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

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

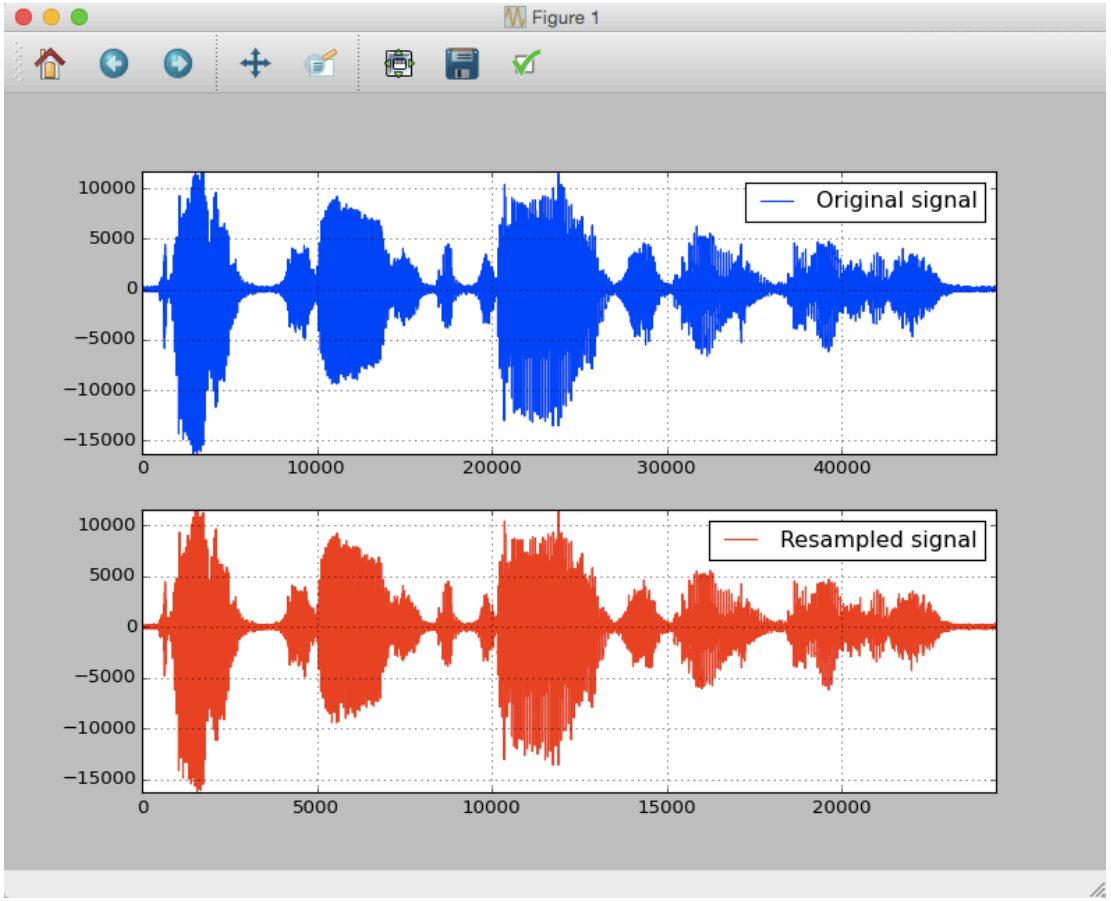
Example:

```
## 3. Modifying sound:
from pylab import specgram, figure, plot, subplot, grid, axis, xlabel, ylabel, title, legend, pause
from scipy.io.wavfile import read, write
import scipy.signal.signaltools as sigtool
from numpy import linspace, pi, fft, abs, exp, hanning, zeros, angle, arange, sin, cos
import sounddevice as sd

# 3.1 Resampling:
(Fs, x) = read('files/lecture9/cheese.wav')
x2=x[0:len(x):2]    # we decrease the original audio samples in half
F = Fs/2            # we now have to decrease Fs by the same amount
F, s = F.as_integer_ratio() # a quick convertion from 'float' to 'int'
write('files/lecture9/cheese_resampled.wav',F,x2)

# Let's listen:
sd.play((x2),F), pause(3)
sd.play((x),Fs), pause(3)

""" Lets plot the two signals: """
figure(1)
subplot(2,1,1)
plot(x,label='Original signal')
axis('tight'); legend(loc='best')
grid(True)
subplot(2,1,2)
plot(x2,'r', label='Resampled signal')
axis('tight'); legend(loc='best')
grid(True)
pause(1)

""" Lets plot the spectrograms of two signals: """
figure(2)
subplot(2,1,1)
specgram(x), axis('tight'), title('Original signal');
ylabel('Frequency')
subplot(2,1,2)
specgram(x2), axis('tight'), title('Resampled (down-sampled) signal');
xlabel('Time'), ylabel('Frequency')
pause(1)
```

Signal Processing

- Signal Processing – sound processing: resampling

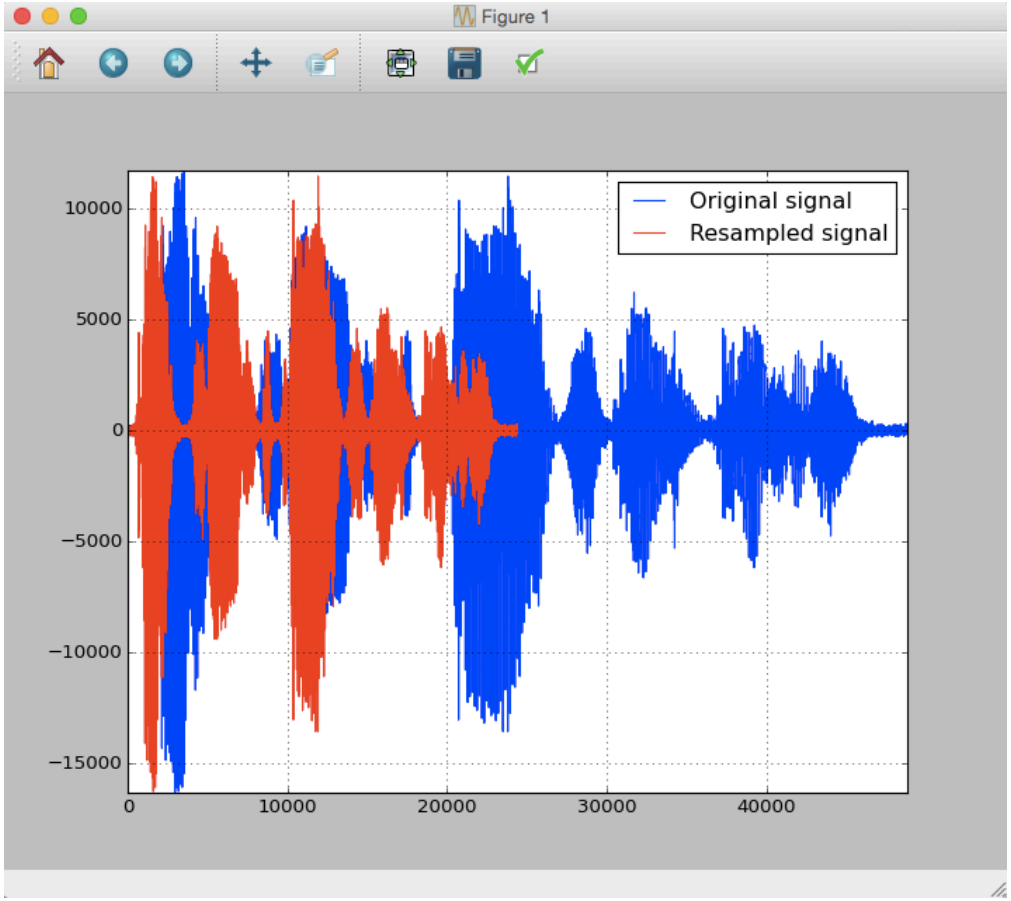
Example:



Signal Processing

- Signal Processing – sound processing: resampling

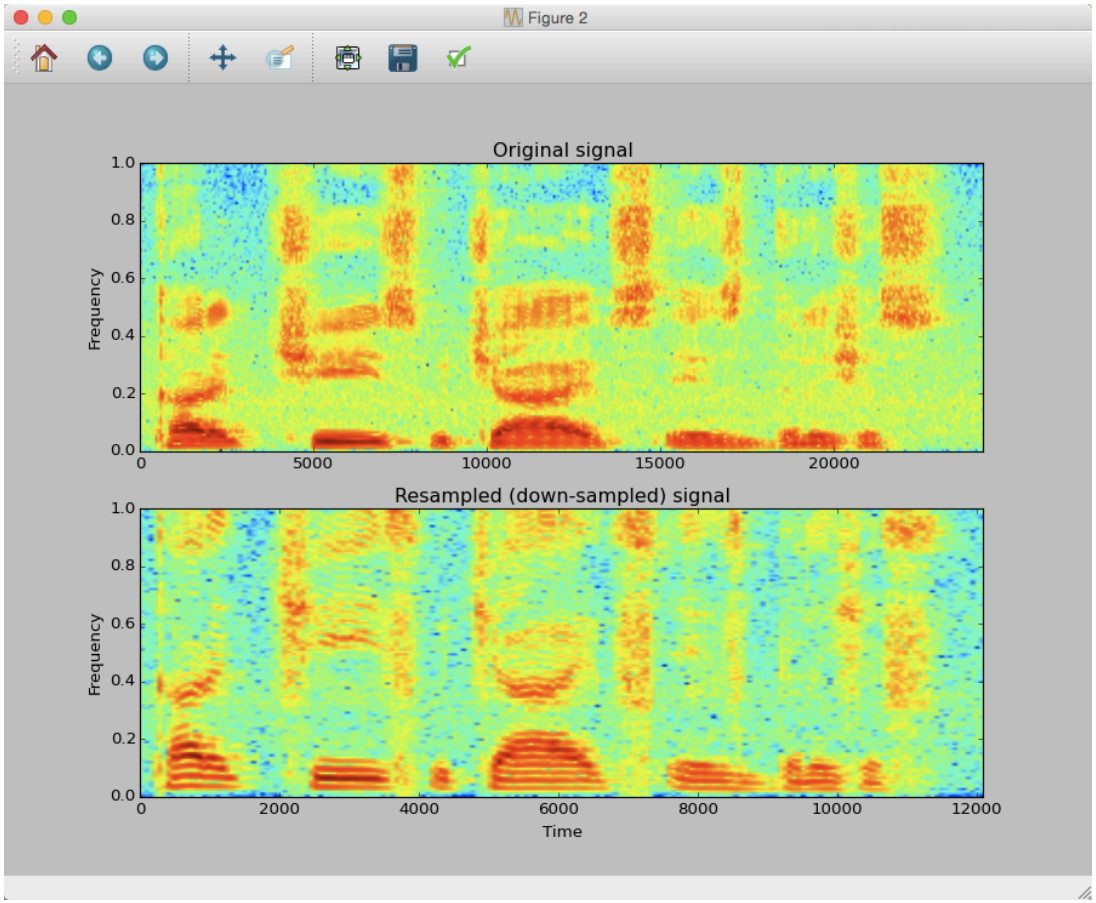
Example:



Signal Processing

- Signal Processing – sound processing: resampling

Example:



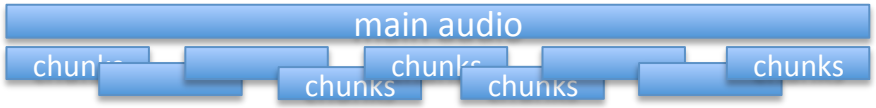
Signal Processing

- 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?

