12th Summer School on SCIENTIFIC VISUALIZATION

Introduction to Scientific Computing and Visualization in Python

Alice Invernizzi– a.invernizzi@cineca.it
SuperComputing Applications and Innovation Department

CINECA

INDEX

- Introduction
- Speeding Up Python: Numpy array data structures
- IPython for interactive computation
- Visualizing 2D Data with matplotlib
- Brief introduction to 3D Visualization with Mayavi

Introduction

- Python is a powerful, flexible, open-source language that is easy to learn, easy to use and has powerful libraries for data manipulation.

- Python has been used in scientific computing and highly quantitative domains such as finance, oil and gas, physics and signal processing…
- http://www.python.org/about/success/#scientific

- What are the key elements that ensure usability of this language in science?

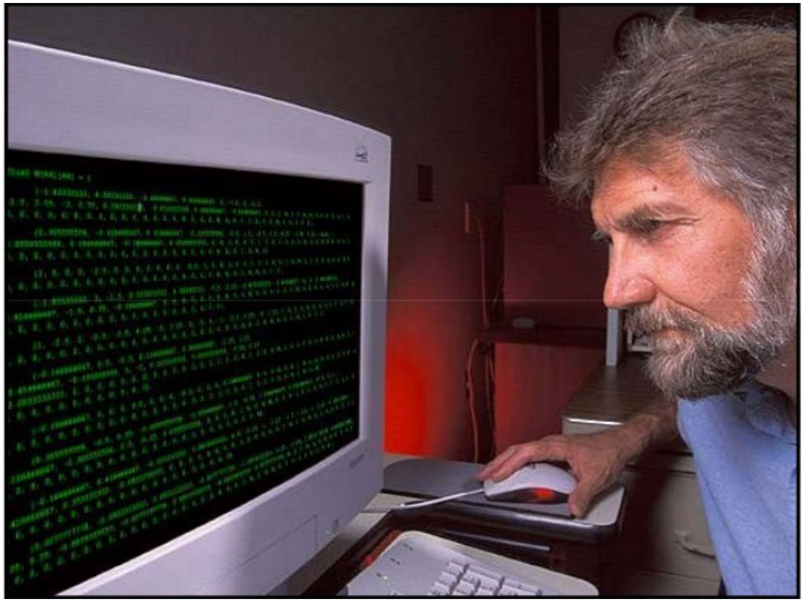
- Python provides easy-to-use tools for data structuring, manipulation, query, analysis and visualization

Introduction

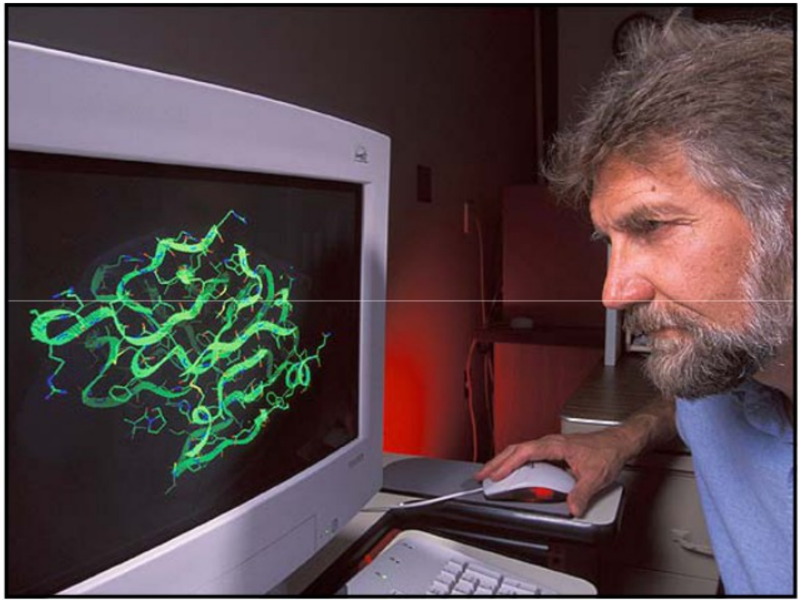
"The purpose of computation is insight, not numbers"

Richard Hamming, *Numerical Analysis for Scientists and Engineer*

From Scientific Data



To Scientific Visualization



To understand the meaning of the numbers we compute, we often need postprocessing, statistical analysis and graphical visualization of our data.

