Python Advance Course via Astronomy street

Sérgio Sousa (IA)
ExoEarths Team (http://www.astro.up.pt/exoearths/)

Python Advance Course via Astronomy street

Advance Course Outline:

- Lesson 1: Python basics (T + P)
- Lesson 2: Python with Numpy (T + P)
- Lesson 3: Matplotlib, Science and Astronomy modules (T + P)

python

Python Advance Course via Astronomy street

Lesson 3: Python with Matplotlib, Scipy, Astropy, PyAstronomy

- Plotting with Matplotlib
- Using Scipy
- Astropy, PyAstronomy, ...

Matplotlib

```
>>> import matplotlib
>>> help(matplotlib)
```

```
Help on package matplotlib:

NAME
    matplotlib - This is an object-oriented plotting library.

FILE
    /usr/lib/pymodules/python2.7/matplotlib/__init__.py

DESCRIPTION
    A procedural interface is provided by the companion pyplot module,
    which may be imported directly, e.g::

        from matplotlib.pyplot import *

    To include numpy functions too, use::

        from pylab import *

    or using ipython::

        ipython -pylab

    For the most part, direct use of the object-oriented library is
    encouraged when programming; pyplot is primarily for working
    interactively.  The
    exceptions are the pyplot commands :func:`~matplotlib.pyplot.figure`,
    :func:`~matplotlib.pyplot.subplot`,
    :func:`~matplotlib.pyplot.subplots`,
```

Using Matplotlib

Simple plot – using procedural interface (pyplot)

```
>>> import numpy as np  #It will be useful to deal with data
>>> import matplotlib.pyplot as pl  #procedural interface
>>>
>>> #simple plot
...
>>> X = np.linspace(-np.pi, np.pi, 256, endpoint=True)
>>> C, S = np.cos(X), np.sin(X)
>>> pl.plot(X, C)
[<matplotlib.lines.Line2D object at 0x2c4f410>]
>>> pl.plot(X, S)
[<matplotlib.lines.Line2D object at 0x350d490>]
>>>
>>> pl.show()
```

numpy useful to deal with data arrays

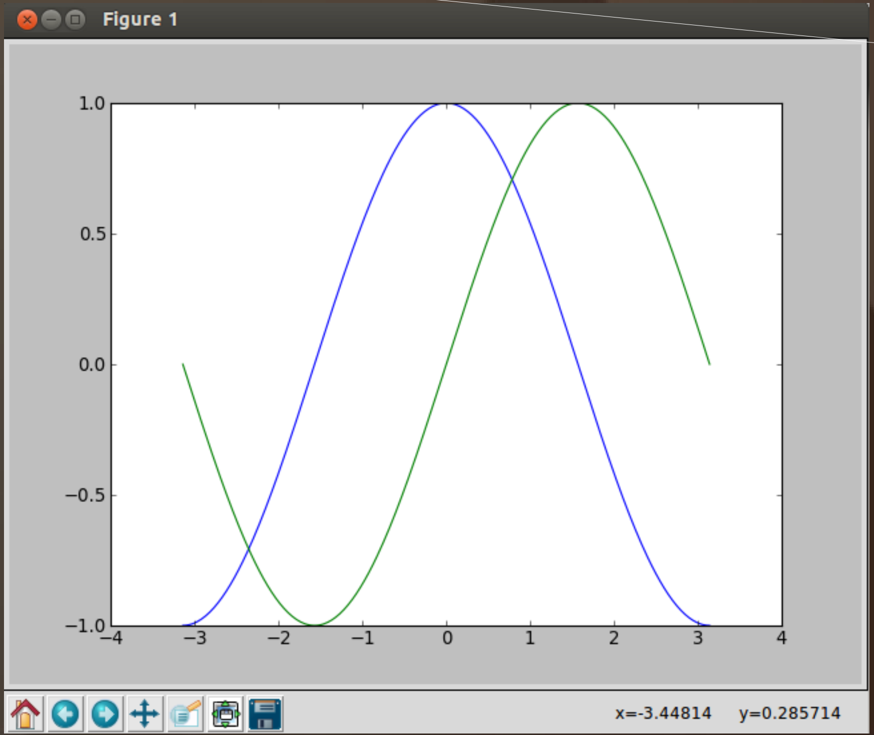
Pyplot – the module to “ignore” objects

Creation of data (x, cos(x)) and (x, sin(x))

Plot each set of data

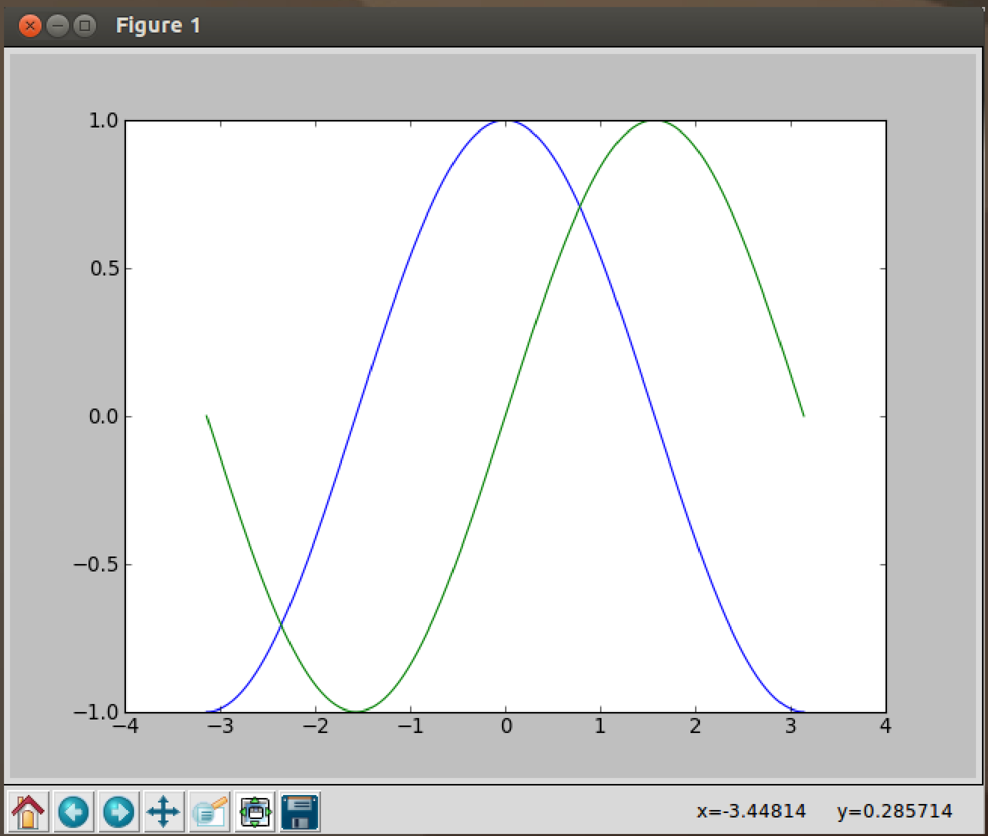
– note that objects are still created and in memory...

Show comand to open the plot window (Freezing the Python interpreter)

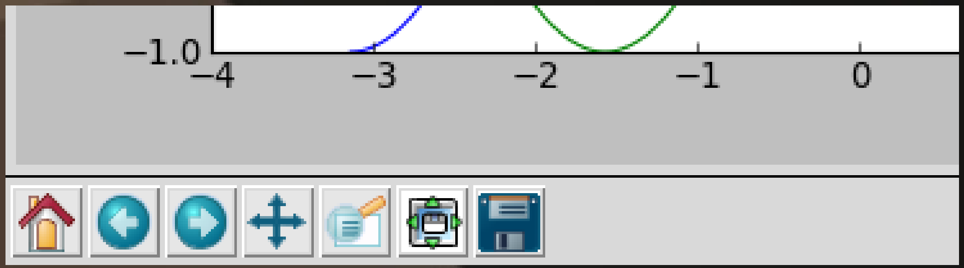


Using Matplotlib

The Plotting window - Figure



The plot window has some nice interactive features that you can easily explore



 Show original plot

 Undo/Redo visualization

 Navigation in the plot

 Zoom Rectangle

 Customize Subplots

 Saving/Exporting Figure

Using Matplotlib

Simple plot – changing default settings – procedural interface

```
# changing some default parameters

# Create a figure of size 8x6 points, 80 dots per inch
pl.figure(figsize=(8, 6), dpi=80)

# Create a new subplot from a grid of 1x1
pl.subplot(1, 1, 1)

X = np.linspace(-np.pi, np.pi, 256, endpoint=True)
C, S = np.cos(X), np.sin(X)

# Plot cosine with a blue continuous line of width 1 (pixels)
pl.plot(X, C, color="blue", linewidth=1.0, linestyle="-")

# Plot sine with a green continuous line of width 1 (pixels)
pl.plot(X, S, color="green", linewidth=1.0, linestyle="-")

# Set x limits
pl.xlim(-4.0, 4.0)

# Set x ticks
pl.xticks(np.linspace(-4, 4, 9, endpoint=True))

# Set y limits
pl.ylim(-1.0, 1.0)

# Set y ticks
pl.yticks(np.linspace(-1, 1, 5, endpoint=True))

# Save figure using 72 dots per inch
# savefig("exercice_2.png", dpi=72)

# Show result on screen
pl.show()
```

Defining the figure

Defining a subplot of the figure

Creation of data
(x, cos(x)) and (x, sin(x))

Plot data, setting color, linewidth and linestyle

Defining x axis limits

Difining x axis thicks

Defining y axis limits

Difining y axis thicks

Possibility to save Figure as .png file

Show comand to open the plot window
(Freezing the Python interpreter)

Using Matplotlib

Ploting lines – linestyles and markers :: >>> help(pl.plot)

```
The following format string characters are accepted to control
the line style or marker:

================    ===============================
character           description
================    ===============================
``'-'``             solid line style
``'--'``            dashed line style
``'-.'``            dash-dot line style
``':'``             dotted line style
``'.'``             point marker
``','``             pixel marker
``'o'``             circle marker
``'v'``             triangle_down marker
``'^'``             triangle_up marker
``'<'``             triangle_left marker
``'>'``             triangle_right marker
``'1'``             tri_down marker
``'2'``             tri_up marker
``'3'``             tri_left marker
``'4'``             tri_right marker
``'s'``             square marker
``'p'``             pentagon marker
``'*'``             star marker
``'h'``             hexagon1 marker
``'H'``             hexagon2 marker
``'+'``             plus marker
``'x'``             x marker
``'D'``             diamond marker
``'d'``             thin_diamond marker
``'|'``             vline marker
``'_'``             hline marker
================    ===============================
```

Using Matplotlib

Ploting lines – colors :: >>> help(pl.plot)

```
The following color abbreviations are supported:

==========  ========
character   color
==========  ========
'b'         blue
'g'         green
'r'         red
'c'         cyan
'm'         magenta
'y'         yellow
'k'         black
'w'         white
==========  ========

In addition, you can specify colors in many weird and
wonderful ways, including full names (``'green'``), hex
strings (``'#008000'``), RGB or RGBA tuples (``(0,1,0,1)``) or
grayscale intensities as a string (``'0.8'``).  Of these, the
string specifications can be used in place of a ``fmt`` group,
but the tuple forms can be used only as ``kwargs``.
```