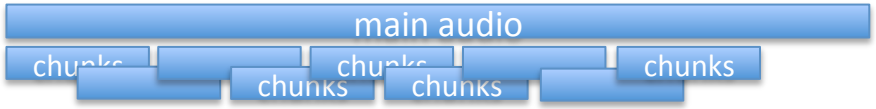
```
76  # 3.2 Time domain manipulations:
77  (Fs, x) = read('files/lecture9/cheese.wav')
78  x2 = x[0:len(x):2] # we lower the original audio samples in half
79  F = Fs/2          # we now lower Fs by the same amount
80  F, s = F.as_integer_ratio() # convertion from 'float' to 'int'
81
82  # We use original 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



      - 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:

```
# 3.3 Frequency domain manipulations (phase vocoder):
from pylab import figure, plot, subplot, grid, axis, xlabel, ylabel, title, legend
from scipy.io.wavfile import read, write
from numpy import pi, fft, exp, hanning, zeros, arange, angle, sin, cos

(Fs, x) = read('files/lecture9/cheese.wav')

def modify(sound, c, win_size, d):

    """ lets define the basic parameters: """
    hann_win = hanning(win_size)
    phase = zeros(win_size)
    final_signal = zeros( len(sound) / c + win_size)
    out = zeros(win_size, dtype=complex)

    for m in arange(0, len(sound)-(win_size + d), d*c):

        """ we define the two neighbouring and overlapping audio parts: """
        aud1 = sound[m:m+win_size]
        aud2 = sound[m+d:m+win_size+d]

        """ we need to synchronize the second to the first window signal: """
        sig1 =  fft.fft(hann_win * aud1)
        sig2 =  fft.fft(hann_win * aud2)

        phase += (angle(sig2) - angle(sig1))

        """ wrap the phase back between pi and -pi: """
        for k in phase:
            if phase[k] < -pi:
                phase[k] = phase[k] + 2*pi
            elif phase[k] >= pi:
                phase[k] = phase[k] - 2*pi

        out.real, out.imag = cos(phase), sin(phase)

        """ increment: """
        win = int(m/c)

        """ combine the newly calculated amplitude and phase parts: """
        final_signal[win : win + win_size] += (fft.ifft(sig2.real*exp(1j*out.imag))).real

    return final_signal.astype('int16')
```

lets discuss

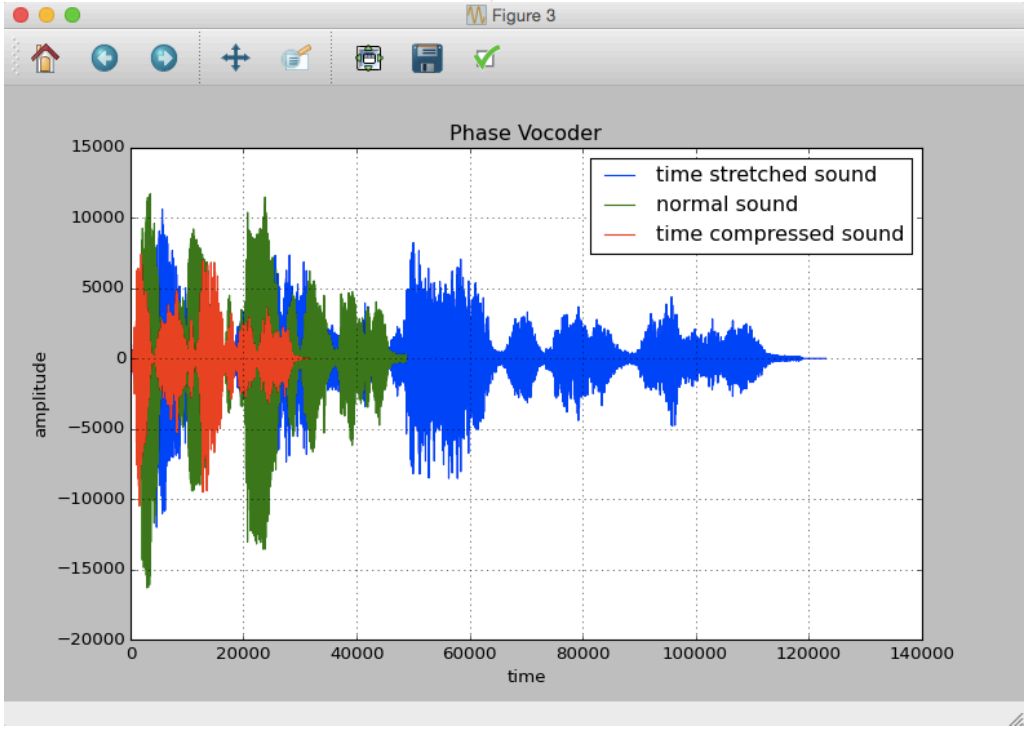
for a minute…

Signal Processing

- Signal Processing – sound processing: frequency domain manipulations

Example:

```
""" Let's test what we have done: """
win_size = 1024; overlap = int(win_size/2)
stretch_coef = 0.4; shrink_coef = 1.6

y = modify(x,stretch_coef,win_size,overlap)
write('files/lecture9/cheese_slower.wav',Fs,y)

y2 = modify(x,shrink_coef,win_size,overlap)
write('files/lecture9/cheese_faster.wav',Fs,y2)

# Now plot the results and compare:
figure(3)
plot(y, label='time stretched sound')
plot(x, label='normal sound')
plot(y2, label='time compressed sound')
legend(loc='best'); title('Phase Vocoder')
xlabel('time'); ylabel('amplitude'); grid(True)
```



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:

```
153 # 3.4 Time-Frequeincy changes / pitch shift:
154 (Fs, x1) = read('files/lecture9/cheese.wav')
155 win_size = 1024; overlap = int(win_size/2)
156 stretch_coef = 0.5
157
158 # Time stretch in frequency domain:
159 y = modify(x,stretch_coef,win_size,overlap)
160
161 # Pitch change in time domain:
162 """ we lower the original audio samples by the 'stretch_coef' """
163 x2 = y[0:len(y):1/stretch_coef]
164 write('files/lecture9/cheese_normalspeed_pitch_up.wav',Fs,x2)
```

Signal Processing

- Signal Processing – sound processing: filter design
  - Usgin 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 filtertobe designed. Choices are: {'lowpass', 'highpass', 'bandpass', 'bandstop'}, the default is 'lowpass'

    analog – (bool), optional parameter. True = analog filter, False = digital filter design

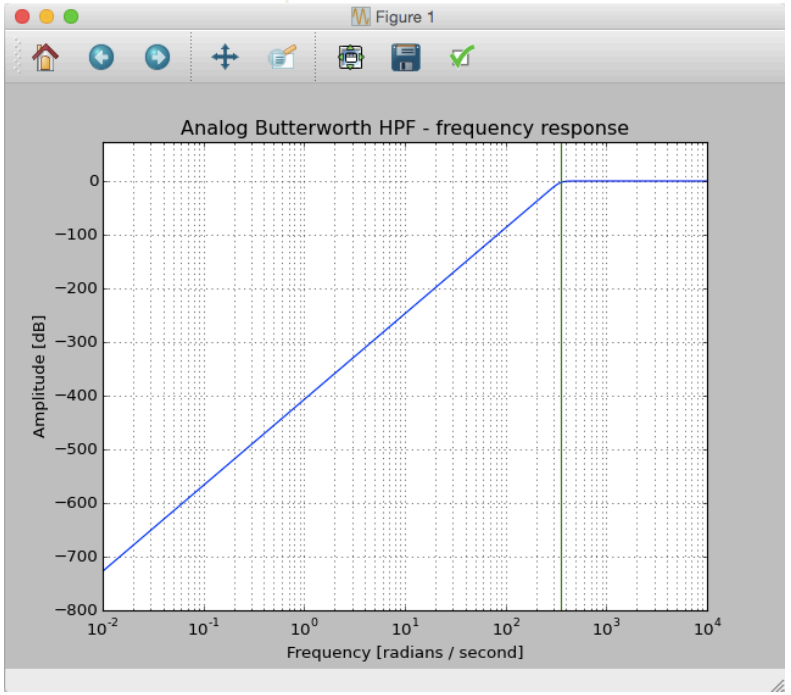
    output – {'ba', 'zpk', 'sos'}, optional parameter.The default is 'ba'. Options are:

    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:

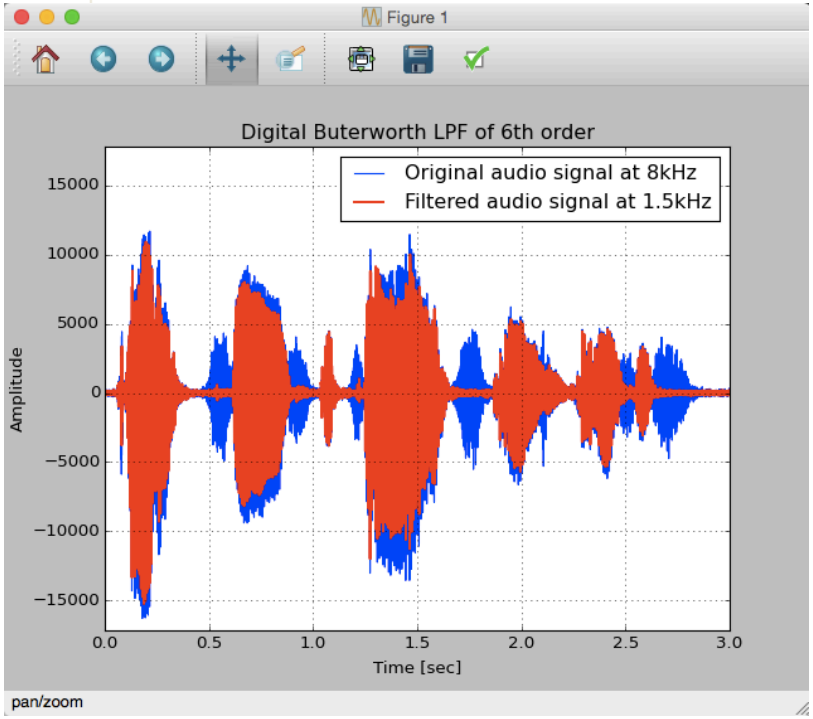
```
174 ## 3.5 Filtering:
175 # Butterworth example of Analog HPF:
176 from scipy import signal, log10
177 from pylab import semilogx, title, xlabel, ylabel, margins, grid, axvline, pause
178
179 b, a = signal.butter(8, 350, 'high', analog=True, output='ba')
180 w, h = signal.freqs(b, a)   # Compute 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:

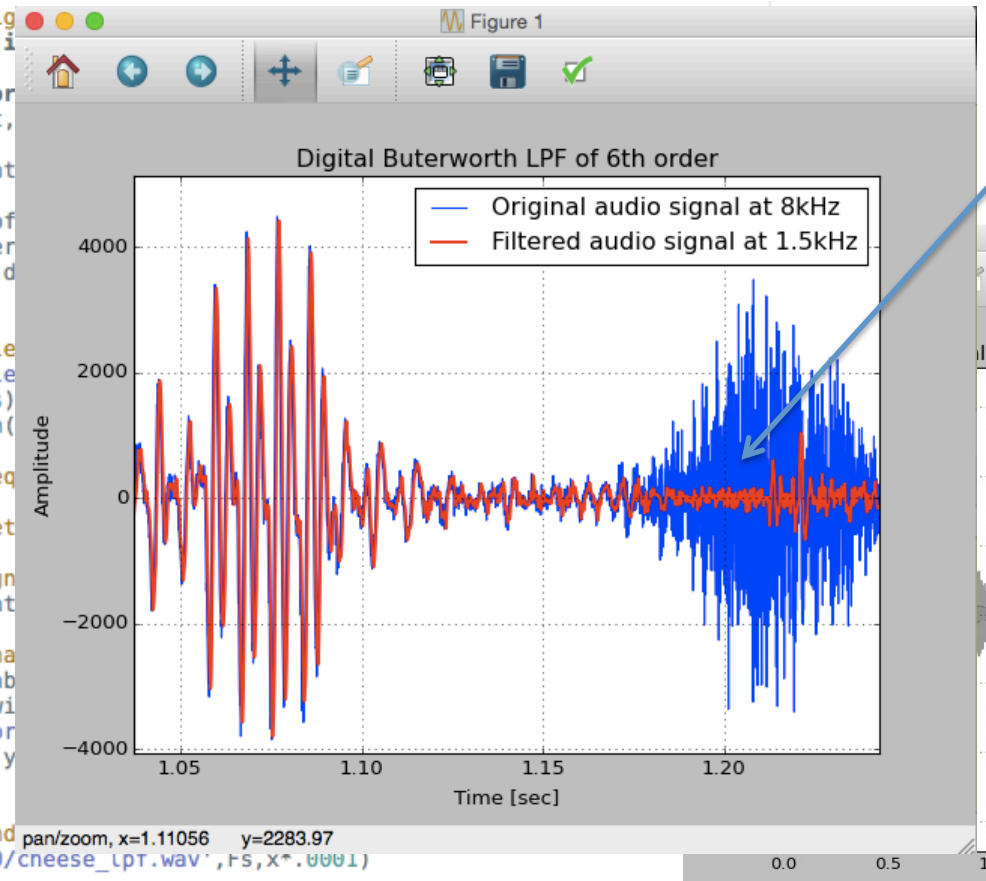
```
182 # Real example of a Digital LP filter:
183 from scipy.io.wavfile import read, write
184 from numpy import int, linspace
185 from scipy.signal import butter, lfilter, freqz
186 from pylab import plot, title, xlabel, ylabel, grid, legend, subplots_adjust, show
187
188 def butterworth_lpf(data, cutoff, fs, order):
189     nyqist = fs / 2
190     cutoff_freq = cutoff / nyqist
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 Buterworth 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)
```



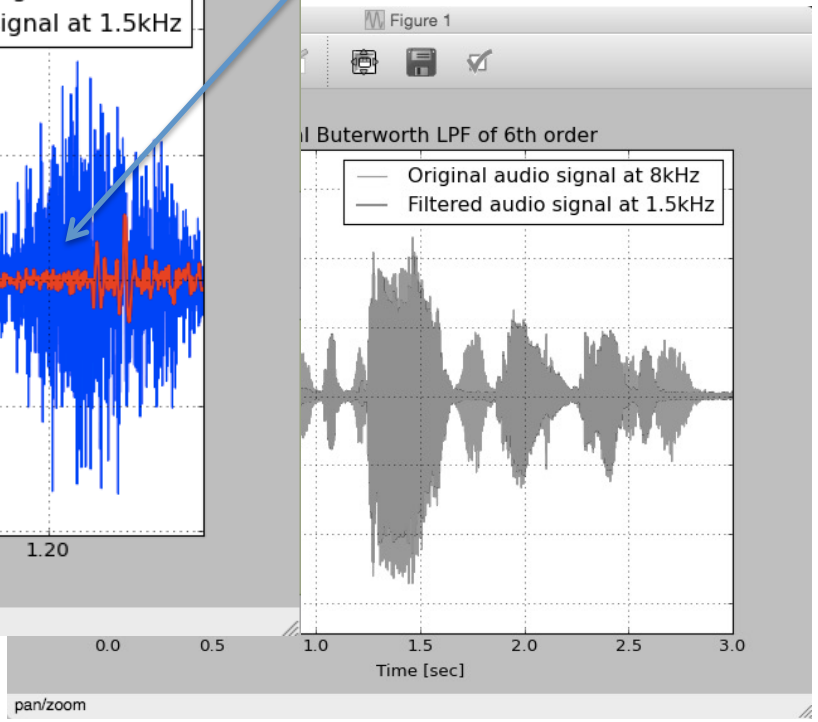
Signal Processing

- 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 impor
186 from pylab import plot,
187
188 def butterworth_lpf(dat
189     nyqist = fs / 2
190     cutoff_freq = cutof
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 origina
208 plot(t, data, 'b-', lab
209 plot(t, x, 'r-', linewi
210 title('Digital Buterwor
211 xlabel('Time [sec]'), y
212 show()
213
214 # Save the filtered and
215 write('files/lecture10/cheese_lpf.wav',Fs,x*.0001)
```



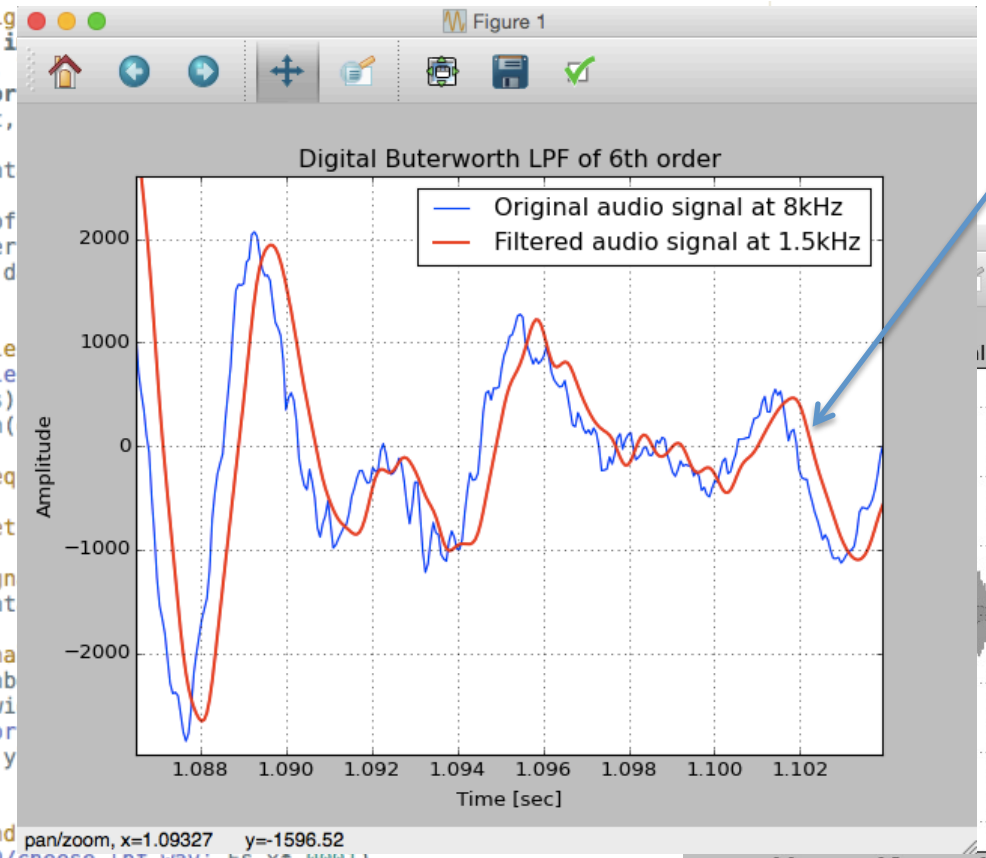
Observe all frequencies above 1.5kHz are filtered out – red signal is lower at the noisy part to the right



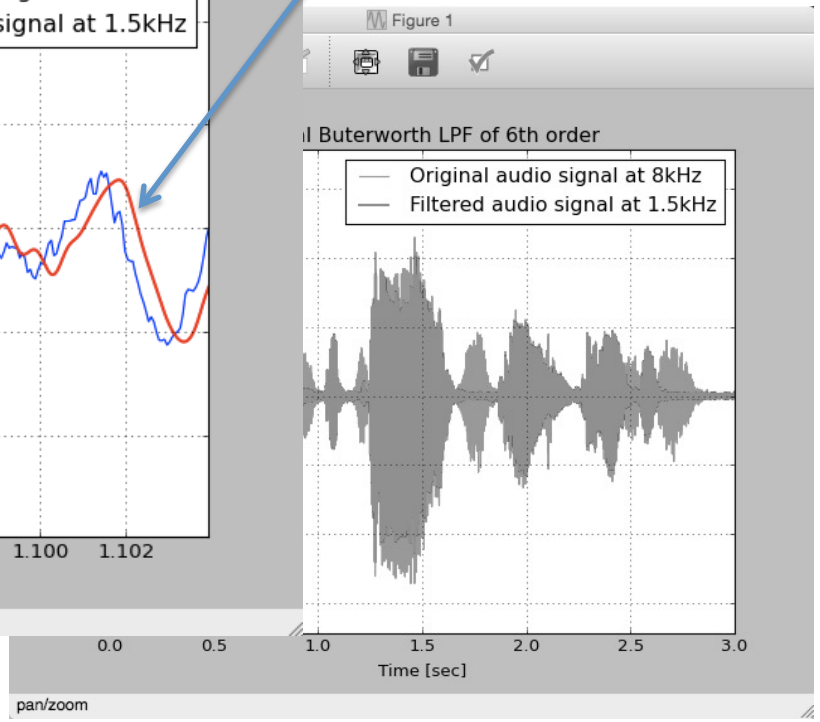
Signal Processing

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