Introduction

The scientist's needs

- Get data (simulation, experiment control)
- Manipulate and process data.
- Visualize results... to understand what we are doing!
- Communicate results: produce figures for reports or publications, write presentations.

Python has all desirable tools for satisfying Scientific Computing users…

- **IPython**, an advanced **Python shell** for interactive computing
- **Numpy** : provides powerful **numerical arrays** objects, and routines to manipulate them
- **Scipy** : high-level data processing routines. Optimization, regression, interpolation
- **Matplotlib** : 2-D visualization, "publication-ready" plot
- **Mayavi** : 3-D visualization

GET DATA	`urllib2`
PARSE IT	`csv, beautifulsoup`
PROCESS	`numpy, scipy`
VISUALIZE	`matplotlib, chaco, mayavi2`
PUBLISH	LaTeX `cherrypy`

Numpy

an efficient multi-dimensional container for generic data

Why Numpy?

How slow is Python?
Let’s add on one to a million numbers.

```
C:\Users\invernizzi>python -m timeit -c
"[i+1 for i in range(1000000)]"
10 loops, best of 3: 59.3 msec per loop
```

Why Python is slow?
•Dynamic typing requires a lot of metadata around variable.
•Python uses heavy frame objects during iteration
Solution:
•Make an object that has a single type and continuous storage.
•Implement common functionality into that object to iterate in C

Why Numpy?

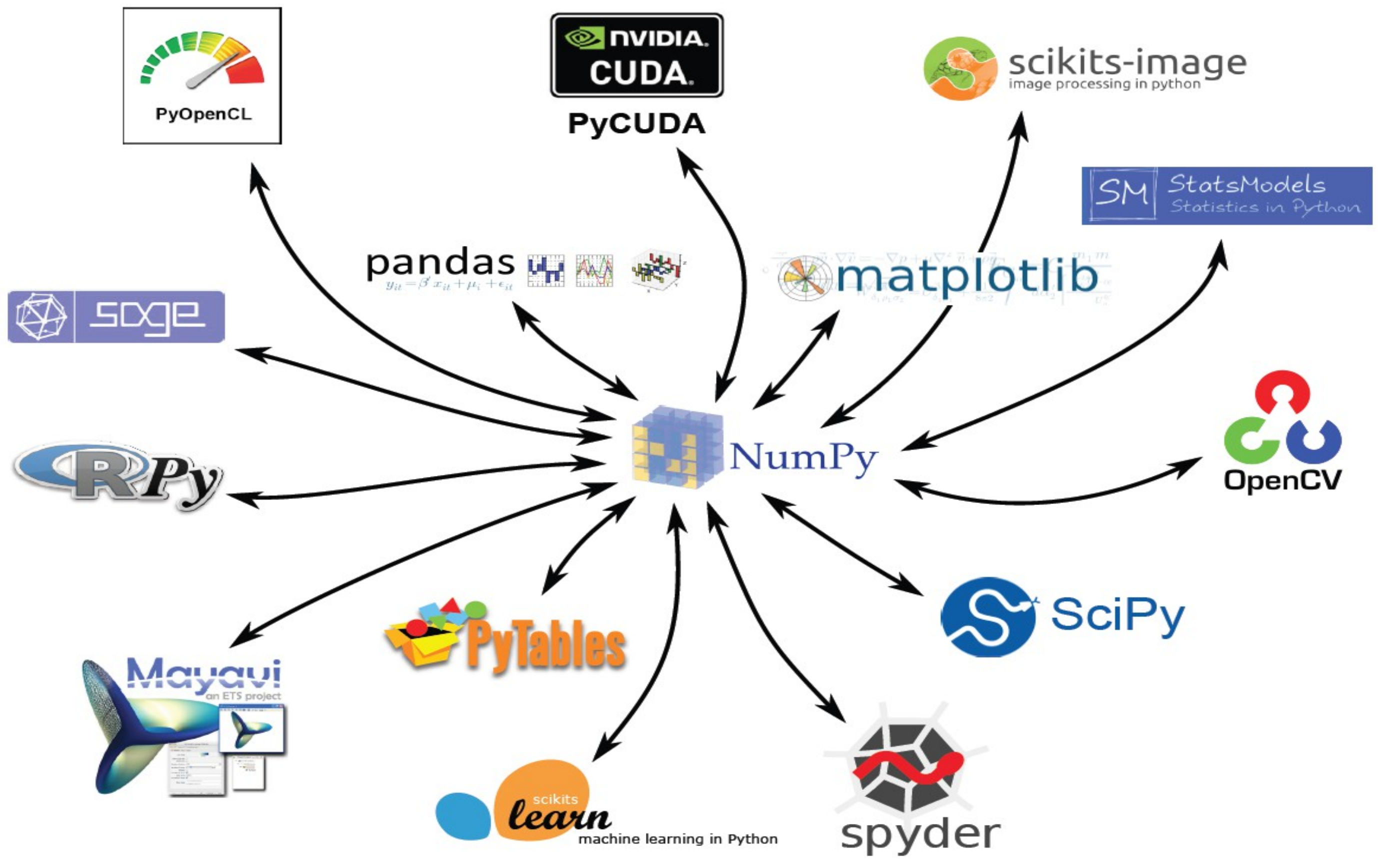
Speeding Up Python:
Let's add on one to a million numbers, using numpy library

```
C:\Users\invernizzi>python -m timeit -s
"import numpy" -c "numpy.arange(1000000)+1"
100 loops, best of 3: 2.91 msec per loop
```