Example of using RGB to get a blue color:

```
>>> pl.plot(X, C, color="#0000CC", linewidth=3.0, linestyle="-.")
[<matplotlib.lines.Line2D object at 0x3537350>]
>>> pl.show()
```

Using Matplotlib

Simple plot with legend – using procedural interface (pyplot)

```
>>> #simple plot with legend
... import numpy as np  #It will be useful to deal with data
>>> import matplotlib.pyplot as pl  #procedural interface
>>>
>>> X = np.linspace(-np.pi, np.pi, 256, endpoint=True)
>>> C, S = np.cos(X), np.sin(X)
>>> pl.plot(X, C, label=r'$\cos(\theta)$')
[<matplotlib.lines.Line2D object at 0x40e2690>]
>>> pl.plot(X, S, label=r'$\sin(\theta)$')
[<matplotlib.lines.Line2D object at 0x3442550>]
>>> pl.legend(loc='upper left')
<matplotlib.legend.Legend object at 0x40e28d0>
>>>
>>> pl.show()
```

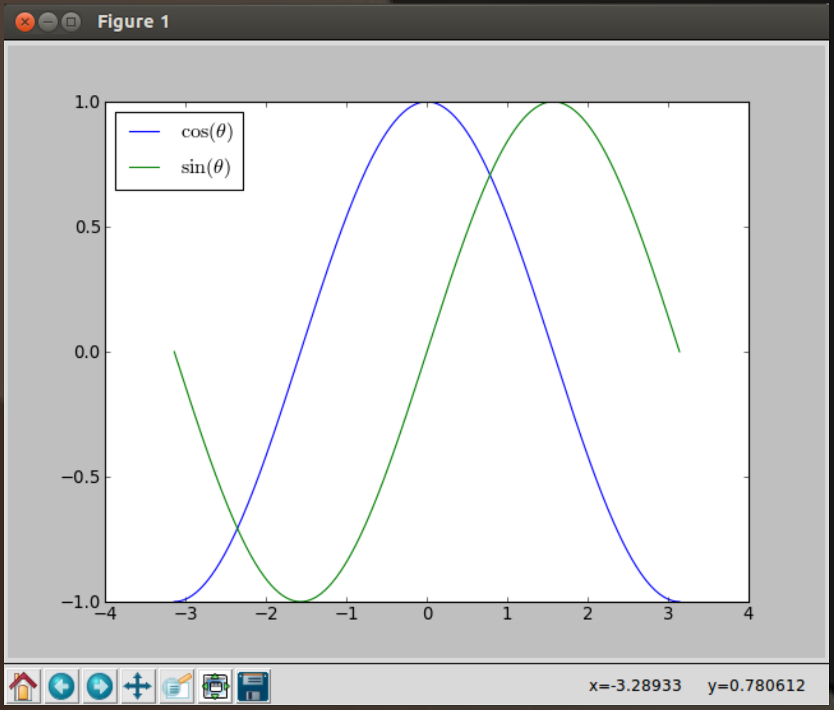
Simply using label in the plot.

You can use latex with the sintaxe:
r' $latex_keyword$'

Then you simple call legend()

You can use the variable loc to set the location of the legend in the plot

```
The location codes are

  ===============   =============
  Location String   Location Code
  ===============   =============
  'best'            0
  'upper right'     1
  'upper left'      2
  'lower left'      3
  'lower right'     4
  'right'           5
  'center left'     6
  'center right'    7
  'lower center'    8
  'upper center'    9
  'center'          10
  ===============   =============
```



Using Matplotlib

Objects in Matplotlib

The awareness of the objects will allow a proper control of the plots that you want to create.

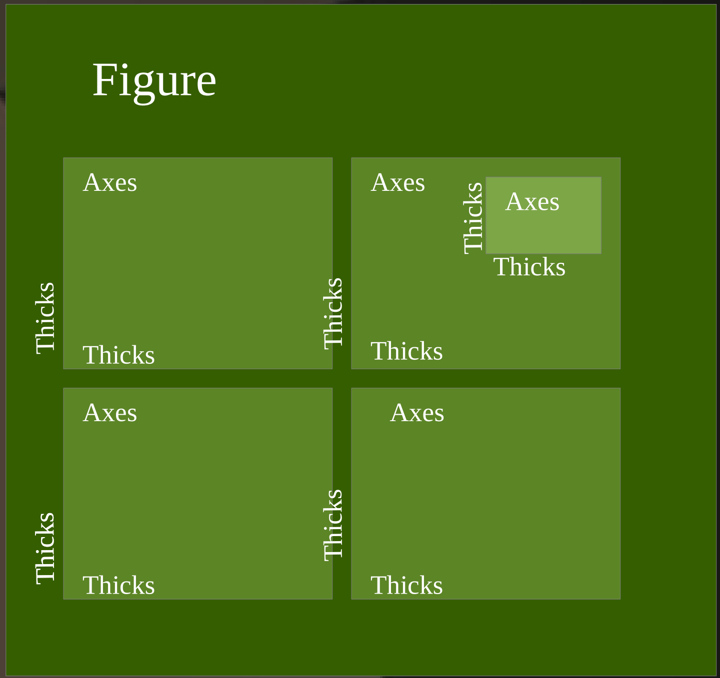
The main objects that you should know:

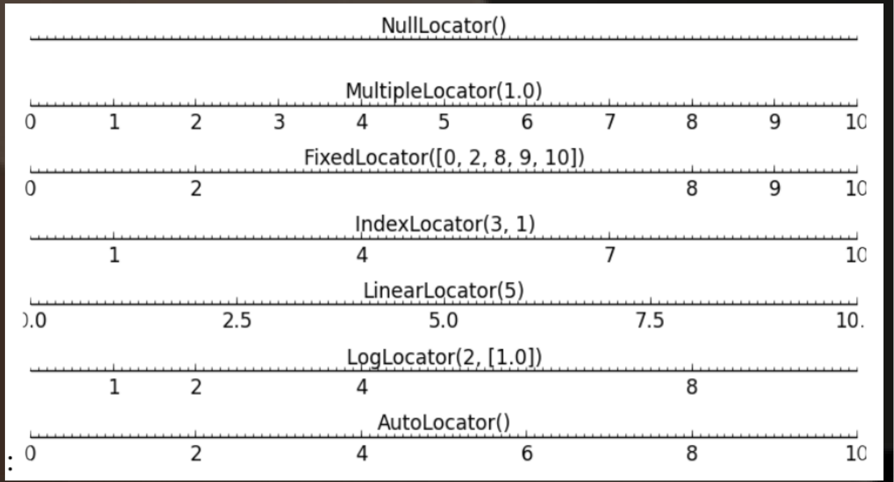
→**Figure:** A Figure in matplotlib corresponds to the whole window in the user interface.

→**Axes:** The axes is the object where you will draw your plot. You can freely control the position of the axes. Within the figure you can create/add axes.

→**Subplot:** This are actually a specific type of axes, where the positions are fixed on the figure. Within the figure you create subplots. You can create a single subplot, or a grid of subplots within the figure.

→**Thicks:** Are the objects that control each axe coordinate, uncluding the type of numbers, scale, intervals, etc...





Using Matplotlib

Objects in Matplotlib – An example

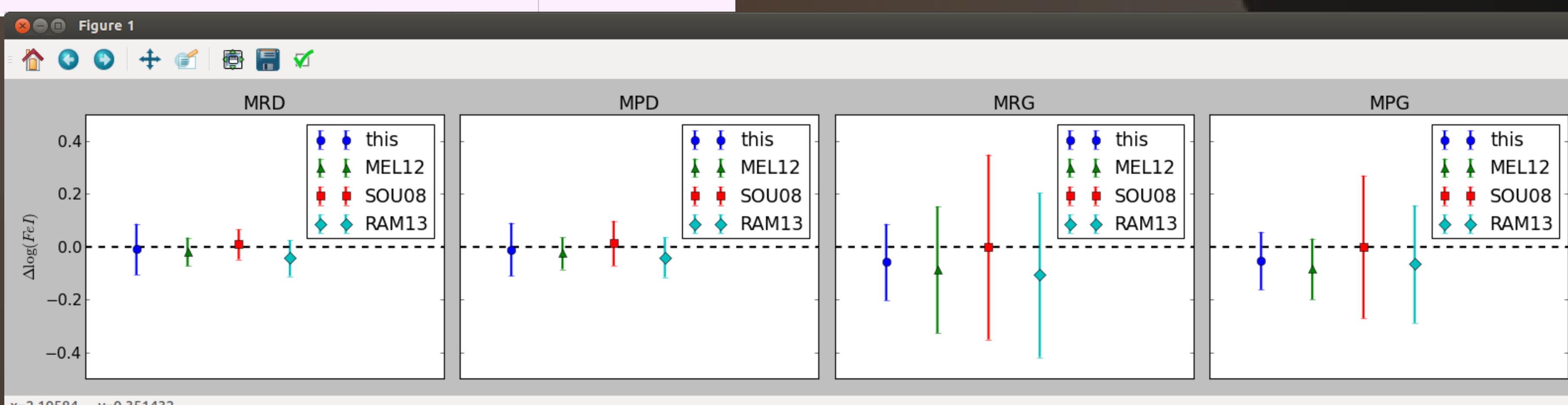
```python
def plot_ax_star(star,result,axe):
  linelists = [('_gesEW_mar',1,'this','o'),('_melendez_kur',2,'MEL12','^'),
               \('_mine_kur' ,3,'SOU08','s'),('_ramirez_mar' ,4,'RAM13','D')]
  axe.plot([0,10],[0,0], linestyle='--', color='k', linewidth=2.0)
  for linelist,i,labelstr,mark in linelists:
    idstr = star+linelist+'.moog'
    res = [ (file_r,star_r,nfei_r, dfei_r, sfei_r) \
            for (file_r,star_r,nfei_r, dfei_r, sfei_r) \
            in result if file_r == idstr]
    (file_r,star_r,nfei_r, dfei_r, sfei_r) = res[0]
    axe.errorbar([i], [dfei_r], yerr=sfei_r,fmt=mark, label=labelstr, linewidth=2.0, markersize=8)
  axe.set_xlim(0,7)
  axe.set_ylim(-0.5,0.5)
  axe.xaxis.set_visible(False)
  axe.set_title(star)
  axe.legend()


def plot_graphs(result):
  plt.rcParams.update({'font.size': 14})

  fig = plt.figure(figsize=(20, 4))
  ax1= fig.add_subplot(141)
  ax1.set_ylabel(r'$\Delta \log(FeI)$')
  ax2= fig.add_subplot(142,sharey=ax1)
  plt.setp(ax2.get_yticklabels(), visible=False)
  ax3= fig.add_subplot(143,sharey=ax1)
  plt.setp(ax3.get_yticklabels(), visible=False)
  ax4= fig.add_subplot(144,sharey=ax1)
  plt.setp(ax4.get_yticklabels(), visible=False)

  plot_ax_star('MRD',result,ax1)
  plot_ax_star('MPD',result,ax2)
  plot_ax_star('MRG',result,ax3)
  plot_ax_star('MPG',result,ax4)

#  fig.subplots_adjust(left=None, bottom=None, right=None, top=None, wspace=None, hspace=None)
  fig.tight_layout()

  plt.show()
```

