: 8  
Out[65]: array([ 1.44863258e-07])
```

Optimization

- Optimization

Example:

- since our function has one **local minima** the algorithm may find it instead of finding the **global minimum** depending on the initial point, which may cause a problem
- when we don't know the proximity of the global minimum and can pick the initial point conveniently, we are forced to use a much **costlier global optimization**
- to find the global minimum in this case, the **brute force** algorithm will suffice, where the function is evaluated at each point of a given vector grid

```
In [66]: # To find a local minima we start at a close point to the minima:  
In [67]: optimize.fmin_bfgs(fun, 4)  
Optimization terminated successfully.  
    Current function value: 16.091014  
    Iterations: 5  
    Function evaluations: 33  
    Gradient evaluations: 11  
Out[67]: array([ 5.66775189])
```

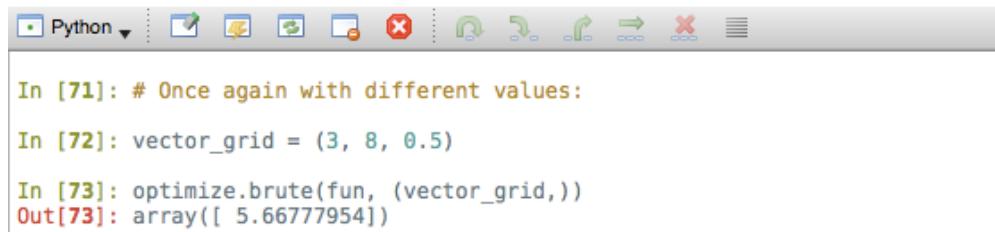
```
In [68]: # Now lets use brute force over a larger region of points:  
In [69]: vector_grid = (-4, 8, 0.5)  
In [70]: optimize.brute(fun, (vector_grid,))  
Out[70]: array([ 0.])
```

Optimization

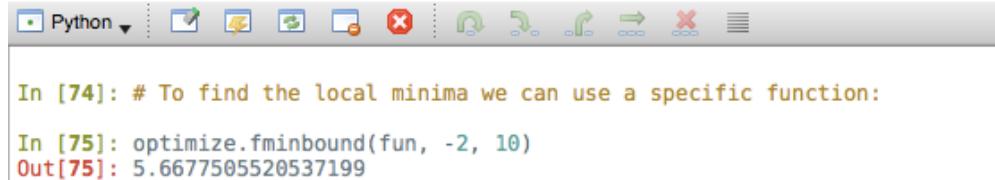
- Optimization

Example:

- to find the local minimum in this case, we once again can use the **brute force** algorithm, but now the function is evaluated at points surrounding the minima
- there is a specific function built for the purpose of specifically finding the **local minima** `optimize.fminbound`



```
In [71]: # Once again with different values:  
In [72]: vector_grid = (3, 8, 0.5)  
In [73]: optimize.brute(fun, (vector_grid,))  
Out[73]: array([ 5.66777954])
```