Why Python is fast?

- Homogenous data type object: every item takes up the same size block of memory .
- Function that operates on ndarray in an element by element fashion
- Vectorize wrapper for a function
- build-in function are implemented in compiled C code.

Numpy



"Life is too short to write C++ code"

David Beazley - EuroScipy 2012 Bruxelles

Numpy

Features:

- A powerful N-dimensional array object
- Broadcasting function
- Tools for integrating C/C++ and Fortran code
- Useful linear algebra, Fourier transform and random number capabilities.
- Ufuncs, function that operates on ndarrays in an element-by-element fashion

History:

- Based originally on `Numeric` by Jim Hugunin
- Also based on `NumArray` by Perry Greenfield
- Written both by Trevis Oliphant to bring both features set together.

Numpy

Sub-Packages	Purpose	Comments
core	basic objects	all names exported to numpy
lib	Addintional utilities	all names exported to numpy
linalg	Basic linear algebra	LinearAlgebra derived from Numeric
fft	Discrete Fourier transforms	FFT derived from Numeric
random	Random number generators	RandomArray derived from Numeric
distutils	Enhanced build and distribution	improvements built on standard distutils
testing	unit-testing	utility functions useful for testing
f2py	Automatic wrapping of Fortran code	a useful utility needed by SciPy

Numeric Array

Array Creation

```
>>> import numpy as np
>>> a = np.array([0,1,2,3])
>>> a
array([0, 1, 2, 3])
>>a=array([0,1,2],dtype=float)
array([  0.,   1.,   2.])
>>> a=np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a=np.linspace(0,10,10)
>>> a
array([  0.        ,   1.11111111,   2.22222222,   3.33333333,
         4.44444444,   5.55555556,   6.66666667,   7.77777778,
         8.88888889,  10.        ])
>>> a=array([[1,2,3],[4,5,6]])
>>> a
array([[1, 2, 3],
       [4, 5, 6]])
```