```
# Real example of a Dig
from scipy.io.wavfile i
from numpy import int,
from scipy.signal impor
from pylab import plot,

def butterworth_lpf(dat
    nyqist = fs / 2
    cutoff_freq = cutof
    b, a = butter(order
    x = lfilter(b, a, d
    return x

# We load an audio file
(Fs, data) = read('file
T = int(len(data) / Fs)
t = linspace(0, T, len(

# We specify filter req
order = 6
cutoff = 1500  # we set

# Filter the audio sign
x = butterworth_lpf(dat

# Plot both the origina
plot(t, data, 'b-', lab
plot(t, x, 'r-', linewi
title('Digital Buterwor
xlabel('Time [sec]'), y
show()

# Save the filtered and
write('files/lecture10/cheese_lpf.wav',Fs,x*.0001)
```



Observe how the filtering process introduced a delay in the filtered signal



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

Special request topics

- **Audio data encoding**
- **Reading from Excel files**
- **Python and MySQL Database**

Audio data encoding

- Talk about audio manipulation …

Reading from Excel files

- Look at the code directly …

Python and MySQL Database

- Look at some examples …

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

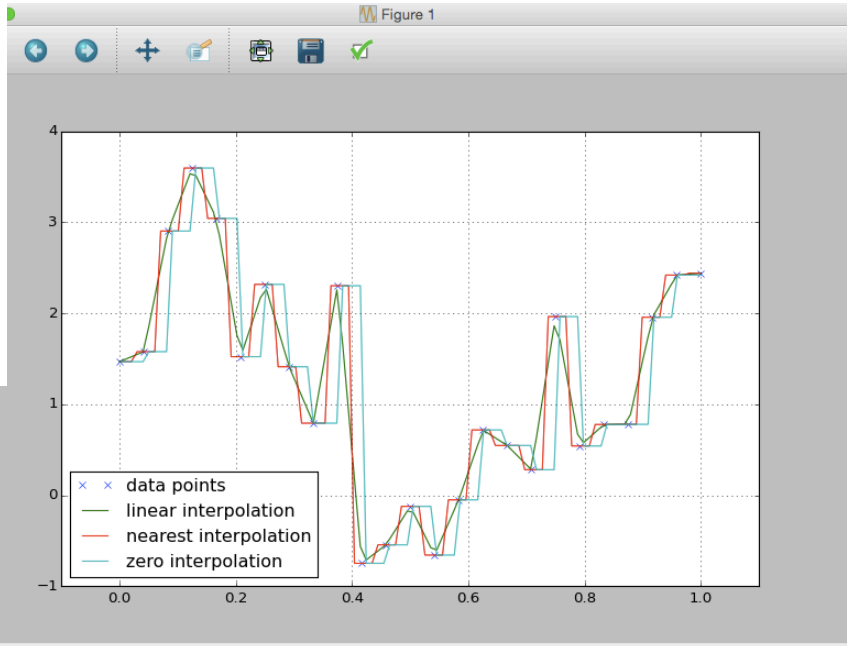
Interpolation

- 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:

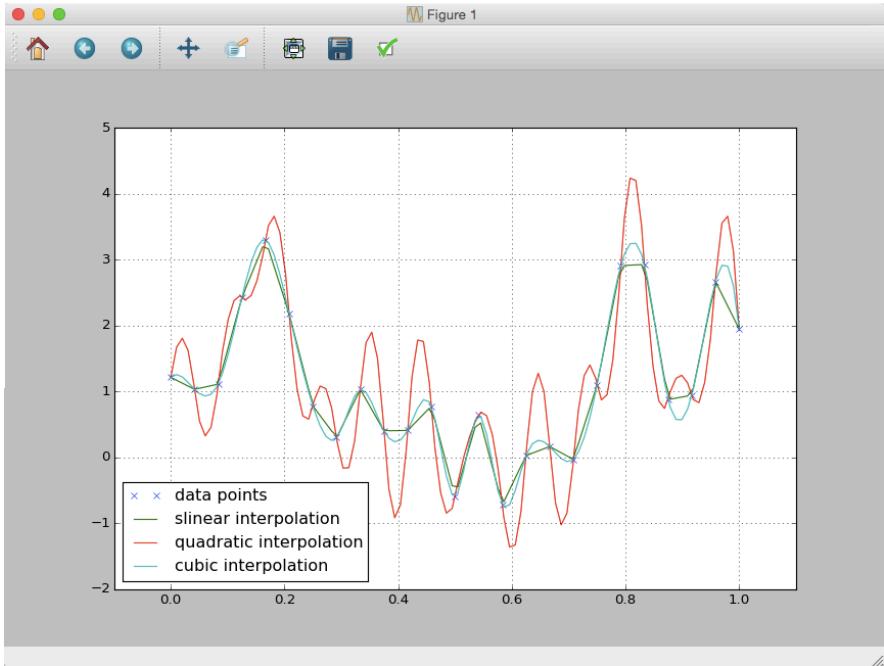
```
## Scipy 2/3

# Interpolation example:
from numpy import linspace, random, cos, pi
from scipy.interpolate import interp1d
import pylab as plb

# We first create the set of data points over which to interpolate:
data_span = linspace(0, 1, 25)
noise = (random.random(25)*3 + 0.5) / 2
data_points = cos(2 * pi * data_span) + noise**2
t_calculated = linspace(0, 1, 100)

# We build the 'linear' interpolation function like this:
linear_i = interp1d(data_span, data_points) # no 'kind' needed as 'linear' is default
linear = linear_i(t_calculated)

# This is how a 'nearest' interpolation is obtained - using the 'kind' property:
near_i = interp1d(data_span, data_points, kind='nearest')
near = near_i(t_calculated)

# This is how a 'zero' interpolation is obtained - using the 'kind' property:
zero_i = interp1d(data_span, data_points, kind='zero')
zero = zero_i(t_calculated)

# Now lets plot them all:
plb.plot(data_span, data_points, 'x', ms=5, label='data points')
plb.plot(t_calculated, linear, label='linear interpolation')
plb.plot(t_calculated, near, label='nearest interpolation')
plb.plot(t_calculated, zero, label='zero interpolation')