```
In [74]: # To find the local minima we can use a specific function:  
In [75]: optimize.fminbound(fun, -2, 10)  
Out[75]: 5.6677505520537199
```

Optimization

- Optimization – **curve fitting**

Example:

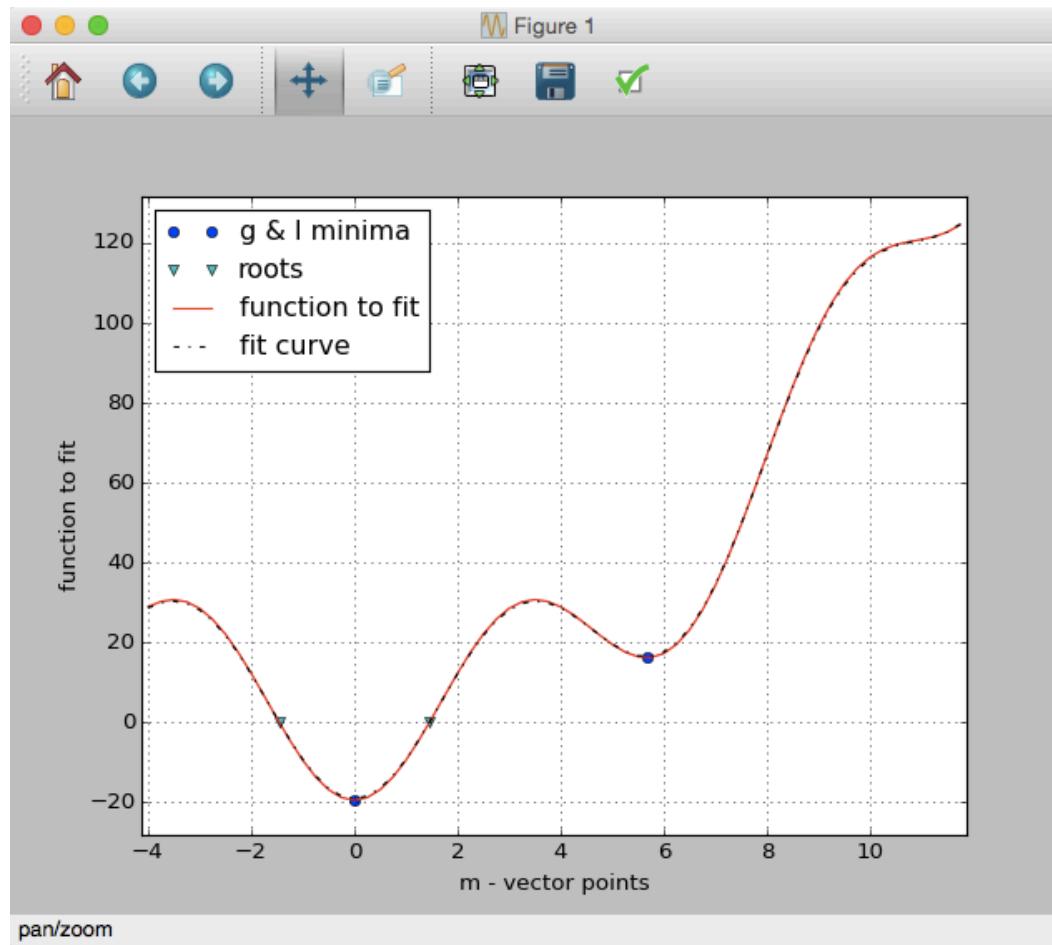
- first we define the curve - **fun1**
- we define the function to fit - **fun2**
- suppose we know the form of **fun1**, but we do not know the amplitudes, so we need to find them by using least-squares curve fitting
- the roots and minima are:

```
267 # Curve fitting:
268 from numpy import linspace, arange, array, random, cos, pi
269 from scipy import optimize
270 from pylab import plot, grid, xlabel, ylabel, legend, grid
271
272 # Here we create the initial curve:
273 def fun1(r):
274     n = r**2 - 6.25*cos(r)*pi
275     return n
276
277 # We define the fit curve function, but passing new (guessed) amplitudes:
278 def fun2(r, x, y):
279     return x*r**2 + y*cos(r)*pi
280
281 # Finding the roots:
282 rt1 = optimize.fsolve(fun1, 4) # we solve 'fun1' with guess=4 to find root1
283 rt2 = optimize.fsolve(fun1, -3) # we solve 'fun1' with guess=-3 to find root2
284 roots = array([rt1, rt2]) # we create an array consisting of the roots
285
286 # Finding the minima points:
287 vector_grid = (-4, 8, 0.5) # we use guess range to find the global minima
288 global_min = optimize.brute(fun1, (vector_grid,)) # find the global minima 'g'
289 local_min = optimize.fminbound(fun1, -2, 12) # find local minima 'l'
290 min_points = array([global_min[0], local_min]) # array with 'g' & 'l' minima
291
292 vector = linspace(-8, 8, num=10) # sample vector for fitting curve - fun2
293 new_vec = fun1(vector) + random.randn(len(vector)) # we add 'randn' noise here
294
295 # To find the amplitudes 'x' & 'y', we use least squares curve fitting:
296 initial_guess = [-1, 4] # we make an initial guess
297 par, cov_par = optimize.curve_fit(fun2, vector, new_vec, initial_guess)
298
299 # Now plot the results:
300 m = arange(-4, 12, 0.25) # we create a vector with sample data points to plot
301 plot(min_points, fun1(min_points), 'bo', label="g & l minima")
302 plot(roots, fun1(roots), 'cv', label="roots")
303 plot(m, fun1(m), 'r-', linewidth=1, label="function to fit")
304 plot(m, fun2(m, *par), 'k-.', linewidth=1, label="fit curve")
305
306 legend(loc='best')
307 xlabel('m - vector points'); ylabel('function to fit')
308 grid(True)
```

Optimization

- Optimization – **curve fitting**

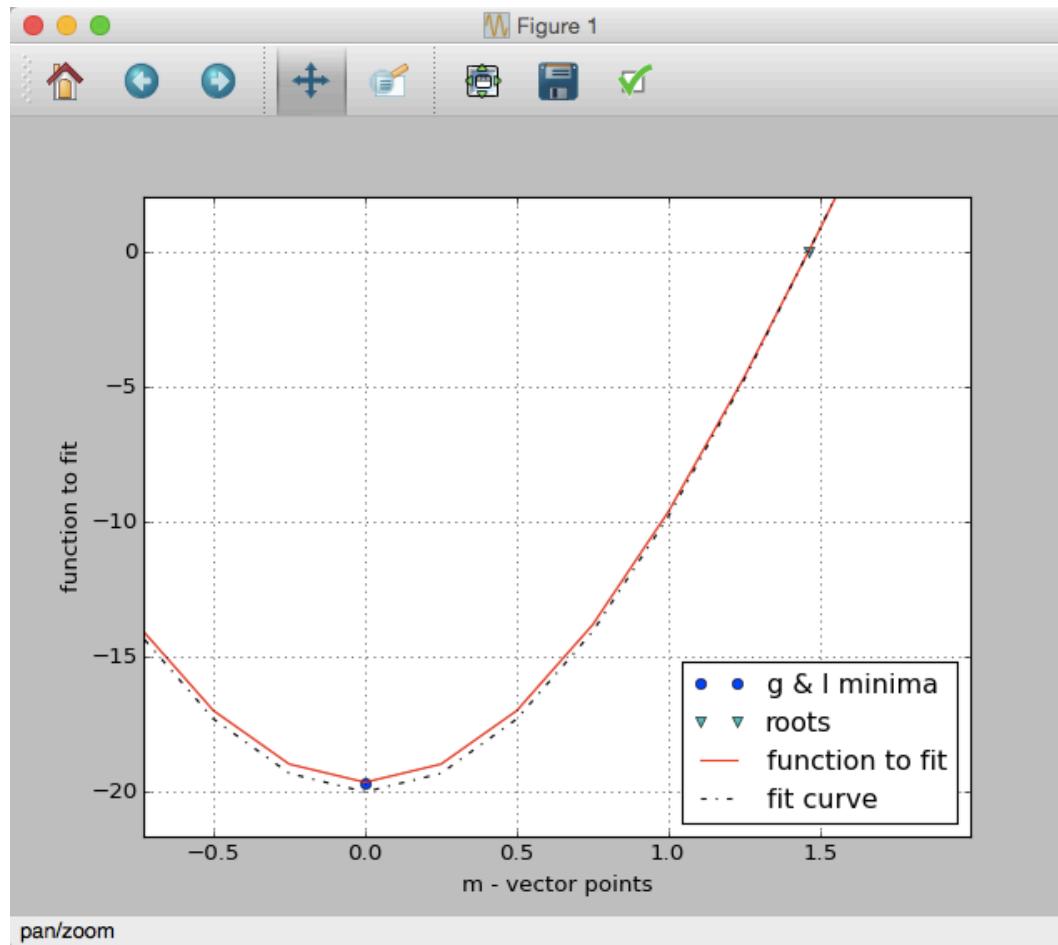
Example:



Optimization

- Optimization – **curve fitting**

Example:

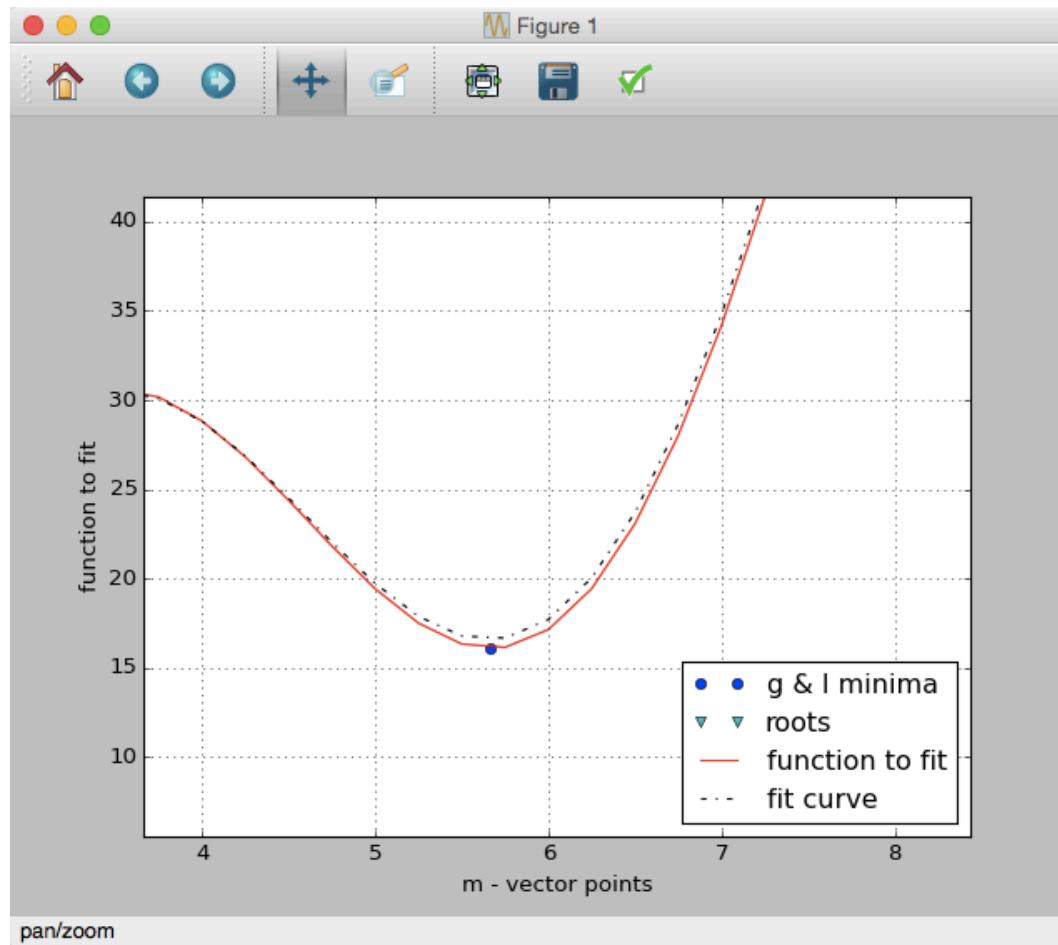


Optimization

- Optimization – **curve fitting**

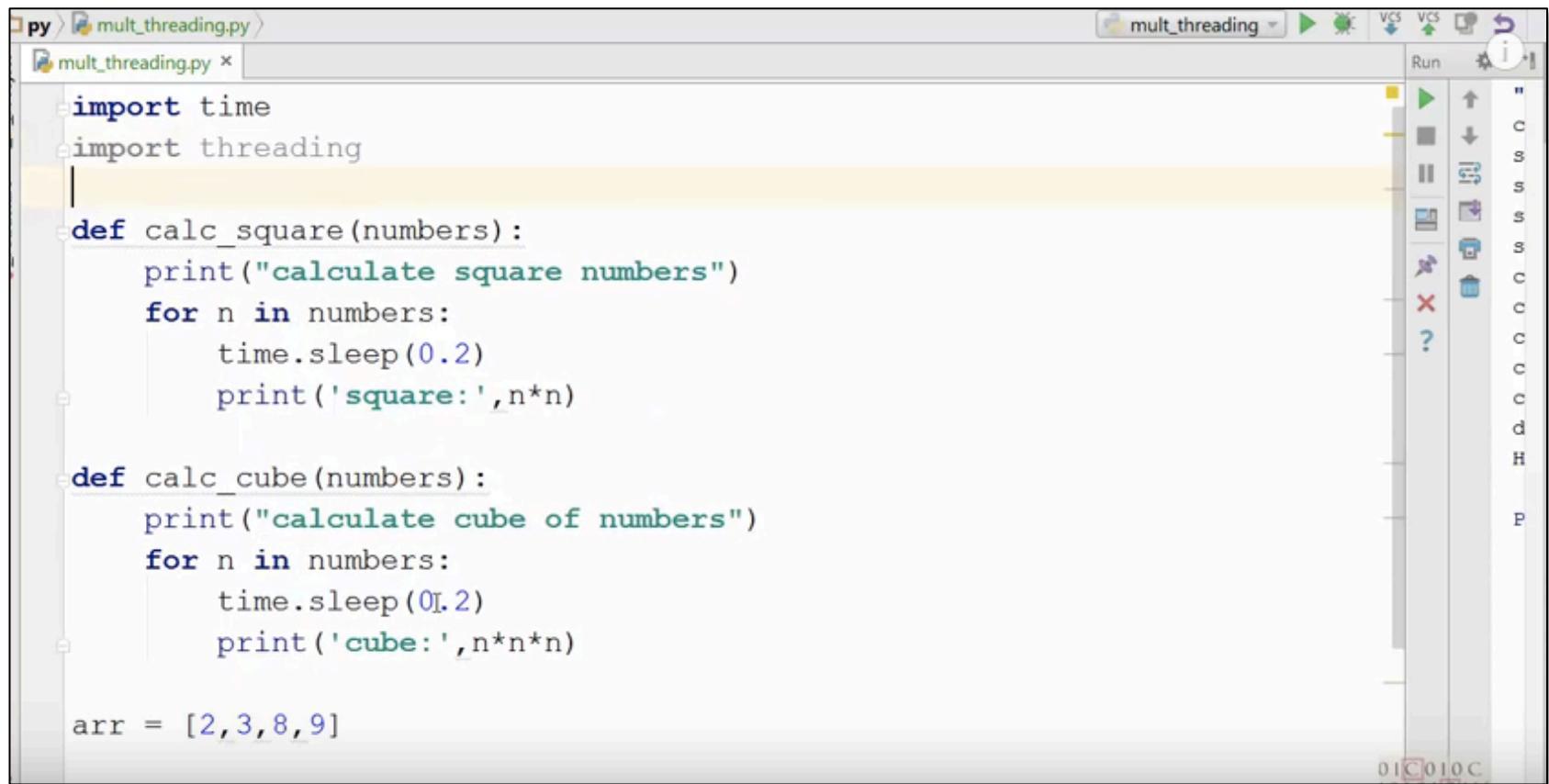
Example:

Local minima →



Multithreading

- Multithreading



The screenshot shows the PyCharm IDE interface with a Python file named `mult_threading.py` open. The code implements multithreading to calculate squares and cubes of numbers. It uses the `threading` module and the `time` module's `sleep` function to introduce delays between calculations.