Numeric Array

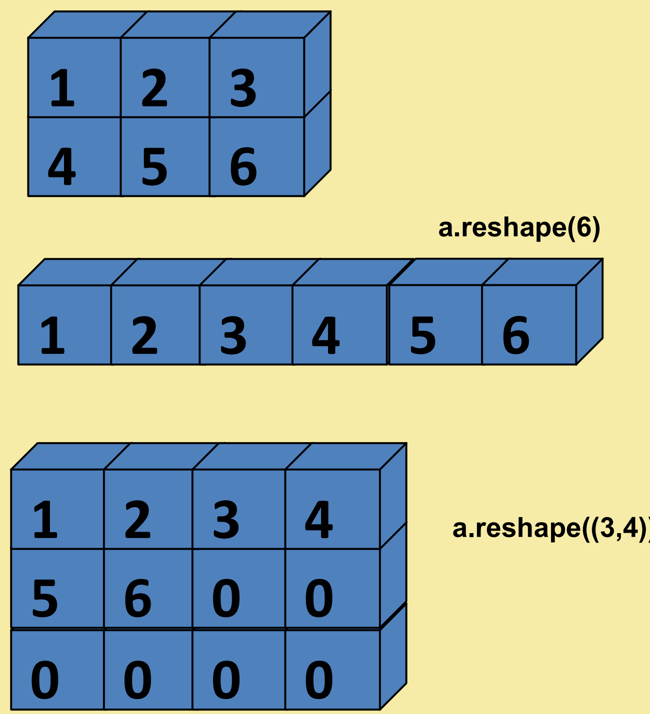
Array Creation

```
array(object, dtype=None, copy=1,order=None, subok=0,ndmin=0)

arange([start,]stop[,step=1],dtype=None)

ones(shape,dtype=None,order='C')

zeros(shape,dtype=float,order='C')

identity(n,dtype='l')

linspace(start, stop, num=50, endpoint=True, retstep=False)

empty( shape, dtype=None, order ='C' )

eye( N, M=None, k=0, dtype=float )
```

Numeric Array

Array Shape

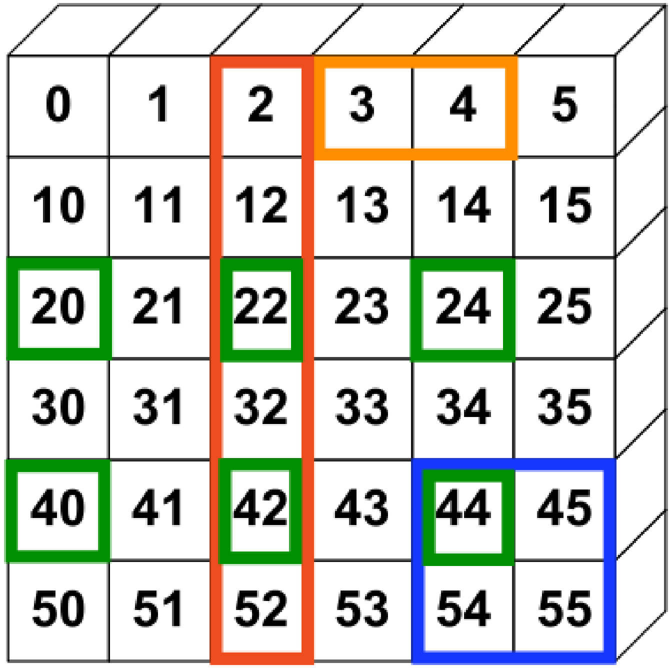
```
>>>a=array([[1,2,3],[4,5,6]])
>>> a.itemsize
4
>>> a.shape
(2, 3)
>>>a.reshape(6)
array([1,2,3,4,5,6])
>>> a.resize((3,4))
>>> a
array([[1, 2, 3, 4],
       [5, 6, 0, 0],
       [0, 0, 0, 0]])
>>> a.size
12
>>> a.mean()
1.75
>>> a.max()
6
>>> a.min()
0
```



Numeric Array

Array Slicing

```
>>> a[0,3:5]
array([3, 4])

>>> a[4:,4:]
array([[44, 45],
[54, 55]])

>>> a[:,2]
array([2,12,22,32,42,52])

>>> a[2::2,::2]
array([[20, 22, 24],
[40, 42, 44]])
```



Numeric Array

Unary/Binary Operation

```
>>> a=array((1,2,3,4))
>>> a
array([1, 2, 3, 4])
>>> a+=1
>>> a
array([2, 3, 4, 5])
>>> a*3
array([ 6,  9, 12, 15])
>>> b=array([[1,2,3,4],[5,6,7,8]])
>>> b
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
>>> b+a
array([[ 7, 11, 15, 19],
       [11, 15, 19, 23]])
```

Numeric Array

Ufunc: is a function that performs elementwise operations on data in ndarrays

```
>>> a
array([2, 3, 4, 5])
>>> pow(a,2)
array([ 4,  9, 16, 25])
```