plb.legend(loc='lower left')
plb.grid(True), plb.xlim(-0.1,1.1)
```



Interpolation

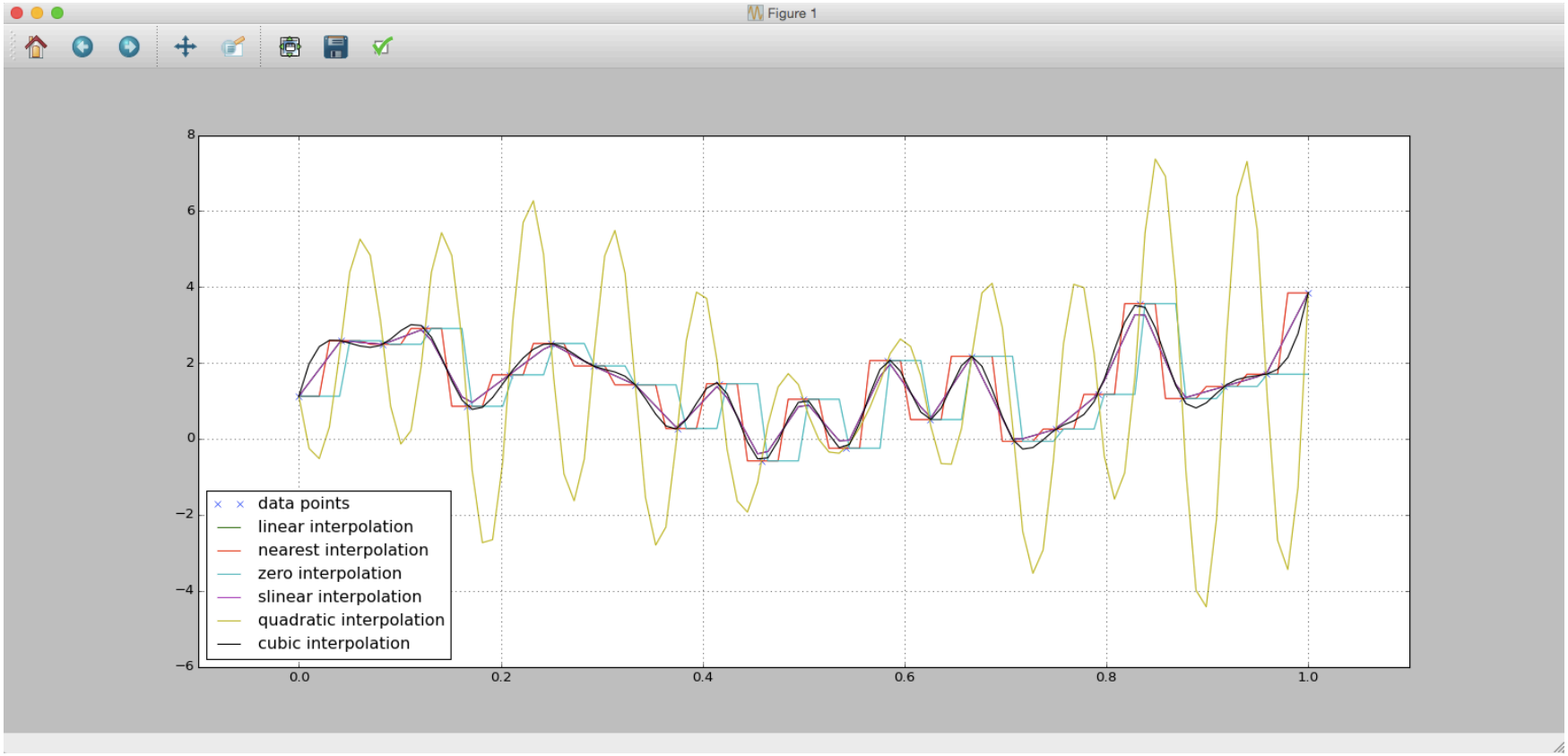
- Interpolation – Example using scipy.interpolate.interp1d:

```
## Scipy 2/3

# Interpolation example:
from numpy import linspace, random, cos, pi
from scipy.interpolate import interp1d
import pylab as plb

# We first create the set of data points over which to interpolate:
data_span = linspace(0, 1, 25)
noise = (random.random(25)*3 + 0.5) / 2
data_points = cos(2 * pi * data_span) + noise**2
t_calculated = linspace(0, 1, 100)

# This is how a 'slinear' interpolation is obtained - using the 'kind' property:
slinear_i = interp1d(data_span, data_points, kind='slinear')
slinear = slinear_i(t_calculated)

# This is how a 'quadratic' interpolation is obtained - using the 'kind' property:
quadratic_i = interp1d(data_span, data_points, kind='quadratic')
quadratic = quadratic_i(t_calculated)

# This is how a 'cubic' interpolation is obtained - using the 'kind' property:
cubic_i = interp1d(data_span, data_points, kind='cubic')
cubic = cubic_i(t_calculated)

# Now lets plot them all:
plb.plot(data_span, data_points, 'x', ms=5, label='data points')
plb.plot(t_calculated, slinear, label='slinear interpolation')
plb.plot(t_calculated, quadratic, label='quadratic interpolation')
plb.plot(t_calculated, cubic, label='cubic interpolation')

plb.legend(loc='lower left')
plb.grid(True), plb.xlim(-0.1,1.1)
```



Interpolation

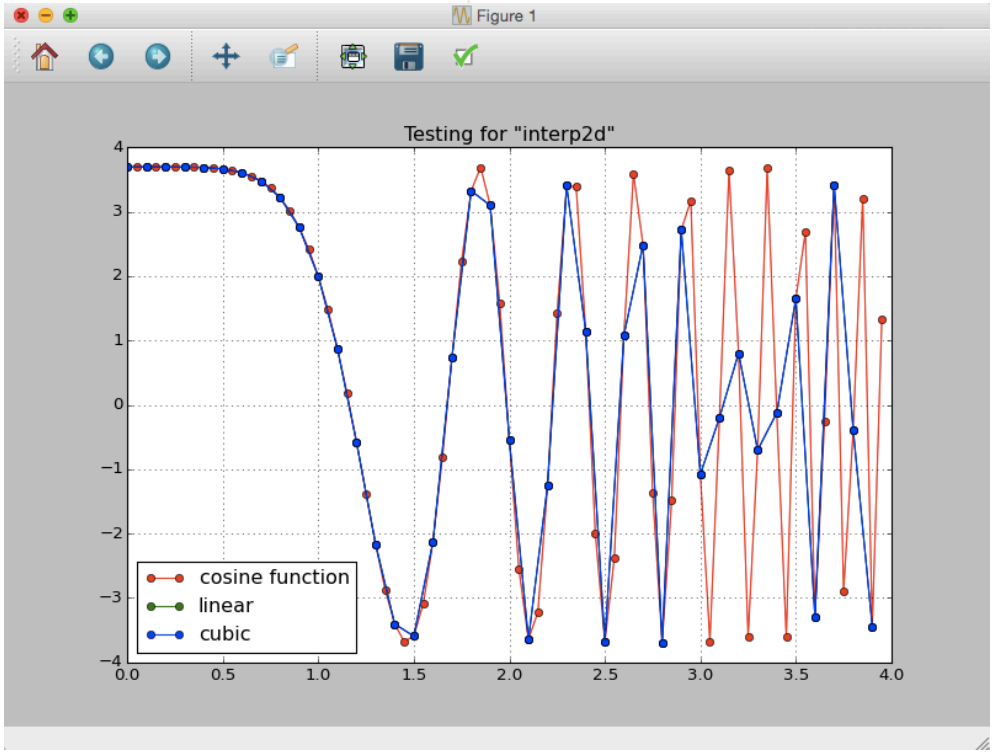
- Interpolation – Example using scipy.interpolate.interp1d:



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(a, 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')
```

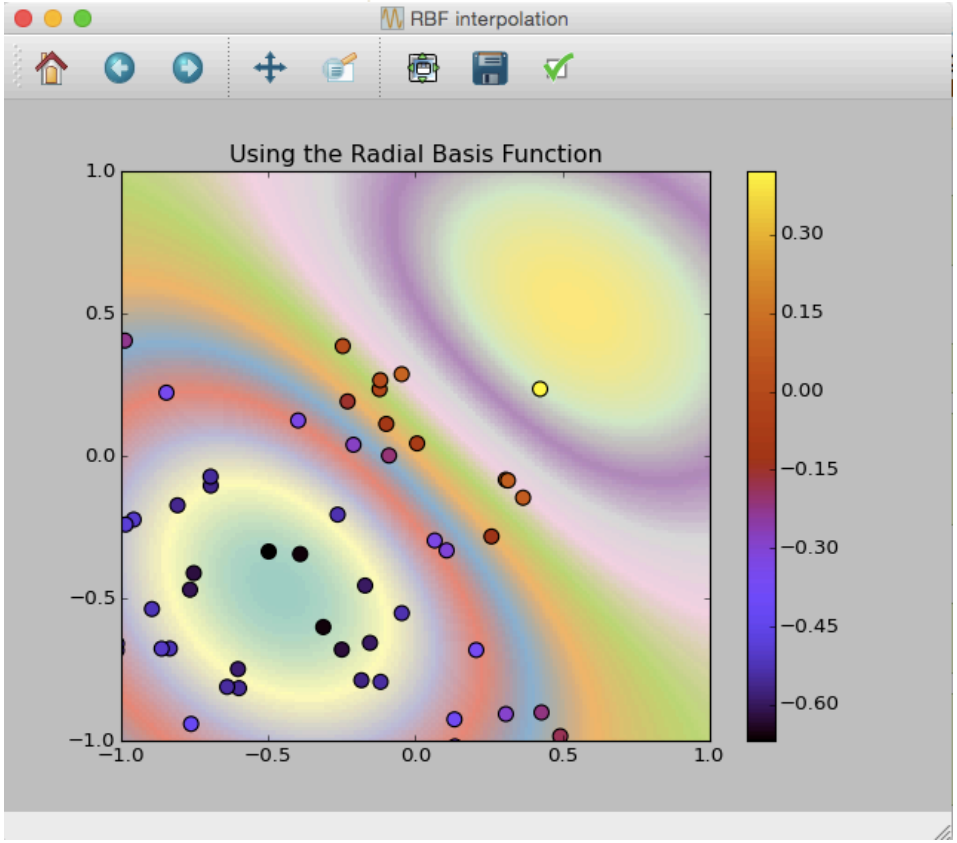


- note that the two kinds of interpolations allowed in interp2d overlap since they perform in a similar fashion in this example

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 plb
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  plb.figure('RBF interpolation')
69  plb.title('Using the Radial Basis Function')
70  plb.pcolor(ai, bi, ci, cmap='Set3')
71  plb.scatter(a, b, pts, c, cmap='gnuplot')
72  plb.xlim(-1, 1), plb.ylim(-1, 1)
73  plb.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