Main objects:

→ 1 Figure
→ 4 Axes constructed with subplots
→ 8 Thicks automatically constructed

Extra stuff in this example:

→ Shared axes
→ Changing font size
→ Making axes invisible
→ Using latex in lables
→ Using different symbols and automatic colors
→ Automatic legends
→ Using error bars



Using Matplotlib

Objects in Matplotlib – An example

```
def plot_graphs(result):
    plt.rcParams.update({'font.size': 14})

    fig = plt.figure(figsize=(20, 4))
    ax1= fig.add_subplot(141)
    ax1.set_ylabel(r'$\Delta \log(FeI)$')
    ax2= fig.add_subplot(142,sharey=ax1)
    plt.setp(ax2.get_yticklabels(), visible=False)
    ax3= fig.add_subplot(143,sharey=ax1)
    plt.setp(ax3.get_yticklabels(), visible=False)
    ax4= fig.add_subplot(144,sharey=ax1)
    plt.setp(ax4.get_yticklabels(), visible=False)

    plot_ax_star('MRD',result,ax1)
    plot_ax_star('MPD',result,ax2)
    plot_ax_star('MRG',result,ax3)
    plot_ax_star('MPG',result,ax4)

#  fig.subplots_adjust(left=None, bottom=None, right
    fig.tight_layout()

    plt.show()
```

Main objects:

→ 1 Figure

→ 4 Axes constructed with subplots

Extra stuff in this example:

→ Changing font size

→ Using latex in lables

→ Shared axes

→ Making axes invisible

Calling a specific function to create each individual plot

Using Matplotlib

Objects in Matplotlib

Only difference is the different data to be plotted

This function does the same plot for each axes.

```
def plot_ax_star(star,result,axe):
  linelists = [('_gesEW_mar',1,'this','o'),('_melendez_kur',2,'MEL12','^'),\
               ('_mine_kur' ,3,'SOU08','s'),('_ramirez_mar' ,4,'RAM13','D')]
  axe.plot([0,10],[0,0], linestyle='--', color='k', linewidth=2.0)
  for linelist,i,labelstr,mark in linelists:
    idstr = star+linelist+'.moog'
    res = [ (file_r,star_r,nfei_r, dfei_r, sfei_r) \
            for (file_r,star_r,nfei_r, dfei_r, sfei_r) \
            in result if file_r == idstr]
    (file_r,star_r,nfei_r, dfei_r, sfei_r) = res[0]
    axe.errorbar([i], [dfei_r], yerr=sfei_r,fmt=mark, label=labelstr, \
             linewidth=2.0, markersize=8)
  axe.set_xlim(0,7)
  axe.set_ylim(-0.5,0.5)
  axe.xaxis.set_visible(False)
  axe.set_title(star)
  axe.legend()
```

Extra stuff in this example:

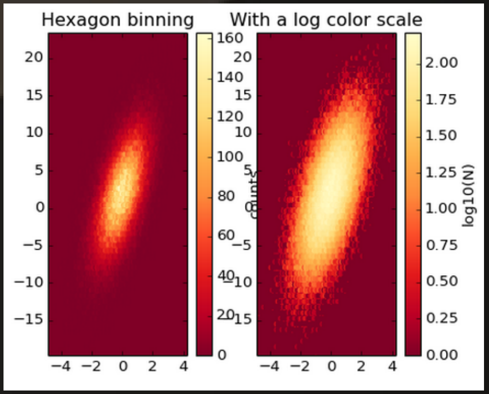
→ Horizontal dashed line
→ Using different symbols and automatic colors
→ Automatic legends
→ Using error bars

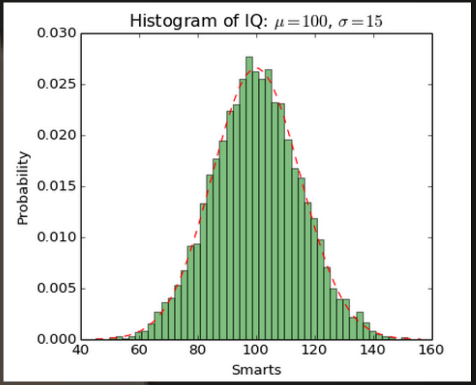
Matplotlib

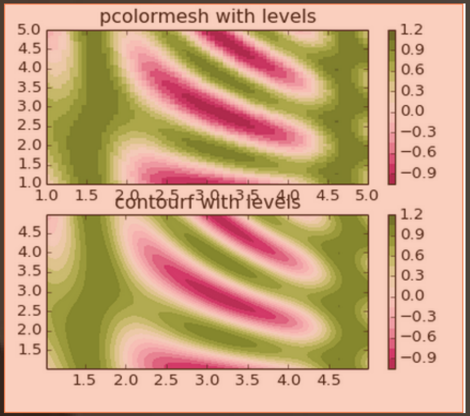
Matplotlib is a huge module which contains many features and many possibilities.

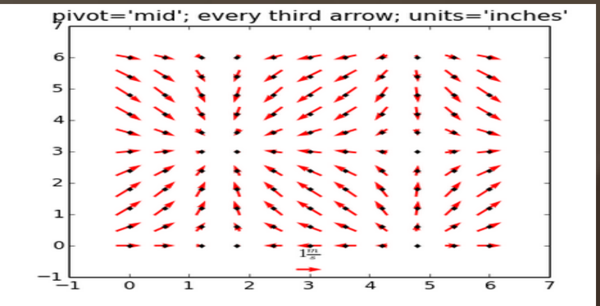
The use of Objects is benefit to control your plot easily. Experience comes with practice

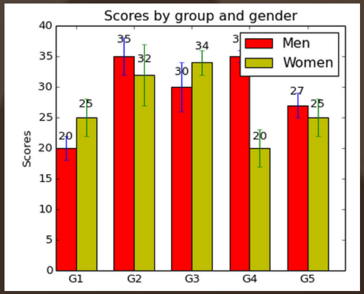
The is a gallery with several examples and source code in Matplotlib: http://matplotlib.org/gallery.html (You can use it for your inspiration...)

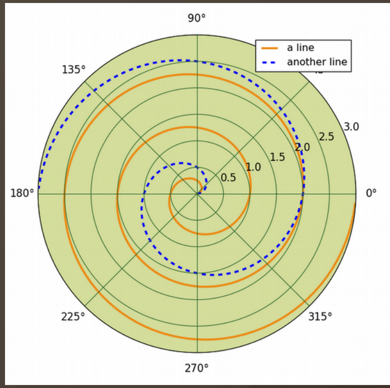


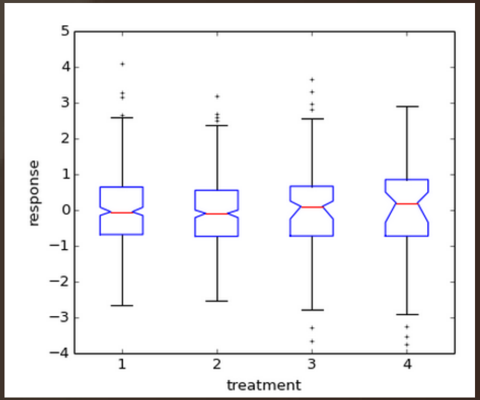


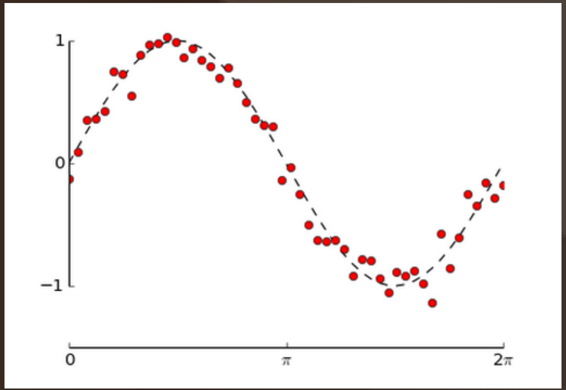


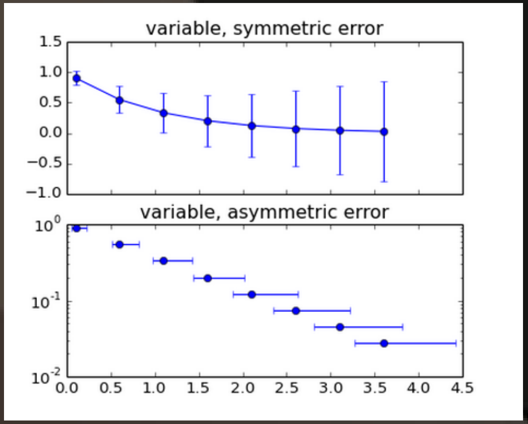




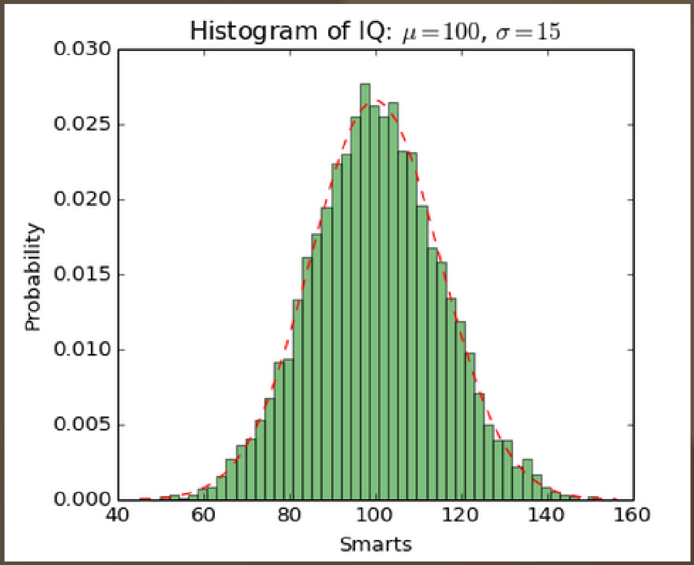








Matplotlib – Gallery Examples




```python
"""
Demo of the histogram (hist) function with a few features.

In addition to the basic histogram, this demo shows a few optional features:

    * Setting the number of data bins
    * The ``normed`` flag, which normalizes bin heights so that the integral of
      the histogram is 1. The resulting histogram is a probability density.
    * Setting the face color of the bars
    * Setting the opacity (alpha value).

"""
import numpy as np
import matplotlib.mlab as mlab
import matplotlib.pyplot as plt


# example data
mu = 100 # mean of distribution
sigma = 15 # standard deviation of distribution
x = mu + sigma * np.random.randn(10000)

num_bins = 50
# the histogram of the data
n, bins, patches = plt.hist(x, num_bins, normed=1, facecolor='green', alpha=0.5)
# add a 'best fit' line
y = mlab.normpdf(bins, mu, sigma)
plt.plot(bins, y, 'r--')
plt.xlabel('Smarts')
plt.ylabel('Probability')
plt.title(r'Histogram of IQ: $\mu=100$, $\sigma=15$')

# Tweak spacing to prevent clipping of ylabel
plt.subplots_adjust(left=0.15)
plt.show()
```

Random Gaussian Data Generation

Defining number of bins

Plotting histogram

Getting a normal probability density function. help(mlab)
(Mlab is a module for Numerical python functions written for compatability with MATLAB commands with the same names.)