TRIGONOMETRIC	
sin(x)	sinh(x)
cos(x)	cosh(x)
arccos(x)	
arccosh(x)	
arctan(x)	arctanh(x)
arcsin(x)	arcsinh(x)
arctan2(x,y)	

OTHERS	
exp(x)	log(x)
log10(x)	sqrt(x)
absolute(x)	conjugate(x)
negative(x)	ceil(x)
floor(x)	fabs(x)
hypot(x,y)	fmod(x,y)
maximum(x,y)	minimum(x,y)

Speeding Up Python with Numpy

This example solves Laplace's equation over a 2-d rectangular grid using using an iterative finite difference scheme: $\Delta u = 0$

```
class Grid:
    """A simple grid class that stores the details and
    solution of the computational grid."""
    def __init__(self, nx=10, ny=10, xmin=0.0, xmax=1.0,
    ymin=0.0, ymax=1.0):
    …
    …

class LaplaceSolver:
    """A simple Laplacian solver that can use different
    schemes to solve the problem."""
        def numericTimeStep(self, dt=0.01):
        …
        def slowTimeStep(self, dt=0.01):
```

Full code: laplace_benchmark.py

Speeding Up Python with Numpy

Pure Python Computational Core

```
def slowTimeStep(self, dt=0.01):
 g = self.grid
 nx, ny = g.u.shape
 dx2, dy2 = g.dx**2, g.dy**2
 dnr_inv = 0.5/(dx2 + dy2)
 u = g.u
err = 0.0
 for i in  range(1, nx-1):
    for j in range(1, ny-1):
        tmp = u[i,j]
        u[i,j] = ((u[i-1, j] + u[i+1, j])*dy2 +(u[i, j-1]
                                 + u[i,j+1])*dx2)*dnr_inv
        diff = u[i,j] - tmp
        err += diff*diff
 return numpy.sqrt(err)
```

Speeding Up Python with Numpy

Numpy Python Computational Core

```
def numericTimeStep(self, dt=0.0):
        """Takes a time step using a NumPy expression."""
        g = self.grid
        dx2, dy2 = g.dx**2, g.dy**2
        dnr_inv = 0.5/(dx2 + dy2)
        u = g.u
        g.old_u = u.copy() # needed to compute the error.

        # The actual iteration
        u[1:-1, 1:-1] = ((u[0:-2, 1:-1] + u[2:, 1:-1])*dy2 +
                         (u[1:-1,0:-2] + u[1:-1, 2:])*dx2)*dnr_inv

        return g.computeError()
```

Speeding Up Python with Numpy

```
%run
C:/Users/invernizzi/Documents/CORSI/2013/SCUOLA_VISUALIZZA
ZIONe/Esempi/laplace_benchmark.py

Solving Equation
Doing 100 iterations on a 500x500 grid
Elapsed Time SlowTimeStep  100.920565005 s
Elapsed Time  NumericTimeStep 0.771977486264 s
```

130 X Faster !!

The entire for i and j loops have been replaced in `NumericTimeStep` by a single NumPy expression. NumPy expressions operate elementwise.

The beauty of the expression is that its completely done in C. This makes the computation *much* faster.

IPython

A System for Interactive Scientific Computing

Why IPython?

Python Shell Limitation

No formatting
No syntax highlighting
No code completion
No function signature assistence
...

IPython

Command history
Tab auto-completion.
In-line editing of code.
Object introspection, and automatic extract of documentation
Good interaction with operating system shell.

```
IPython
File  Edit  View  Kernel  Magic  Window  Help
kernel 1 | kernel 2

In [2]: help
Out[2]: Type help() for interactive help, or help(object) for help about object.

In [3]: %magic

In [4]: %history
%quickref
help
%magic
%history

In [5]: import numpy as np

In [6]: def myfunc():
   ...:     return 1

In [7]: %who
myfunc   np      usage

In [8]: %whos
Variable   Type        Data/Info
--------------------------------
myfunc     function    <function myfunc at 0x04F91DF0>
np         module      <module 'numpy' from 'C:\<...>ages\numpy\__init__.pyc'>
usage      module      <module 'IPython.core.usa<...>\IPython\core\usage.pyc'>

In [9]:
```

Ipython Magic

- IPython will treat any line whose first character is a % as a special call to a ‘magic’ function. These allow you to control the behavior of IPython itself, plus a lot of system-type features.