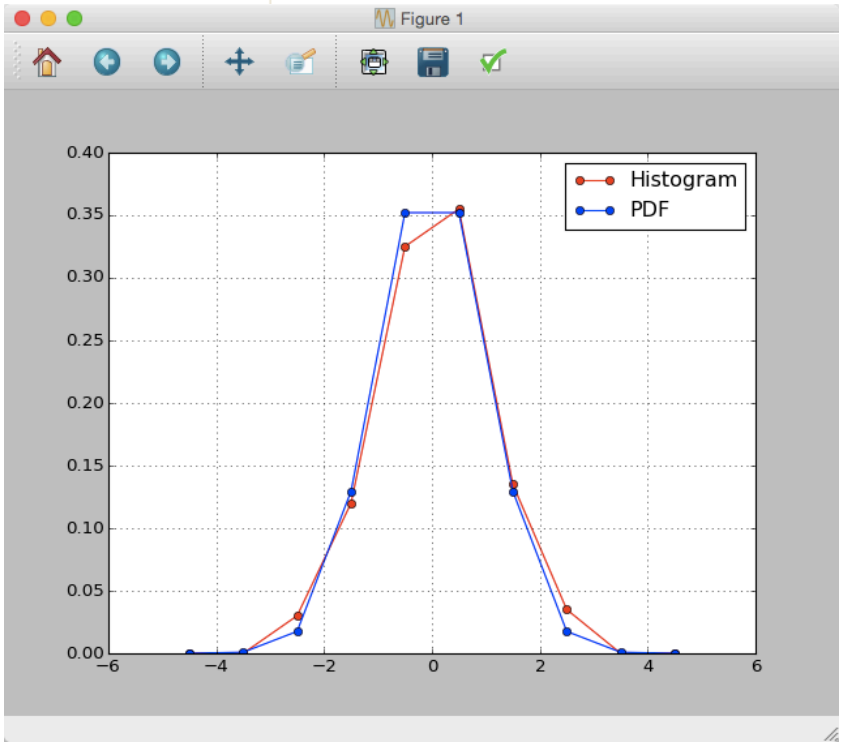
- Statistics – Random Variables (RV)
  - Random Variables are represented in two main classes:
    - continuous random variables
    - discrete random variables
  - there are over 80 continuous random variables and 10 discrete random variables
  - common Random Variables methods are:
    - pdf: Probability 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. $X <= x$ *(distribution function X is evaluated at x)*
    - rvs: Random Variates
    - 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

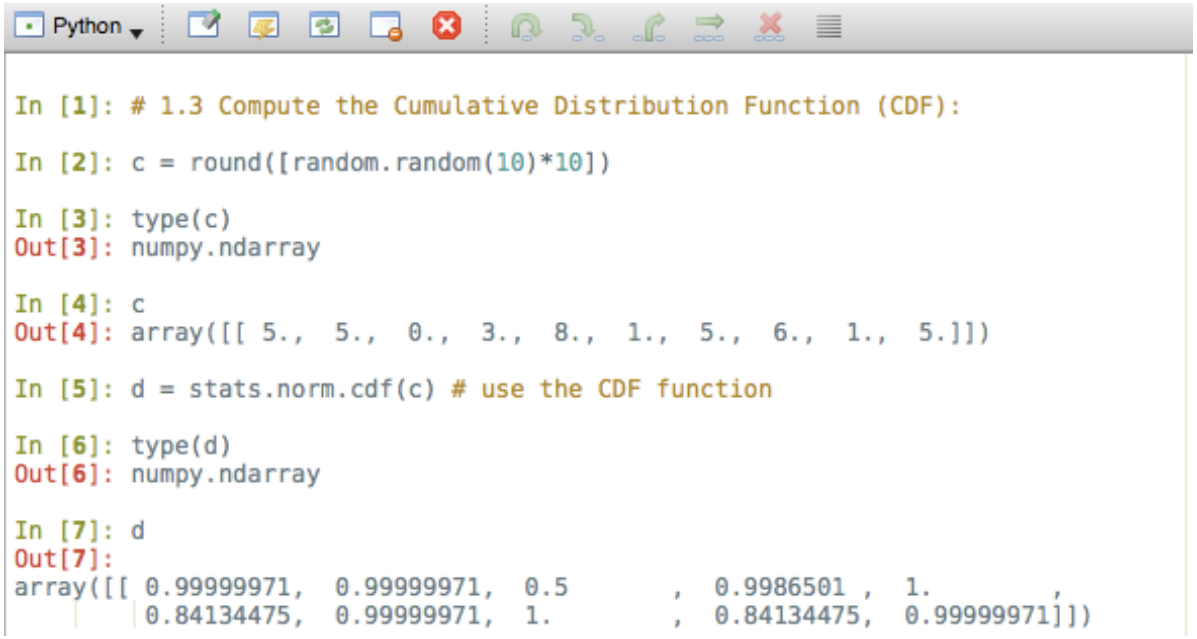
```
122 # Statistics:
123 # 1. Random Variables:
124 from numpy import array, arange, random, histogram
125 from scipy import stats
126 from pylab import plot, grid, legend
127
128 # Lets create a NumPy array first:
129 a = random.normal(size=200)
130 x = arange(-5, 6)
131 y = 0.5*(x[1:] + x[:-1])
132
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= 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:



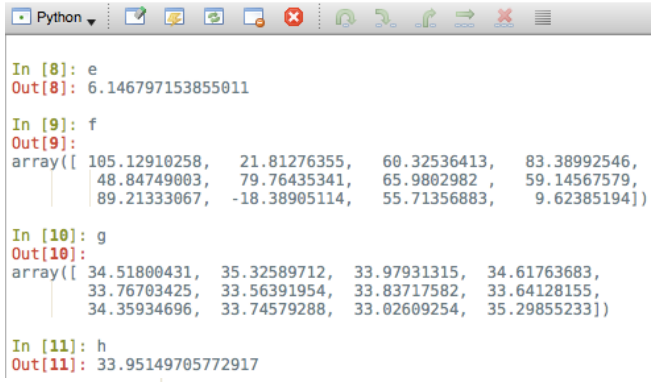
Statistics

- 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

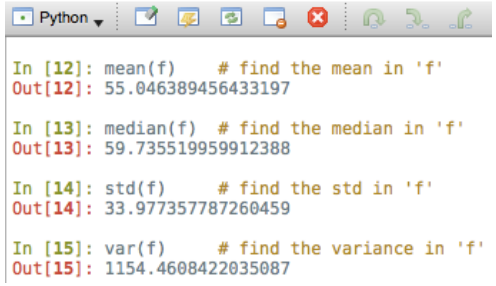
Example:

```
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 sintax will generate a different result:
175 h = stats.norm.rvs(34)
176 h
```

where:

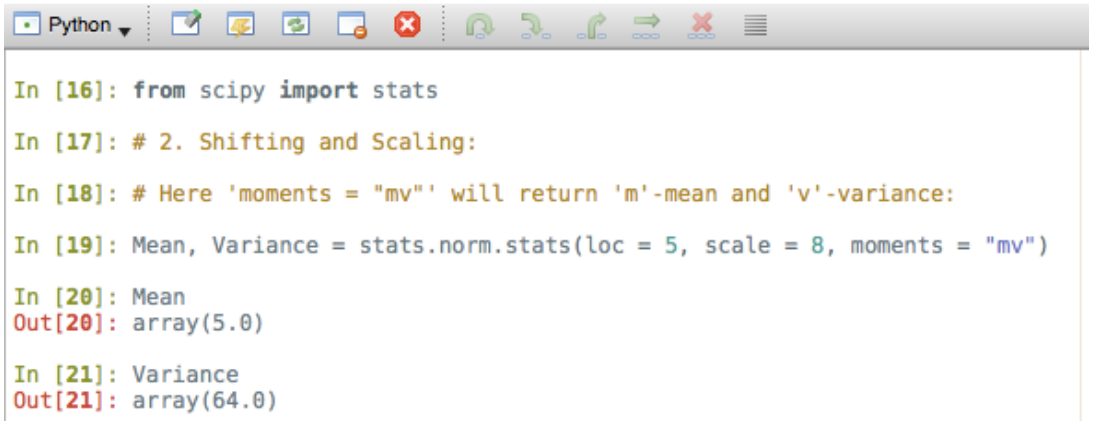


and:



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

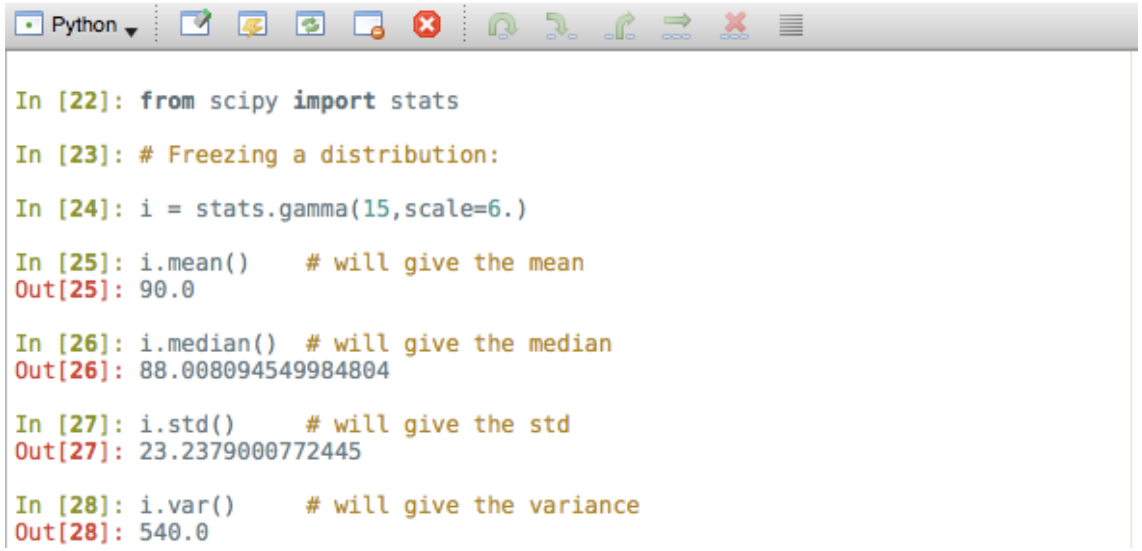


  - the standardized distribution for a random variable X equals (X - loc) / scale, hence the default values for loc and scale are respectively 0 and 1
  - so the output of variable h in the RV example of the previous slide stats.norm.rvs(34) generates a normally distributed random variate with mean loc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



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
    - nnlf: 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:

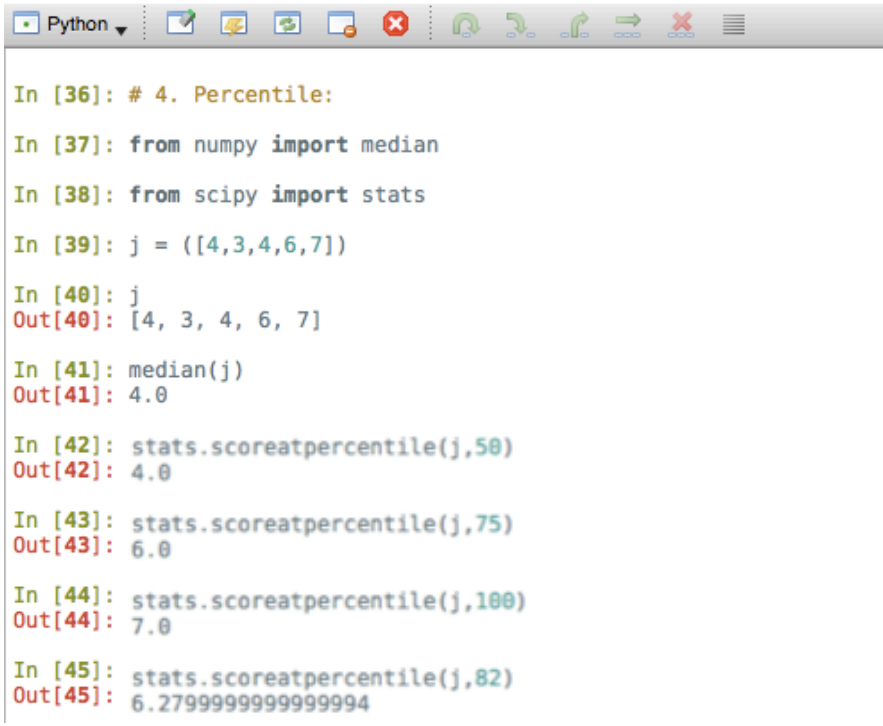
```
Python