```
py > mult_threading.py >
mult_threading.py x
import time
import threading

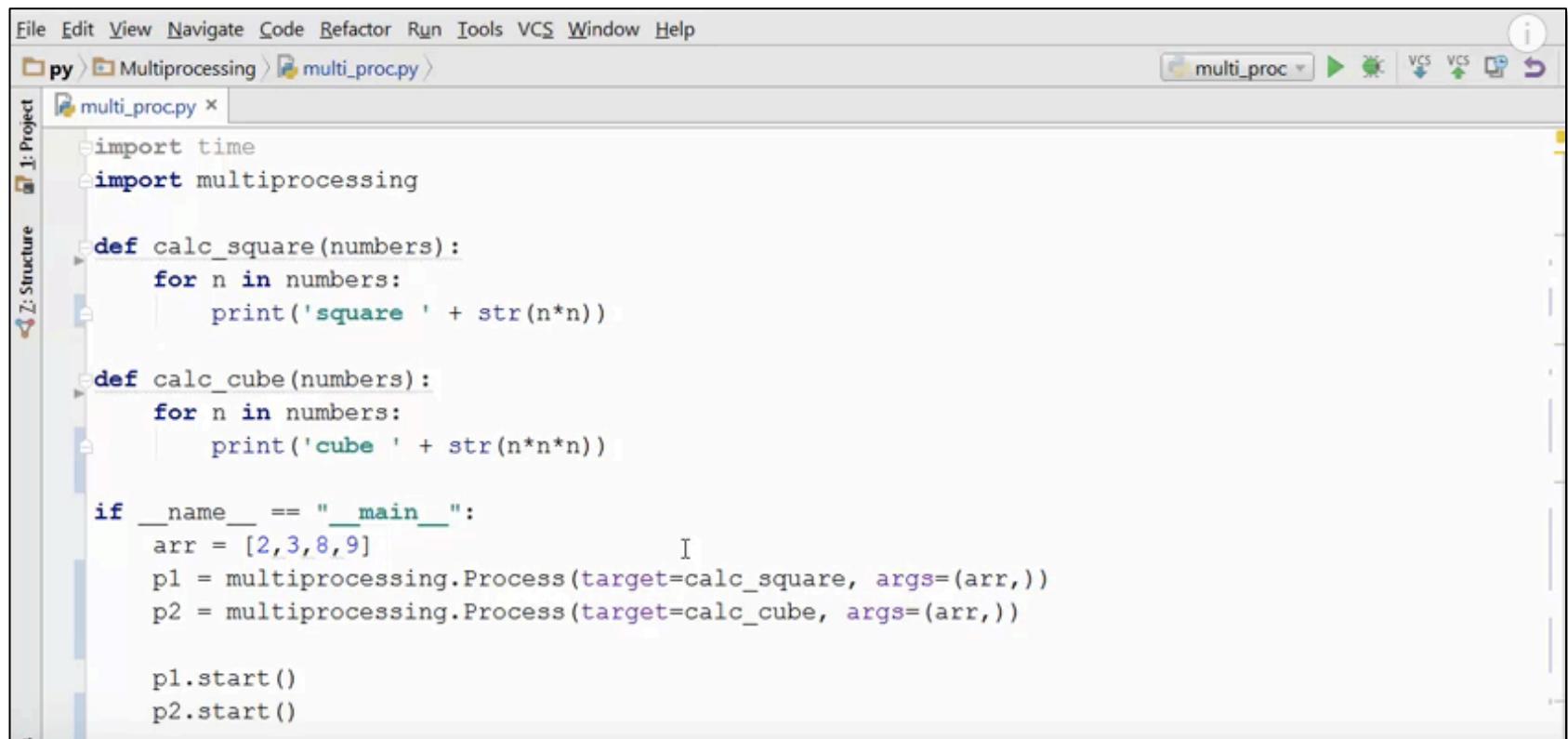
def calc_square(numbers):
    print("calculate square numbers")
    for n in numbers:
        time.sleep(0.2)
        print('square:', n*n)