%autocall: Insert parentheses in calls automatically, e.g. range 3 5
%debug: Debug the current environment
%edit: Run a text editor and execute its output
%gui: Specify a GUI toolkit to allow interaction while its event loop is running
%history: Print all or part of the input history
%loadpy: Load a Python file from a filename or URL (!)
%logon and **%logoff**: Turn logging on and off
%macro: Names a series of lines from history for easy repetition
%pylab: Loads numpy and matplotlib for interactive use
%quickref: Load a quick-reference guide
%recall: Bring a line back for editing
%rerun: Re-run a line or lines
%run: Run a file, with fine control of its parameters, arguments, and more
%save: Save a line, lines, or macro to a file
%timeit: Use Python’s timeit to time execution of a statement, expression, or block

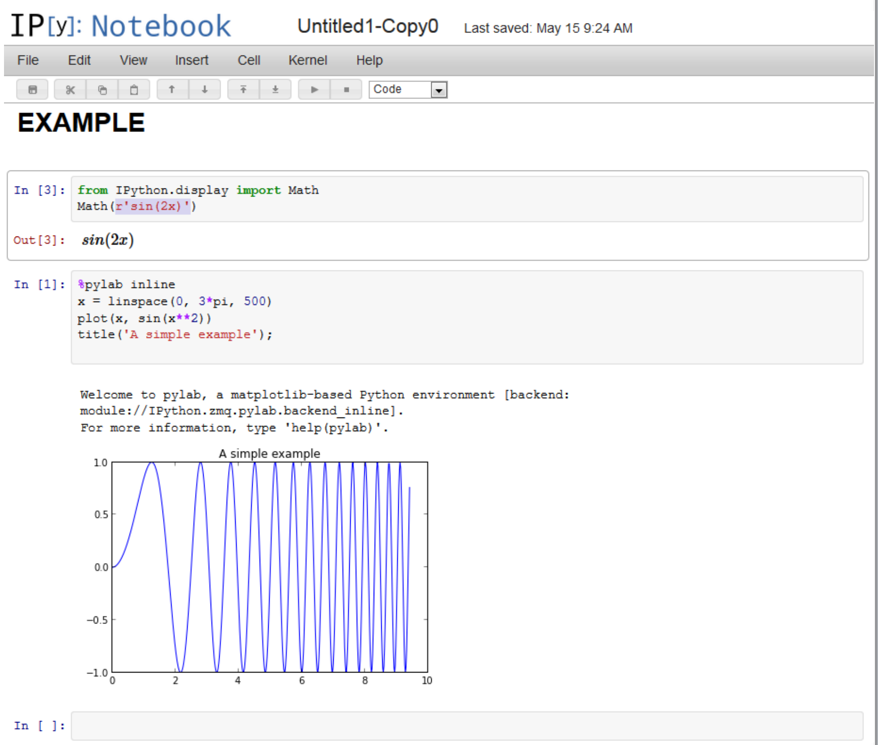
More on IPython

IPython NoteBook

The IPython Notebook is a web-based interactive computational environment where you can combine code execution, text, mathematics, plots and rich media into a single document.

Embedding IPython

It is possible to start an IPython instance inside your own Python programs. This allows you to evaluate dynamically the state of your code, operate with your variables, analyze them



Matplotlib

Plotting and Graphing tool in Python

Matplotlib

"Matplotlib tries to make easy things easy and hard things possible"

John Hunting

Matplotlib is a powerful Python module to creating 2D figures. Matplotlib was modeled on MATLAB, because graphing is something that MATLAB do very well.

What are the points that built the success of Matplotlib?