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
```

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

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:



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

source: http://docs.scipy.org

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

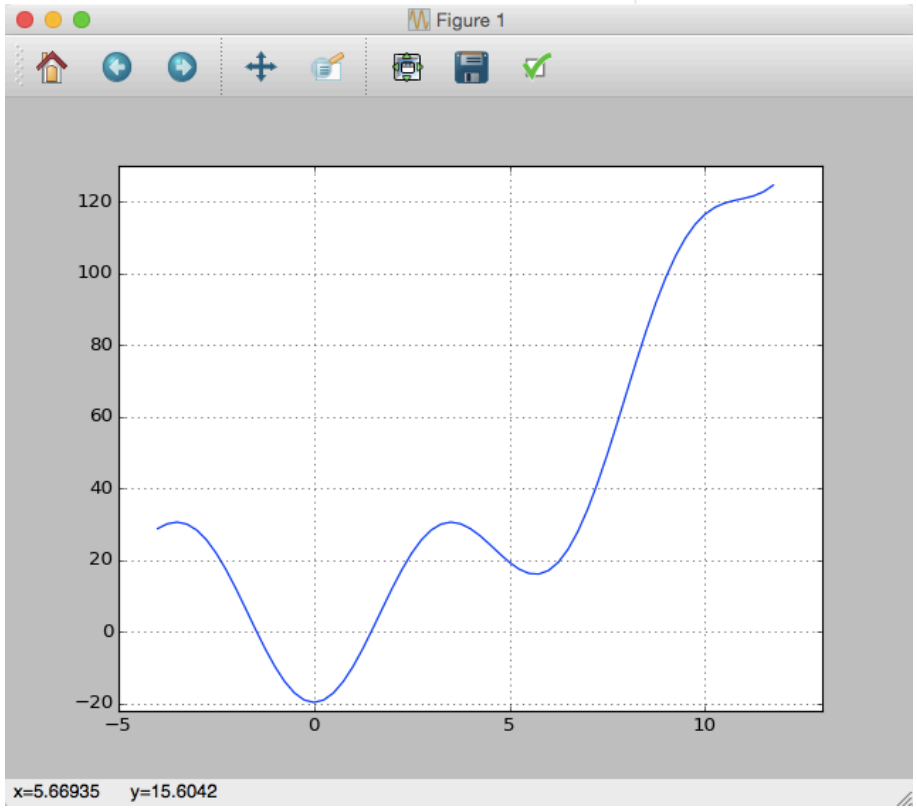
source: http://docs.scipy.org

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

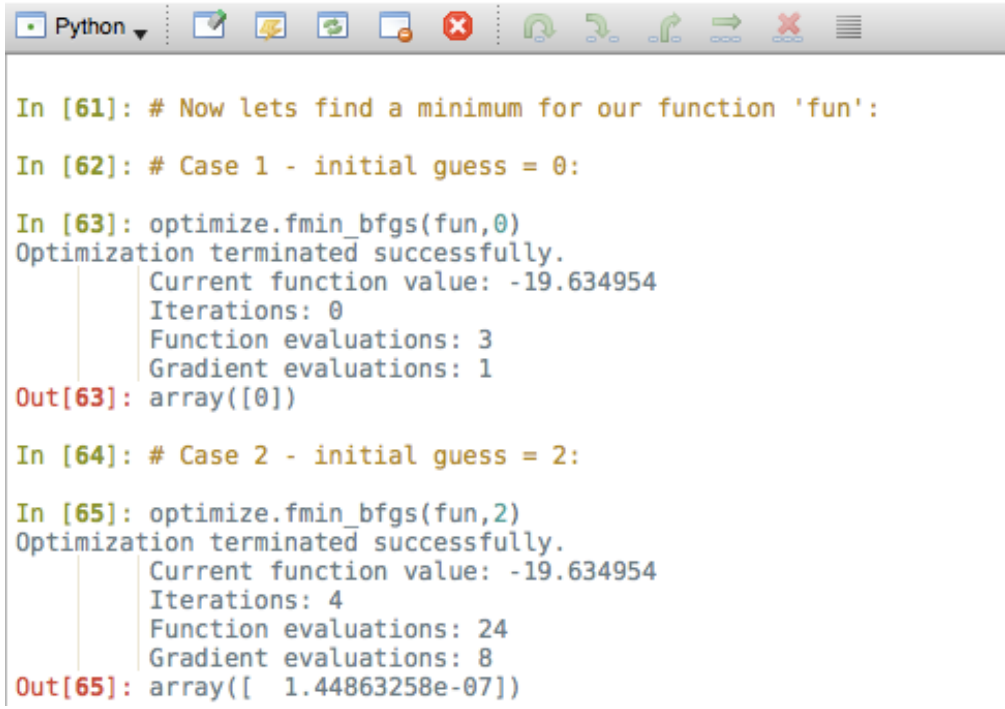
  Example:

  – 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



Optimization

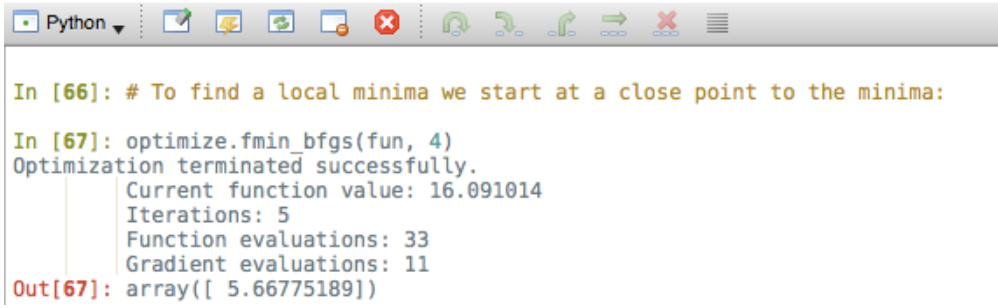
- Optimization

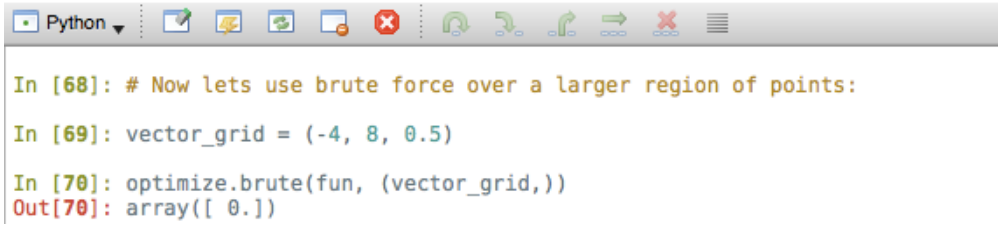
  Example:

  – 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



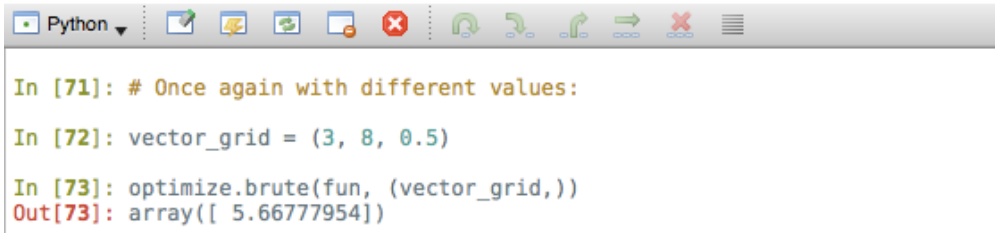


Optimization

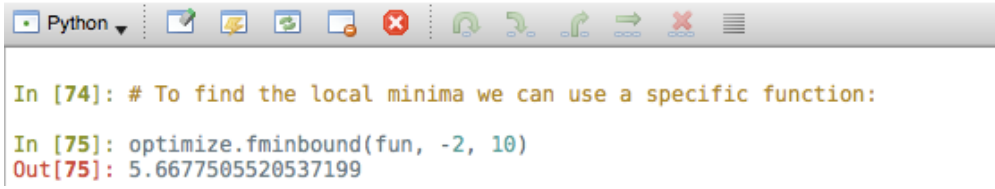
- Optimization

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

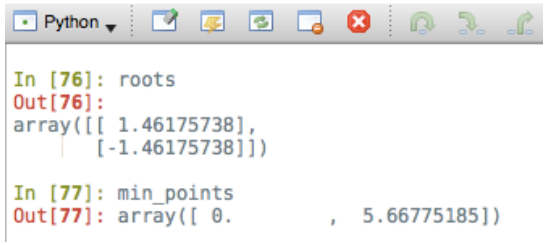


Optimization

- 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:

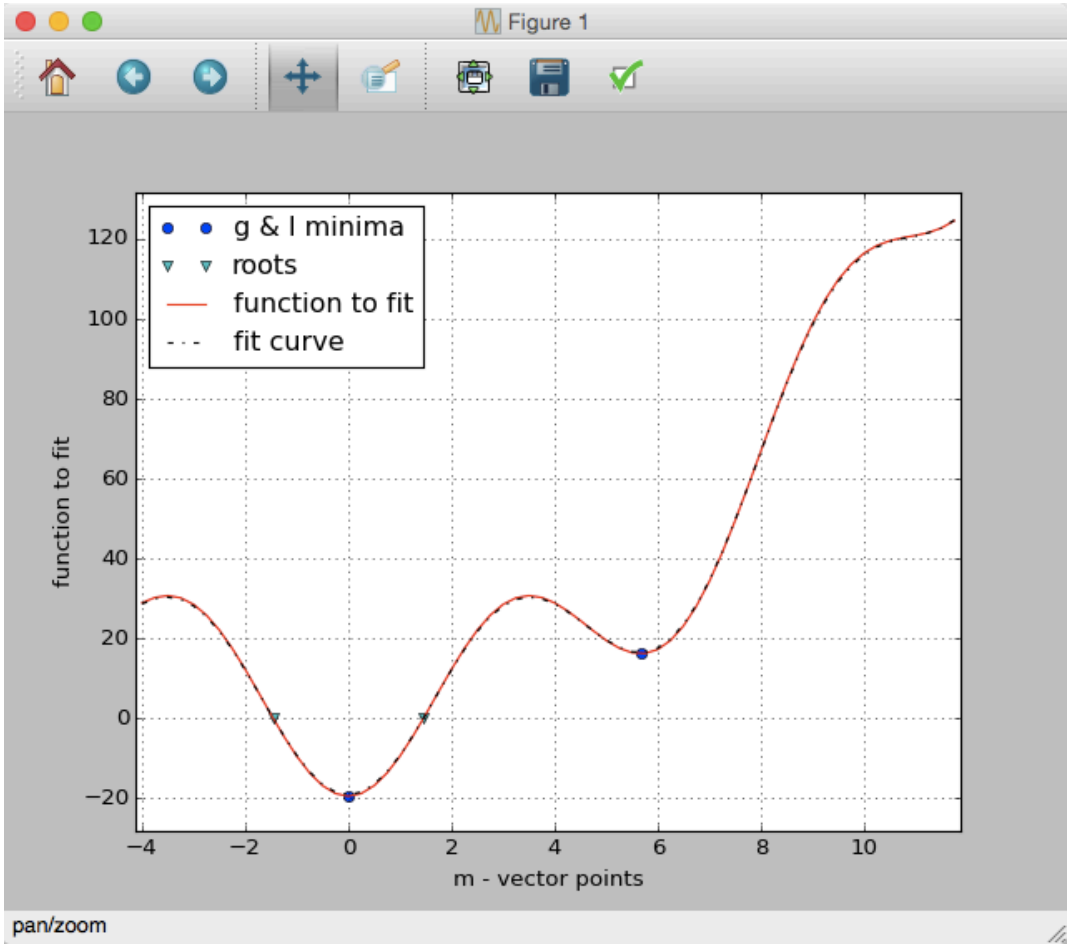