Plotting pdf

Setting titles

Adjusting subplot in Figure

Matplotlib – Gallery Examples

```
import matplotlib.pyplot as plt
from matplotlib.colors import LogNorm
import numpy as np
from matplotlib.mlab import bivariate_normal


N = 100
X, Y = np.mgrid[-3:3:complex(0, N), -2:2:complex(0, N)]

# A low hump with a spike coming out of the top right.
# Needs to have z/colour axis on a log scale so we see both hump and spike.
# linear scale only shows the spike.
Z1 = bivariate_normal(X, Y, 0.1, 0.2, 1.0, 1.0) + 0.1 * bivariate_normal(X, Y, 1.0, 1.0, 0.0, 0.0)

plt.subplot(2,1,1)
plt.pcolor(X, Y, Z1, norm=LogNorm(vmin=Z1.min(), vmax=Z1.max()), cmap='PuBu_r')
plt.colorbar()

plt.subplot(2,1,2)
plt.pcolor(X, Y, Z1, cmap='PuBu_r')
plt.colorbar()


plt.show()
```

Normalize a given value to the 0-1 range on a log scale

Create the parameter space grid for the surface plot

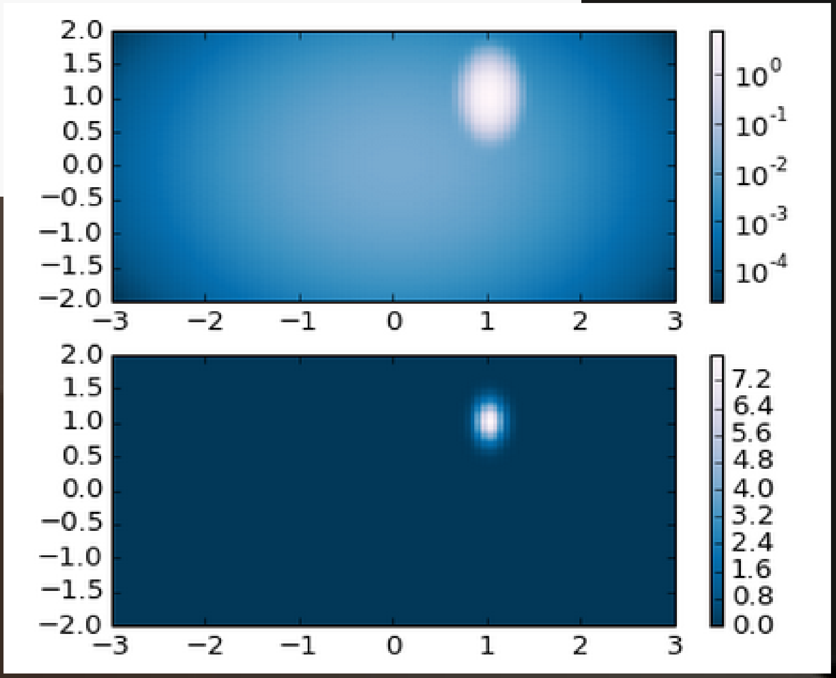
Creating data from bivariate normal distributions

This function does the same plot for each axes.

Creating the subplots – a grid of 2 rows, 1 column

Ploting a colar map

Creating a colar bar



Matplotlib – Gallery Examples

Necessary Imports

Changing the thick direction at the x and y axes.

Generating the random data

Creating the figure

Ploting the contours

Defining locations manually

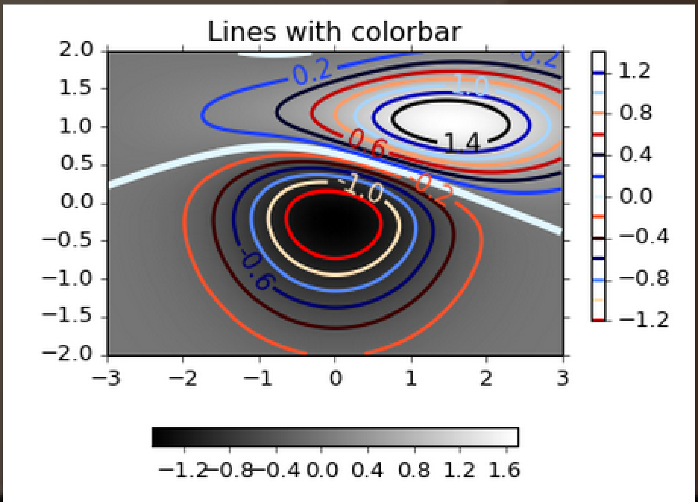
Plotting labels on contours

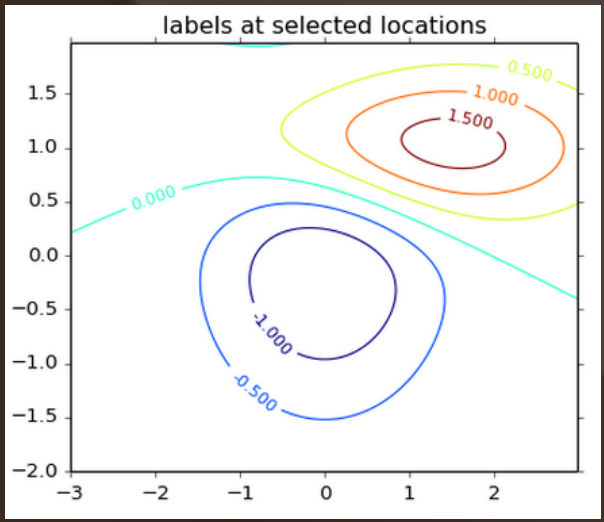
Adding title

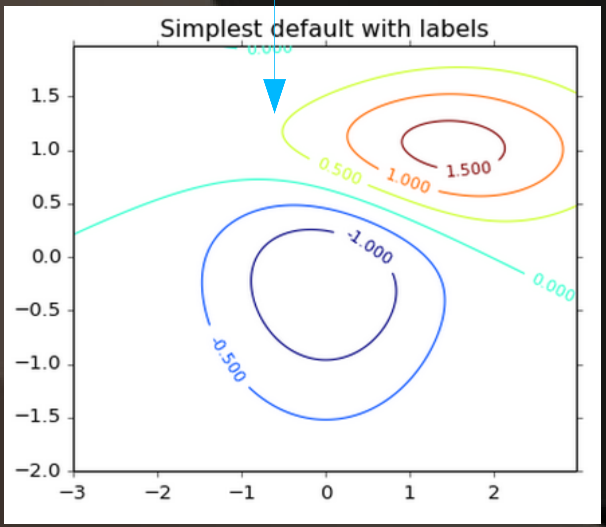
```
import matplotlib
import numpy as np
import matplotlib.cm as cm
import matplotlib.mlab as mlab
import matplotlib.pyplot as plt

matplotlib.rcParams['xtick.direction'] = 'out'
matplotlib.rcParams['ytick.direction'] = 'out'

delta = 0.025
x = np.arange(-3.0, 3.0, delta)
y = np.arange(-2.0, 2.0, delta)
X, Y = np.meshgrid(x, y)
Z1 = mlab.bivariate_normal(X, Y, 1.0, 1.0, 0.0, 0.0)
Z2 = mlab.bivariate_normal(X, Y, 1.5, 0.5, 1, 1)
# difference of Gaussians
Z = 10.0 * (Z2 - Z1)
# contour labels can be placed manually by providing list of positions
# (in data coordinate). See ginput_manual_clabel.py for interactive
# placement.
plt.figure()
CS = plt.contour(X, Y, Z)
manual_locations = [(-1, -1.4), (-0.62, -0.7), (-2, 0.5), (1.7, 1.2), (2.0, 1.4), (2.4, 1.7)]
plt.clabel(CS, inline=1, fontsize=10, manual=manual_locations)
plt.title('labels at selected locations')
```

Ignoring manual locations you get this plot

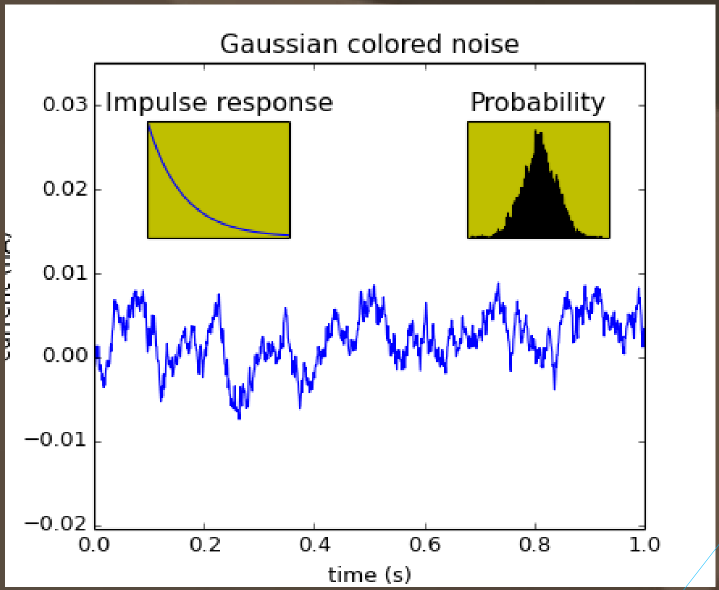
You have more advanced examples:







Matplotlib – Gallery Examples




```
#!/usr/bin/env python

from pylab import *

# create some data to use for the plot
dt = 0.001
t = arange(0.0, 10.0, dt)
r = exp(-t[:1000]/0.05)               # impulse response
x = randn(len(t))
s = convolve(x,r)[:len(x)]*dt  # colored noise

# the main axes is subplot(111) by default
plot(t, s)
axis([0, 1, 1.1*amin(s), 2*amax(s) ])
xlabel('time (s)')
ylabel('current (nA)')
title('Gaussian colored noise')

# this is an inset axes over the main axes
a = axes([.65, .6, .2, .2], axisbg='y')
n, bins, patches = hist(s, 400, normed=1)
title('Probability')
setp(a, xticks=[], yticks=[])

# this is another inset axes over the main axes
a = axes([0.2, 0.6, .2, .2], axisbg='y')
plot(t[:len(r)], r)
title('Impulse response')
setp(a, xlim=(0,.2), xticks=[], yticks=[])


show()
```

Example using pylab directly:

Generating the random data

Ploting main axes.
(Figure is automatically created)

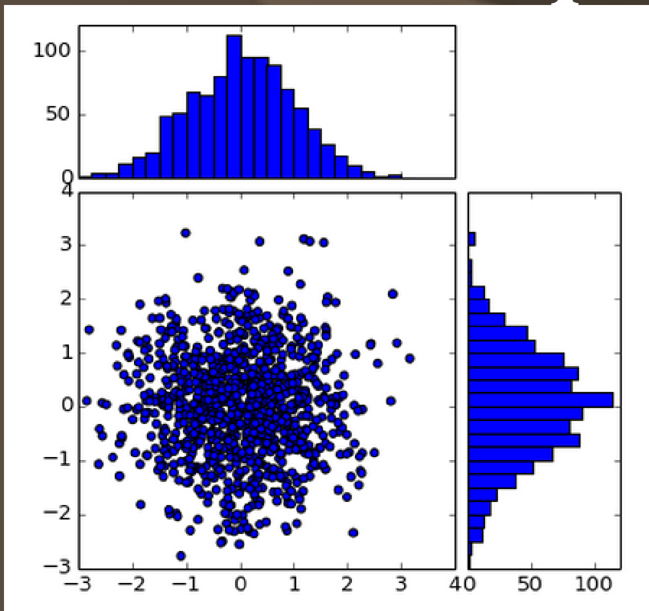
Changing limits for x & y axes(thicks)

Setting titles

Creating 1st axes inside – top right
(yellow background)
Plot Histogram
Empty thicks from axes with setp()

Creating 2nd axes inside – top left
Simple plot
Empty thicks from axes and setting x limit

Matplotlib – Gallery Examples



Import modules
Need mpl_toolkits.axes
Random data creation
Figure and axes created with subplot
Create a scater plot
Set equal scale for xy
Creation of the 2 attached axes for xy
Making x axes invisible in attached axes
Defining limits for the histograms
Defining bins
Plotting histograms in each attached axes
Adjusting thicks manually
Making invisible the default ones
And add thicks manually

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import make_axes_locatable

# the random data
x = np.random.randn(1000)
y = np.random.randn(1000)


fig, axScatter = plt.subplots(figsize=(5.5,5.5))

# the scatter plot:
axScatter.scatter(x, y)
axScatter.set_aspect(1.)

# create new axes on the right and on the top of the current axes
# The first argument of the new_vertical(new_horizontal) method is
# the height (width) of the axes to be created in inches.
divider = make_axes_locatable(axScatter)
axHistx = divider.append_axes("top", 1.2, pad=0.1, sharex=axScatter)
axHisty = divider.append_axes("right", 1.2, pad=0.1, sharey=axScatter)

# make some labels invisible
plt.setp(axHistx.get_xticklabels() + axHisty.get_yticklabels(),
         visible=False)

# now determine nice limits by hand:
binwidth = 0.25
xymax = np.max( [np.max(np.fabs(x)), np.max(np.fabs(y))] )
lim = ( int(xymax/binwidth) + 1) * binwidth

bins = np.arange(-lim, lim + binwidth, binwidth)
axHistx.hist(x, bins=bins)
axHisty.hist(y, bins=bins, orientation='horizontal')

# the xaxis of axHistx and yaxis of axHisty are shared with axScatter,
# thus there is no need to manually adjust the xlim and ylim of these
# axis.

#axHistx.axis["bottom"].major_ticklabels.set_visible(False)
for tl in axHistx.get_xticklabels():
    tl.set_visible(False)
axHistx.set_yticks([0, 50, 100])

#axHisty.axis["left"].major_ticklabels.set_visible(False)
for tl in axHisty.get_yticklabels():
    tl.set_visible(False)
axHisty.set_xticks([0, 50, 100])

plt.draw()
plt.show()
```

Python Advance Course via Astronomy street

Lesson 3: Python with Matplotlib, Scipy, Astropy, PyAstronomy

- Plotting with Matplotlib
- Using Scipy
- Astropy, PyAstronomy, ...

SciPy

SciPy.org Sponsored By ENTHOUGHT

http://www.scipy.org/

>>> import scipy
>>> help(scipy)

```
Help on package scipy:

NAME
    scipy

FILE
    /usr/lib/python2.7/dist-packages/scipy/__init__.py

MODULE DOCS
    http://docs.python.org/library/scipy

DESCRIPTION
    SciPy: A scientific computing package for Python
    ================================================

    Documentation is available in the docstrings and
    online at http://docs.scipy.org.

    Contents
    --------
    SciPy imports all the functions from the NumPy namespace, and in
    addition provides:
```

SciPy

http://www.scipy.org/

- Scipy is a big module with several toolboxes for scientific computing;
- It is divided in several specific submodules:
    - interpolation;
    - integration;
    - optimization;
    - image processing;
    - statistics;
- Scipy is comparable with GSL (Gnu Scientific Library) for C/C++ or Matlab's toolboxes
- Strong efficient dependence on Numpy arrays

Advice: Check the content of Scipy module before starting to reinvent the weel

SciPy.org Sponsored By ENTHOUGHT

SciPy

http://www.scipy.org/

File Input-Output : scipy.io

There are couple of functions to read specific files:

- Matlab;
- IDL;
- Fortran unformatted;
- WAV sound;
- other stuff: Matrix Market files, Arff files, Netcdf

Example IDL sav file: scipy.io.readsav(...)

```
>>> import scipy.io as sio
>>> idl_stuff = sio.readsav('EWs.sav')
>>> type(idl_stuff)
<class 'scipy.io.idl.AttrDict'>
>>> idl_stuff.keys()
['star', 'teff', 'erteff', 'ewsun', 'filevec', 'ele', 'num', 'ew_mat', 'loggf', 'ep', 'lambda']
>>> teff_idl_var = idl_stuff['teff']
>>> type(teff_idl_var)
<type 'numpy.ndarray'>
>>> teff_idl_var.dtype
dtype('>f8')
>>> teff_idl_var[2:4]
array([ 5536.,  4738.])
>>> star_idl_var = idl_stuff['star']
>>> star_idl_var.dtype
dtype('O')
>>> type(star_idl_var)
<type 'numpy.ndarray'>
```

Importing the module

Reading an IDL ".sav" file

The result is a Dictionary of Numpy arrays with different types

SciPy.org Sponsored By ENTHOUGHT

SciPy

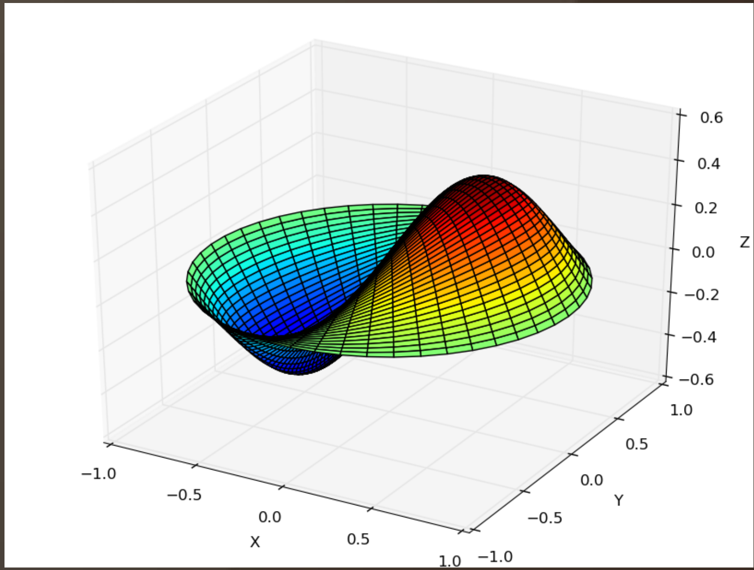
http://www.scipy.org/

Special Functions : scipy.special

This submodule includes a series of built in special functions:

- Airy functions;
- Elliptic functions and integrals;
- Bessel functions, zeros, integrals, derivatives, spherical;
- Raw statistical Functions;
- Gamma and related functions;
- Legendre Functions
- Orthogonal Polynomials
- Spheroidal Wave functions
- ...

Example: Using the bessel function



http://docs.scipy.org/doc/scipy/reference/tutorial/special.html

```
from scipy import *
from scipy.special import jn, jn_zeros
def drumhead_height(n, k, distance, angle, t):
  nth_zero = jn_zeros(n, k)
  return cos(t)*cos(n*angle)*jn(n, distance*nth_zero)

theta = r_[0:2*pi:50j]
radius = r_[0:1:50j]
x = array([r*cos(theta) for r in radius])
y = array([r*sin(theta) for r in radius])
z = array([drumhead_height(1, 1, r, theta, 0.5) for r in radius])
import pylab
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
fig = pylab.figure()
ax = Axes3D(fig)
ax.plot_surface(x, y, z, rstride=1, cstride=1, cmap=cm.jet)
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
pylab.show()
```

SciPy.org Sponsored By ENTHOUGHT

SciPy

http://www.scipy.org/

Linear Algebra Operations : scipy.linalg

- This submodule includes a series operations of linear algebra.
- Many of these overlap the operations that we can do directly with Numpy;

```
>>> import numpy as np
>>> from scipy import linalg
>>> arr = np.array([[1,2],[3,4]])
>>> linalg.det(arr)
-2.0
>>> linalg.det(np.array([1,2]))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/python2.7/dist-packages/scipy/linalg/basic.py", line 439, in det
    raise ValueError('expected square matrix')
ValueError: expected square matrix
>>> iarr = linalg.inv(arr)
>>> iarr
array([[-2. ,  1. ],
       [ 1.5, -0.5]])
```

Importing the module

Computing the determinant of the matrix

Exception handling functionalities

- There are however some extra interesting stuff: - help(scipy.linalg) to see the list of funcionalities!
  Example: singular-value decomposition (SVD):

```
>>> arr = np.arange(9).reshape((3, 3)) + np.diag([1, 0, 1])
>>> uarr, spec, vharr = linalg.svd(arr)
>>> sarr = np.diag(spec)
>>> svd_mat = uarr.dot(sarr).dot(vharr)
>>> np.allclose(svd_mat, arr)
True
```

allclose() → checks if arrays are equal considering numerical errors in computation