- It uses Python: MATLAB lacks many of the features of general purpose languages
- It is opensource
- It is cross-platform: can run on Linux,Windows, Mac OS and Sun Solaris
- It is very customizable and extensible
- Plots should look great - publication quality.
- Postscript output for inclusion with TeX documents
- Embeddable in a graphical user interface for application development
- Code should be easy enough that I can understand it and extend it
- Making plots should be easy

Matplotlib

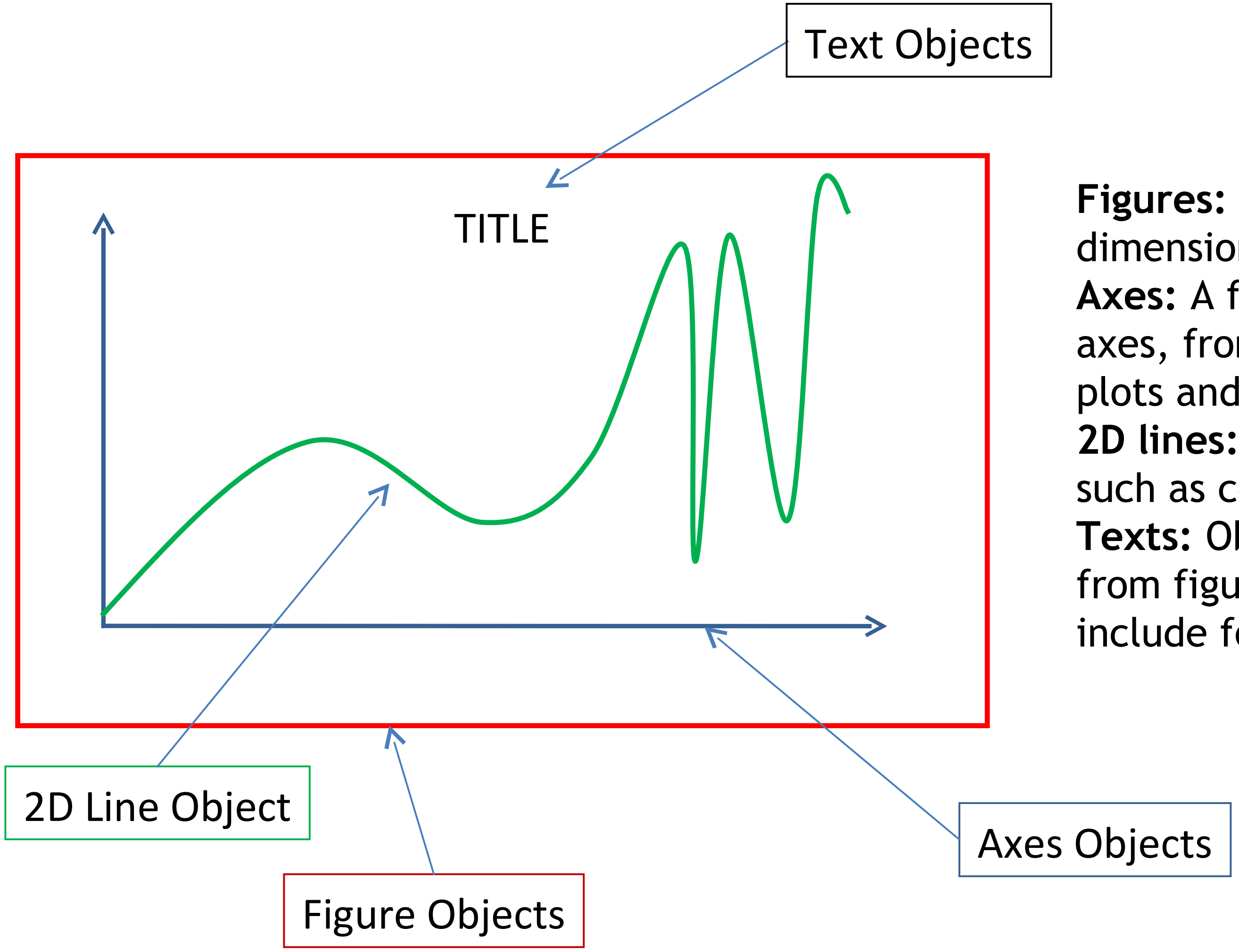
The Matplotlib code is conceptually divided into three parts:

•the *pylab interface*: the set of functions provided by matplotlib.pylab which allow the user to create plots with code quite similar to MATLAB figure generating code

•The *matplotlib frontend* or *matplotlib API* : the set of classes that do the heavy lifting, creating and managing figures, text, lines, plots.