def calc_cube(numbers):
    print("calculate cube of numbers")
    for n in numbers:
        time.sleep(0.2)
        print('cube:', n*n*n)

arr = [2, 3, 8, 9]
```

Multiprocessing

- Multiprocessing



The screenshot shows a PyCharm IDE interface with a Python file named `multi_proc.py` open. The code implements the `multiprocessing` module to calculate squares and cubes of a list of numbers.

```
File Edit View Navigate Code Refactor Run Tools VCS Window Help
py > Multiprocessing > multi_proc.py >
multi_proc.py x
Project Structure
import time
import multiprocessing

def calc_square(numbers):
    for n in numbers:
        print('square ' + str(n*n))

def calc_cube(numbers):
    for n in numbers:
        print('cube ' + str(n*n*n))

if __name__ == "__main__":
    arr = [2,3,8,9]
    p1 = multiprocessing.Process(target=calc_square, args=(arr,))
    p2 = multiprocessing.Process(target=calc_cube, args=(arr,))

    p1.start()
    p2.start()
```

Summary

- Here is what we covered in this class:
- **Introduction to Python®**
 - Python - pros and cons
 - Installing the environment with core packages
 - Python modules, packages and scientific blocks
 - Working with the shell, IPython and the editor
 - Basic arithmetic operations, assignment operators, data types, containers
 - Iterative programming (if/elif/else)
 - Conditional expressions
 - Recursion programming (for/continue/while/break)
 - Functions: definition, return values, local vs. global variables
 - Classes / Functions (cont.): objects, methods, passing by value and reference
 - Scripts, modules, packages
 - I/O interaction with files
 - Standard library
 - Exceptions
 - Multithreading and Multiprocessing
- **Matplotlib**
 - What is Matplotlib?
 - Basic plotting
 - Tools: title, labels, legend, axis, points, subplots, etc.
 - Advanced plotting: scatter, pie, bar, 3D plots, et
- **NumPy:**
 - Data type objects
 - NumPy arrays
 - NumPy matrices
 - Indexing and slicing of arrays
 - Array and Matrix operations
 - Reductions
 - Broadcasting
 - Data sorting
 - Type casting
 - Masking data
 - Organizing arrays
 - Loading data files: txt, audio, image, mat, npy
 - Dealing with polynomials
 - Good coding practices
- **Scipy:**
 - Working with files – txt, audio, image
 - Algebraic operations
 - The Fast Fourier Transform
 - Signal Processing:
 - convolution, cross- and autocorrelation
 - speech and audio effects,
 - image transformation and filtering
 - Interpolation
 - Statistics
 - Optimization
 - Curve fitting
- **Special formats:**
 - Working with: MySQL, .xls, .xlsx, mysql