```
>>> arr
array([[1, 1, 2],
       [3, 4, 5],
       [6, 7, 9]])
>>> svd_mat
array([[ 1.,  1.,  2.],
       [ 3.,  4.,  5.],
       [ 6.,  7.,  9.]])
```

SciPy.org Sponsored By ENTHOUGHT

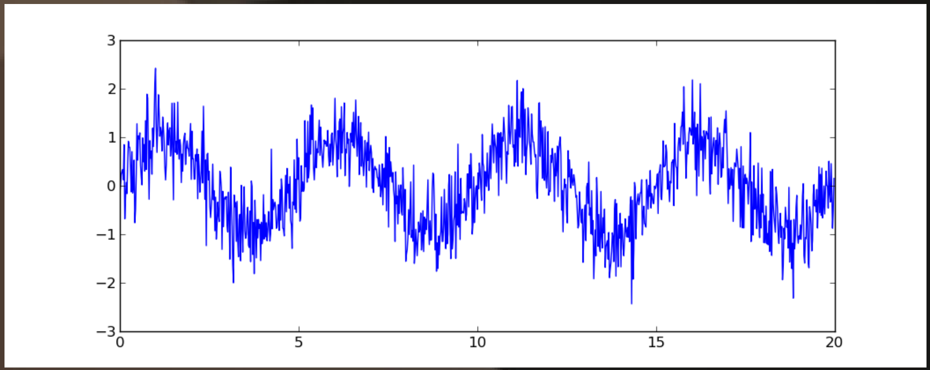
SciPy

http://www.scipy.org/

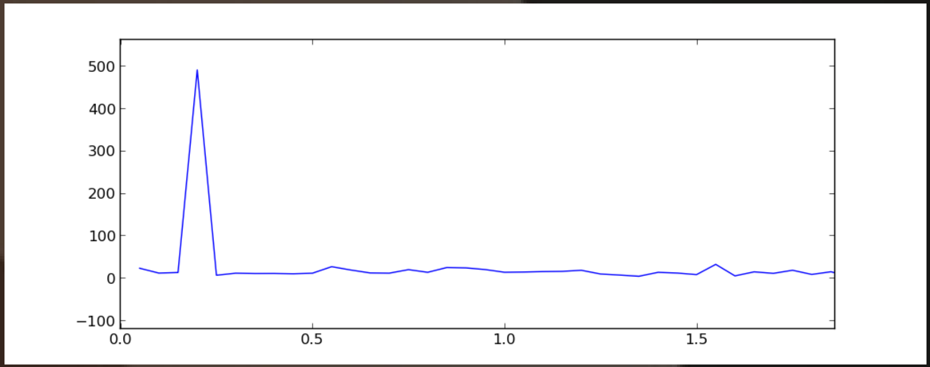
Fast Fourier transforms: scipy.fftpack

- This submodule allows to compute fast Fourier transforms

```
>>> import numpy as np
>>> time_step = 0.02
>>> period = 5.
>>> time_vec = np.arange(0, 20, time_step)
>>> sig = np.sin(2 * np.pi / period * time_vec) + \
...       0.5 * np.random.randn(time_vec.size)
>>> import matplotlib.pyplot as plt
>>> plt.plot(time_vec,sig)
[<matplotlib.lines.Line2D object at 0x379f3d0>]
>>> plt.show()
```




```
>>> from scipy import fftpack
>>> sample_freq = fftpack.fftfreq(sig.size, d=time_step)
>>> sig_fft = fftpack.fft(sig)
>>> pidxs = np.where(sample_freq > 0)
>>> freqs = sample_freq[pidxs]
>>> power = np.abs(sig_fft)[pidxs]
>>> plt.plot(freqs,power)
[<matplotlib.lines.Line2D object at 0x378e210>]
>>> plt.show()
```



Checking the derived frequency:

```
>>> freq = freqs[power.argmax()]
>>> freq
0.20000000000000001
>>> np.allclose(freq, 1./period)
True
```

Numpy also has an implementation of FTT (numpy.ftt). However, in general the scipy version should be prefered, because it uses more efficient underlying implementation.

SciPy.org Sponsored By ENTHOUGHT

SciPy

http://www.scipy.org/

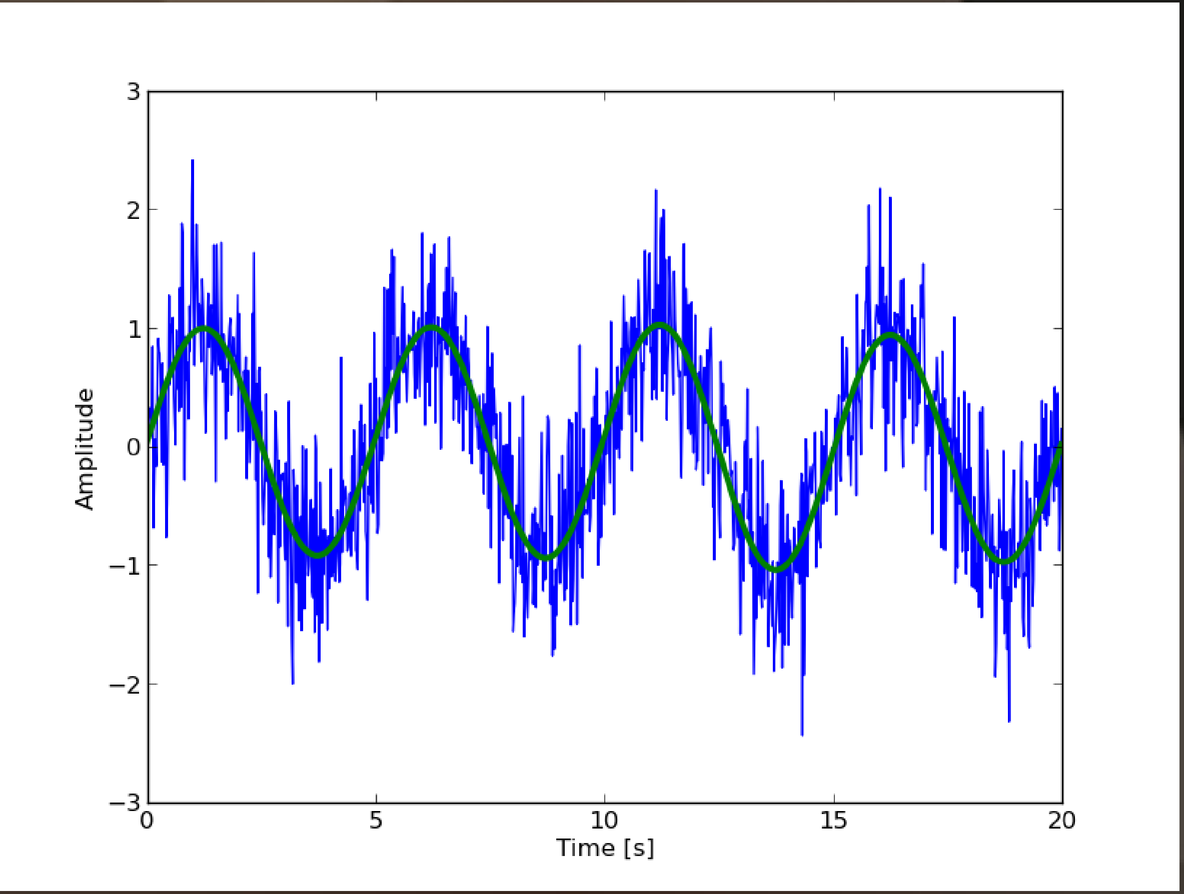
Fast Fourier transforms: scipy.fftpack

Example: Filtering the data

```
>>> sig_fft[np.abs(sample_freq) > freq] = 0
>>> main_sig = fftpack.ifft(sig_fft)
```

Filtering the data

Inverting the fourier transform



SciPy.org Sponsored By ENTHOUGHT

SciPy

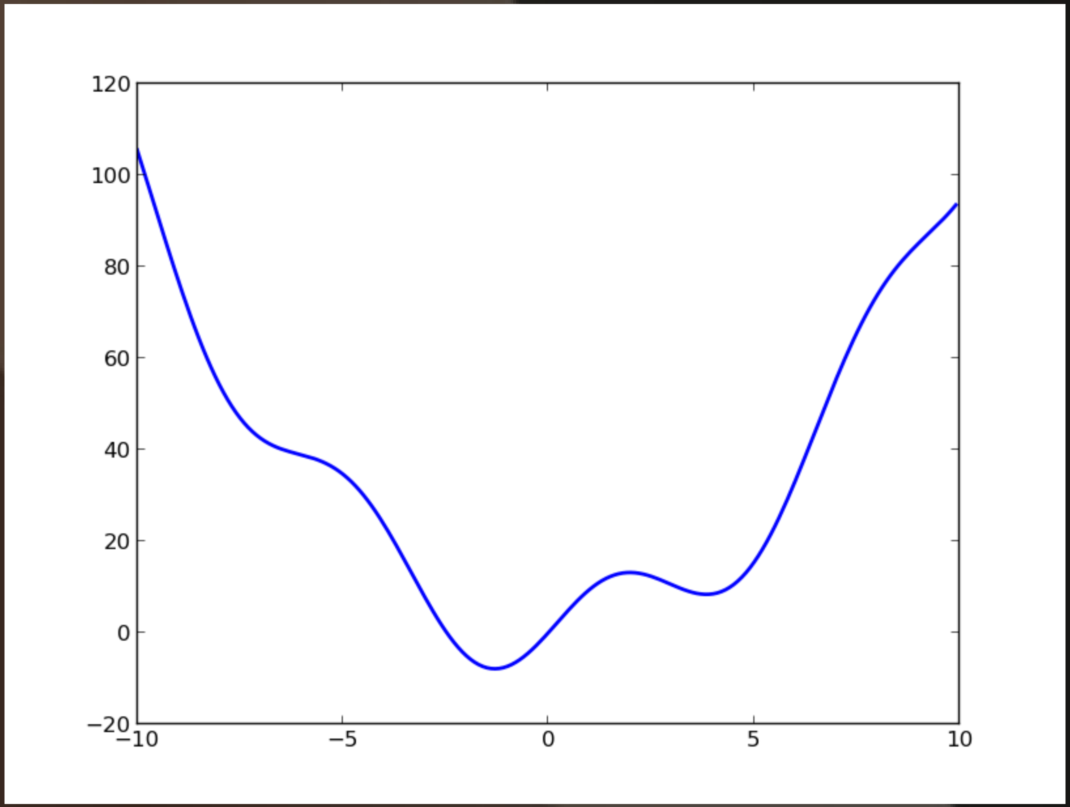
http://www.scipy.org/

Optimization and fitting : scipy.optimize

Optimization is the problem of finding a numerical solution to a minimization or equality.

<u>Finding the minimum of a scalar function:</u>

```
>>> import numpy as np
>>> import scipy.optimize as sop
>>> def f(x):
...     return x**2 + 10*np.sin(x)
...
>>> x = np.arange(-10,10,0.1)
>>> import matplotlib.pyplot as plt
>>> plt.plot(x,f(x))
[<matplotlib.lines.Line2D object at 0x3da4f90>]
>>> plt.show()
```



Using the BFGS algorithm:

```
>>> sop.fmin_bfgs(f,0)
Optimization terminated successfully.
         Current function value: -7.945823
         Iterations: 5
         Function evaluations: 24
         Gradient evaluations: 8
array([-1.30644003])
```

SciPy.org Sponsored By ENTHOUGHT

SciPy

http://www.scipy.org/

Optimization and fitting : scipy.optimize

Optimization is the problem of finding a numerical solution to a minimization or equality.

Finding the minimum of a scalar function:

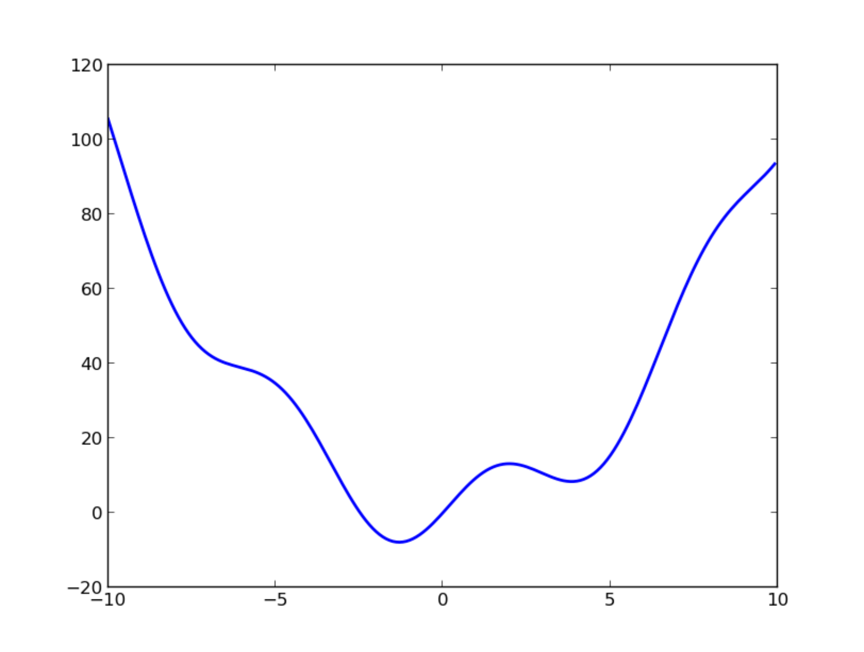
Careful with local minimum
(changing initial guess):

```
>>> sop.fmin_bfgs(f, 3)
Optimization terminated successfully.
         Current function value: 8.315586
         Iterations: 5
         Function evaluations: 24
         Gradient evaluations: 8
array([ 3.83746663])
```

When you don't know the function, you can simply use brute force:

```
>>> grid = (-10,10,0.1)
>>> xmin_global = sop.optimize.brute(f,(grid,))
>>> xmin_global
array([-1.30641113])
```

Tip: You can always use *help(module.function)* to know how to use it



Brute force can be quite heavy, in the case where you need large grids.
Alternatively you can use *scipy.optimize.anneal()* which uses the simulating annealing minimization algorithm.

Other modules for Minimization:
OpenOpt, IPOPT, PyGMO and PyEvolve

SciPy.org Sponsored By ENTHOUGHT

SciPy

http://www.scipy.org/

Optimization and fitting : scipy.optimize

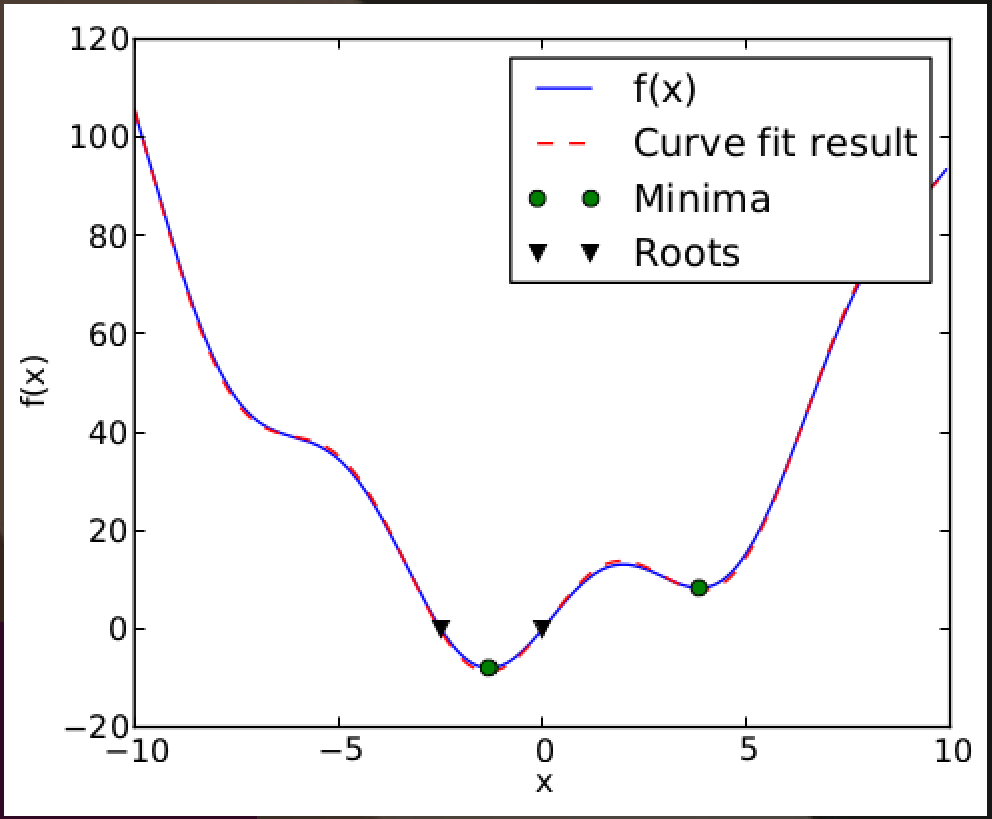
Optimization is the problem of finding a numerical solution to a minimization or equality.

Finding roots of a scalar function:

```
>>> sop.fsolve(f,1)
array([ 0.])
>>> sop.newton(f,1)
-1.0117953699300726e-20
```

Using fsolve()

Using Newton-Raphson or secant method



Curve Fitting:

```
>>> xdata = np.linspace(-10, 10, num=20)
>>> ydata = f(xdata) + np.random.randn(xdata.size)
>>> def f2(x, a, b):
...     return a*x**2 + b*np.sin(x)
...
>>> guess = [2, 2]
>>> params, params_covariance = sop.curve_fit(f2, xdata, ydata, guess)
>>> params
array([  1.0075586 ,  10.05212948])
```

SciPy.org Sponsored By ENTHOUGHT

SciPy

http://www.scipy.org/

Statistics and Random Numbers : scipy.stats

This module contains a large number of probability distributions as well as a growing library of statistical functions.

This module is divided into:

- Continuous Distributions;
- Discrete Distributions;
- Statistical functions;
- Contingency table functions;
- General linear model;
- Plot-tests;
- Univariate and multivariate kernel density estimation

The detailled information of the module containts can be found using: help(scipy.stats)

SciPy.org Sponsored By ENTHOUGHT

SciPy

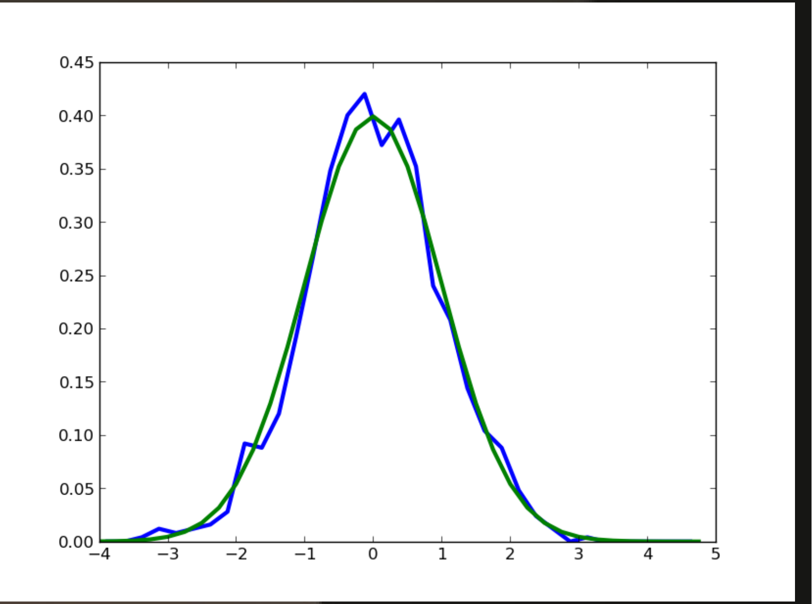
http://www.scipy.org/

Statistics and Random Numbers : scipy.stats

Examples: Using the Normal function:

```
>>> import numpy as np
>>> import scipy.stats as sst
>>> import matplotlib.pyplot as plt
>>> a = np.random.normal(size=1000)
>>> bins = np.arange(-4,5,0.25)
>>> bin_plot = 0.5*(bins[1:] + bins[:-1])
>>> histogram = np.histogram(a, bins=bins, normed=True)[0]
>>> b = sst.norm.pdf(bins)
>>> plt.plot(bin_plot,histogram)
[<matplotlib.lines.Line2D object at 0x45c1910>]
>>> plt.plot(bins,b)
```

Creating random data



Creating an histogram to plot

Geting the values of a Normal Function

Statistical Tests:

```
>>> loc, std = sst.norm.fit(a)
>>> loc,std
(0.032302008648340083, 1.0041030968829048)
```

Using Maximum Likelihood Estimate to extract the mean and the std...

```
>>> a = np.random.normal(0,1,size=100)
>>> b = np.random.normal(1,1,size=10)
>>> sst.ttest_ind(a,b)
(array(-3.3025291483817876), 0.0012977291859732 65)
>>> sst.ks_2samp(a,b)
(0.47000000000000003, 0.023495863518120642)
>>> b = np.random.normal(0,1,size=10)
>>> sst.ks_2samp(a,b)
(0.20000000000000007, 0.81585145499712663)
```

Using a T Test on two samples

Using a Kolmogorov-Smirnov Test

SciPy.org Sponsored By ENTHOUGHT

SciPy

http://www.scipy.org/

Interpolation: scipy.interpolate

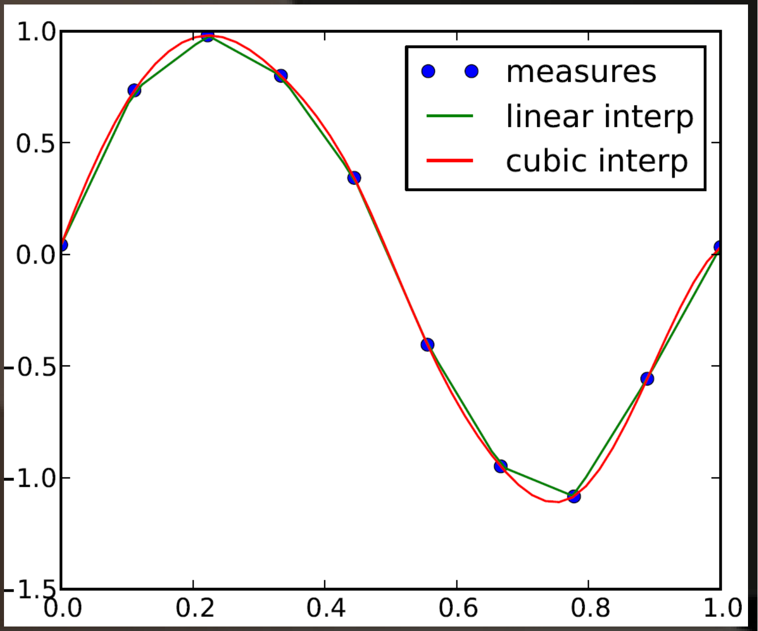
The scipy.interpolate is useful for fitting a function from experimental data and thus evaluating points where no measure exists. (The module is based on the FITPACK Fortran subroutines from the netlib project)