```
267 # Curve fitting:
268 from numpy import linspace, arange, array, random, cos, pi
269 from scipy import optimize
270 from pylab import plot, grid, xlabel, ylabel, legend, grid
271
272 # Here we create the initial curve:
273 def fun1(r):
274     n = r**2 - 6.25*cos(r)*pi
275     return n
276
277 # 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 'fun2' 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:

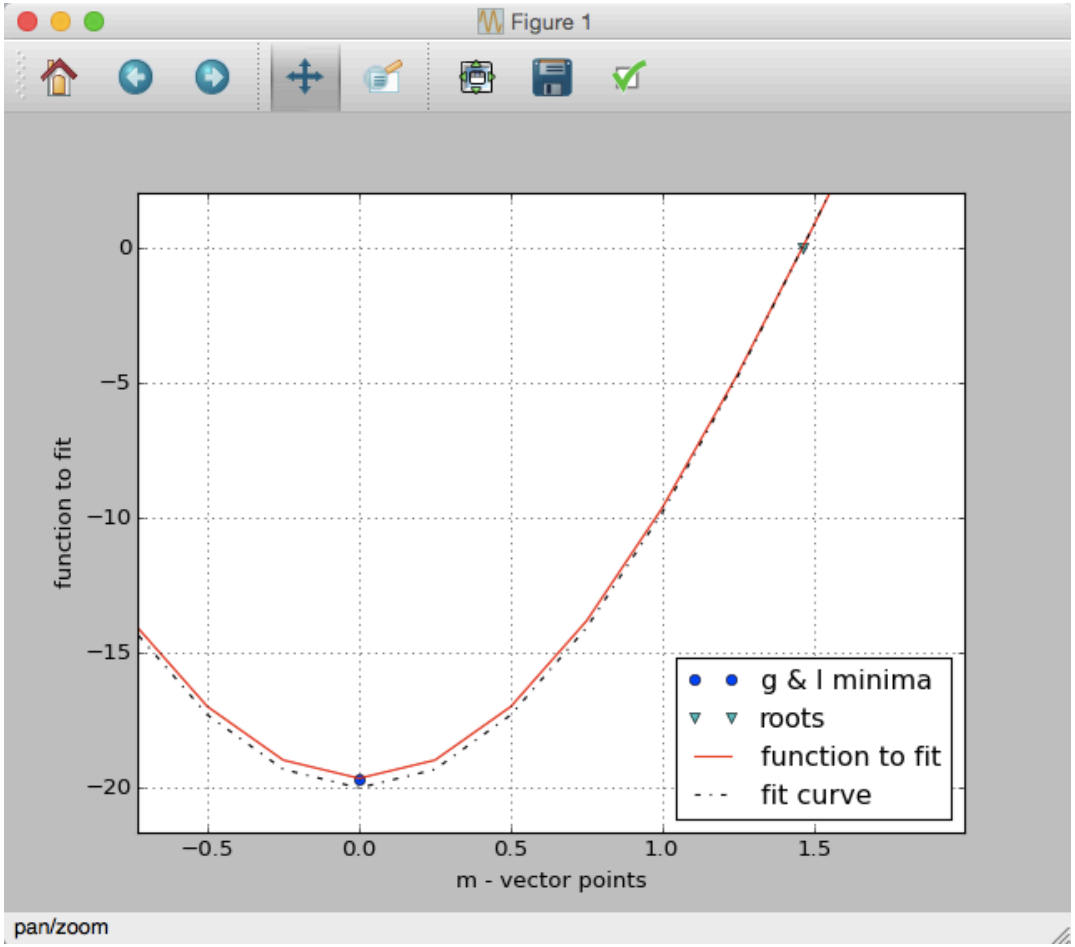


Roots ->

Optimization

- Optimization – curve fitting

Example:

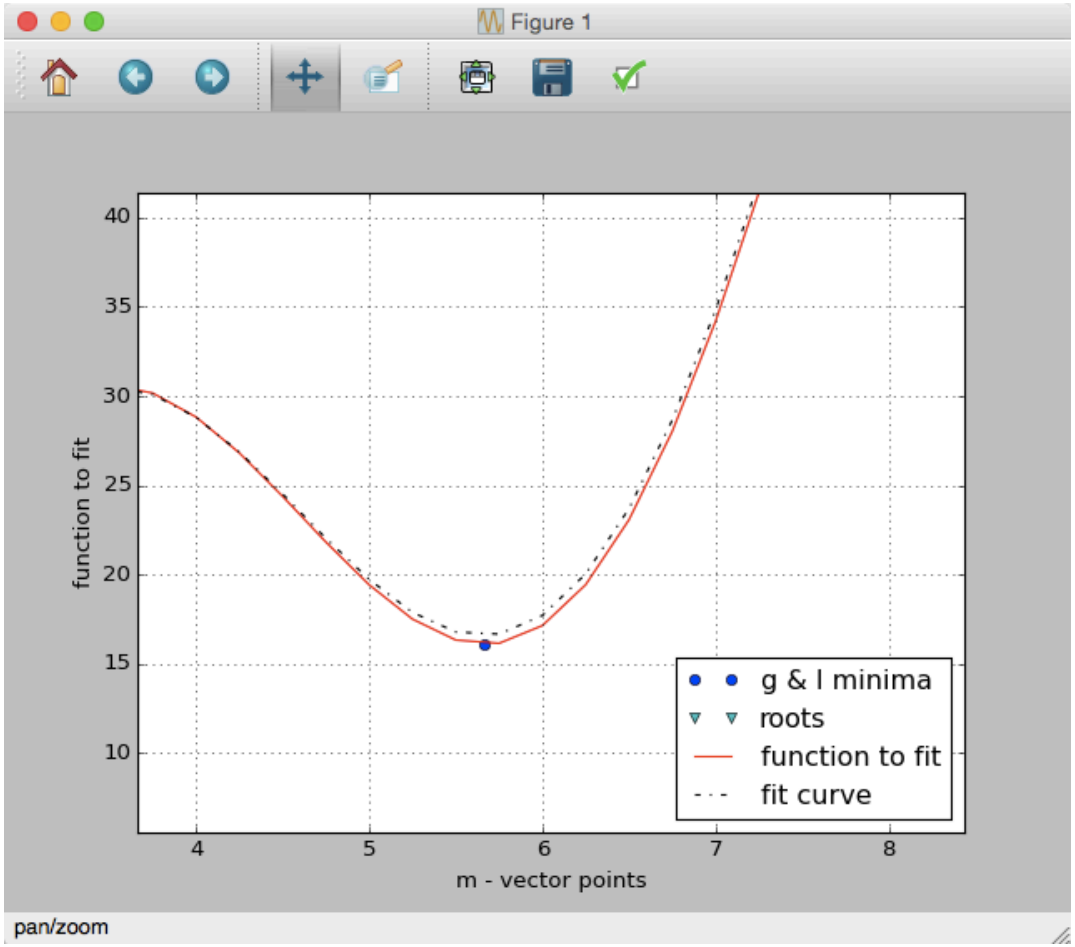


Global minima ->

Optimization

- Optimization – curve fitting

Example:



Local minima ->

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, IPyton 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

- **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: .xls, .xlsx, mysql