```
>>> import numpy as np
>>> import scipy.interpolate as sip
>>> measured_time = np.linspace(0,1,10)
>>> noise = (np.random.random(10)*2 - 1) * 0.1
>>> measures = np.sin(2 * np.pi * measured_time) + noise
>>> linear_interp = sip.interp1d(measured_time, measures)
>>> linear_interp(0.33)
array(0.8524539281420858)
>>> computed_time = np.linspace(0,1,50)
>>> linear_results = linear_interp(computed_time)
>>> cubic_interp = sip.interp1d(measured_time, measures, kind='cubic')
>>> cubic_results = cubic_interp(computed_time)
```

Creating the data sampled with some noise

Linear Interpolation

Cubic Interpolation



When you use linear_interp the result is function (object) that can be used directly like any other defined function.

This function then returns the interpoled values for the input data values.

Note. There are several interpolation functions that can be used in this module. Check: help(scipy.interpolate)

SciPy.org Sponsored By ENTHOUGHT

SciPy

http://www.scipy.org/

Numerical Integration : scipy.integrate

This module contains a several functions to integrate functions / data:

This module is divided into:

- Integrating functions, given function object (ex. quad, romberg);
- Integrating functions, given fixed samples (ex. trapezoidal rule, simpson)
- Integrators of Ordinary Differential equiations (ODE) systems;

The detailled information of the module containts can be found using: help(scipy.integrate)

Most generic integration routine is: scipy.integrate.quad() which uses a technique from the Fortran library QUADPACK:

```
>>> import numpy as np
>>> import scipy.integrate as sit
>>> res, err = sit.quad(np.sin,0,np.pi/2)
>>> res,err
(0.9999999999999999, 1.1102230246251564e-14)
```

Integrating the sin function between 0 and Pi/2.

SciPy.org Sponsored By ENTHOUGHT

SciPy

http://www.scipy.org/

Signal Processing : scipy.signal

This module contains a large collection of functionalities:

- Convolution;
- B-splines;
- Filtering;
- Filter Design (include some MatLab like functions);
- Continuous (and Discrete)-Time Linear Systems
- Waveforms (ex. Gaussian modulated sinusoid)
- Window Functions (ex. Boxcar window)
- Wavelets
- Peak Finding
- Spectral Analysis (periodogram, lombscarle)

For more information: help(scipy.signal)

SciPy

http://www.scipy.org/

SciPy.org Sponsored By ENTHOUGHT

Reference

- Clustering package (**scipy.cluster**)
- Constants (**scipy.constants**)
- Discrete Fourier transforms (**scipy.fftpack**)
- Integration and ODEs (**scipy.integrate**)
- Interpolation (**scipy.interpolate**)
- Input and output (**scipy.io**)
- Linear algebra (**scipy.linalg**)
- Miscellaneous routines (**scipy.misc**)
- Multi-dimensional image processing (**scipy.ndimage**)
- Orthogonal distance regression (**scipy.odr**)
- Optimization and root finding (**scipy.optimize**)
- Signal processing (**scipy.signal**)
- Sparse matrices (**scipy.sparse**)
- Sparse linear algebra (**scipy.sparse.linalg**)
- Compressed Sparse Graph Routines (**scipy.sparse.csgraph**)
- Spatial algorithms and data structures (**scipy.spatial**)
- Special functions (**scipy.special**)
- Statistical functions (**scipy.stats**)
- Statistical functions for masked arrays (**scipy.stats.mstats**)
- C/C++ integration (**scipy.weave**)

Python Advance Course via Astronomy street

Lesson 3: Python with Matplotlib, Scipy, Astropy, PyAstronomy

- Plotting with Matplotlib
- Using Scipy
- Astropy, PyAstronomy, ...

python

astropy

A Community Python Library for Astronomy

Welcome to the Astropy documentation! Astropy is a community-driven package intended to contain much of the core functionality and some common tools needed for performing astronomy and astrophysics with Python.

http://www.astropy.org/

http://docs.astropy.org/en/stable/

Core data structures and transformations

- Constants (**`astropy.constants`**)
- Units and Quantities (**`astropy.units`**)
- N-dimensional datasets (**`astropy.nddata`**)
- Data Tables (**`astropy.table`**)
- Time and Dates (**`astropy.time`**)
- Astronomical Coordinate Systems (**`astropy.coordinates`**)
- World Coordinate System (**`astropy.wcs`**)
- Models and Fitting (**`astropy.modeling`**)
- Analytic Functions (**`astropy.analytic_functions`**)

Connecting up: Files and I/O

- Unified file read/write interface
- FITS File handling (**`astropy.io.fits`**)
- ASCII Tables (**`astropy.io.ascii`**)
- VOTable XML handling (**`astropy.io.votable`**)
- Miscellaneous Input/Output (**`astropy.io.misc`**)
- Virtual Observatory Access (**`astropy.vo`**)

astropy

A Community Python Library for Astronomy

Welcome to the Astropy documentation! Astropy is a community-driven package intended to contain much of the core functionality and some common tools needed for performing astronomy and astrophysics with Python.

http://www.astropy.org/

http://docs.astropy.org/en/stable/

Astronomy computations and utilities

- Cosmological Calculations (**`astropy.cosmology`**)
- Convolution and filtering (**`astropy.convolution`**)
- Data Visualization (**`astropy.visualization`**)
- Astrostatistics Tools (**`astropy.stats`**)

Nuts and bolts of Astropy

- Configuration system (**`astropy.config`**)
- I/O Registry (**`astropy.io.registry`**)
- Logging system
- Python warnings system
- Astropy Core Package Utilities (**`astropy.utils`**)
- Astropy Testing helpers (**`astropy.tests.helper`**)

https://github.com/sczesla/PyAstronomy

http://www.hs.uni-hamburg.de/DE/Ins/Per/Czesla/PyA/PyA/index.html

PyAstronomy

What is it?

PyAstronomy is a collection of astronomy-related packages written in Python.

Currently, the following subpackages are available:

```
funcFit:    A convenient fitting package providing support
            for minimization and MCMC sampling.

modelSuite: A Set of astrophysical models (e.g., transit
            light-curve modeling), which can be used
            stand-alone or with funcFit.

AstroLib:   A set of useful routines including a number
            of ports from IDL's astrolib.

Constants:  The package provides a number of often-needed
            constants.

Timing:     Provides algorithms for timing analysis such as
            the Lomb-Scargle and the Generalized Lomb-Scargle
            periodogram

pyaGUI:     A collection of GUI tools for interactive work.
```

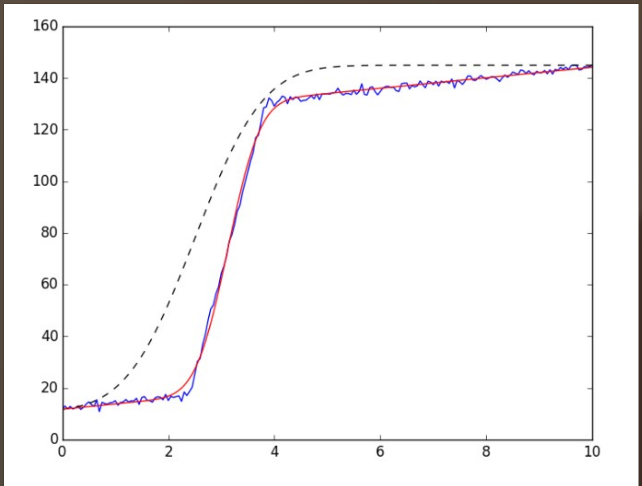
LMFIT

Non-Linear Least-Squares Minimization and Curve-Fitting for Python

[intro | parameters | minimize | model | builtin models | confidence intervals | bounds | constraints]

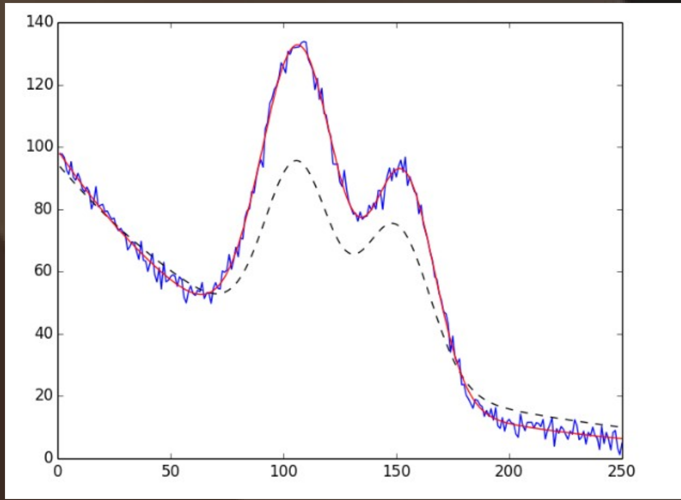
https://lmfit.github.io/lmfit-py/

$$\chi^2 = \sum_i^N \frac{[y_i^{\mathrm{meas}} - y_i^{\mathrm{model}}(\mathbf{v})]^2}{\epsilon_i^2}$$



Fit data to a Composite Model with pre-defined models

StepModel + LinearModel



Fitting Multiple Peaks

2 Gaussians + Exponential

Built-in Fitting Models in the **models** module

- Peak-like models
  - **GaussianModel**
  - **LorentzianModel**
  - **VoigtModel**
  - **PseudoVoigtModel**
  - **MoffatModel**
  - **Pearson7Model**
  - **StudentsTModel**
  - **BreitWignerModel**
  - **LognormalModel**
  - **DampedOcsillatorModel**
  - **ExponentialGaussianModel**
  - **SkewedGaussianModel**
  - **DonaichModel**
- Linear and Polynomial Models
  - **ConstantModel**
  - **LinearModel**
  - **QuadraticModel**
  - **ParabolicModel**
  - **PolynomialModel**
- Step-like models
  - **StepModel**
  - **RectangleModel**
- Exponential and Power law models
  - **ExponentialModel**
  - **PowerLawModel**
- User-defined Models
  - **ExpressionModel**

Good Practices in Python

- Explicit variable names (no need of a comment to explain what is in the variable)
- Style: spaces after commas, around =, etc. A certain number of rules for writing “beautiful” code (and, more importantly, using the same conventions as everybody else!) are given in the Style Guide for Python Code and the Docstring Conventions page (to manage help strings).
- Except some rare cases, variable names and comments in English.
- Good identation (forced in Python)

Python is also an object

Sérgio Sousa (CAUP)
ExoEarths Team (http://www.astro.up.pt/exoearths/)

```
Python 2.7.5+ (default, Sep 19 2013, 13:48:49)
[GCC 4.8.1] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```