•The *backends* are device dependent drawing devices that transform the frontend representation to hardcopy or a display device. Example backends: PS hardcopy, SVG hardcopy, PNG output, GTK GTKAgg, PDF, WxWidgets, Tkinter etc

Matplotlib: main objects



Figures: The plot itself, include dimensions and resolution
Axes: A figure can have multiple axes, from which can be defined plots and text
2D lines: 2D lines have properties such as color, thickness, etc
Texts: Objects which can be used from figures or axes. Properties include font, colour, etc.

How to work with Matplotlib

Matplotlib is designed for object oriented programming. This allows to define objects such as colours, lines, axes, etc. Plots can also be designed using functions, in a Matlab-like interface.

There are three ways to use Matplotlib:

pyplot: provides an interface to the underlying plotting library in matplotlib. This means that figures and axes are implicitly and automatically created to achieve the desired plot.

pylab: A module to merge Matplotlib and NumPy together in an

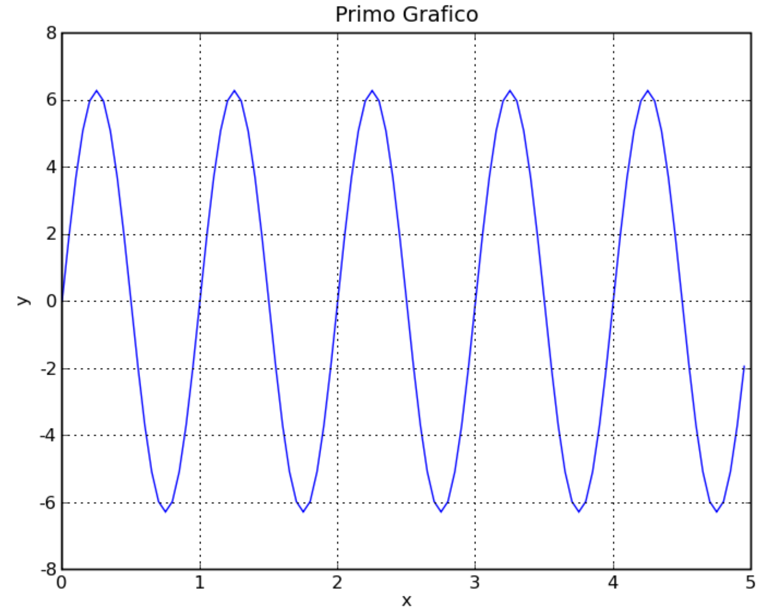
environment closer to MATLAB = pyplot+numpy

Object-oriented way: The Pythonic way to interface with Matplotlib

NOTE: The object-oriented is generally preferred for non-interactive plotting (i.e., scripting). The pylab interface is convenient for interactive calculations and plotting.

Simple Example

```
>>>from pylab import *
>>>t=arange(0,5,0.05)
>>>f=2*pi*sin(2*pi*t)
>>>plot(t,f)
>>>grid()
>>>xlabel('x')
>>>ylabel('y')
>>>title('Primo grafico')
>>>show()
```



The function show() opens up an interactive window with the plot.
The function show() starts a TK mainloop that blocks the mainloop of the program.
You need to close the new window to continue the execution of the script.

Interactive mode

IPython is the designed Python shell for interactive script. If we are in interactive mode, then the figure is redrawn on every plot command. If we are not in interactive mode, a figure state is updated on every plot command, but the figure is actually drawn only when an explicit call to draw() or show() is made.

In order to use IPython for interactive plotting, start it in *pylab mode*.

```
>>>ipython pylab
```

Or from the IPython shell using magic word %pylab

```
IPython 0.13.1 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.
%guiref   -> A brief reference about the graphical user interface.

%pylab
```

NOTE: interactive property is available in `rcParams` dictionary

pylab , pyplot

```
>>>from pylab import *
>>>t=arange(0,5,0.05)
>>>f=2*pi*sin(2*pi*t)
>>>plot(t,f)
>>>grid()
>>>xlabel('x')
>>>ylabel('y')
>>>title('Primo grafico')
>>>show()
```

```
import numpy as np
import matplotlib.pyplot as plt
t=np.arange(0,5,0.05)
f=2*np.pi*np.sin(2*np.pi*t)
plt.plot(t,f)
plt.grid()
plt.xlabel('x')
plt.ylabel('y')
plt.title('Primo grafico')
plt.show()
```

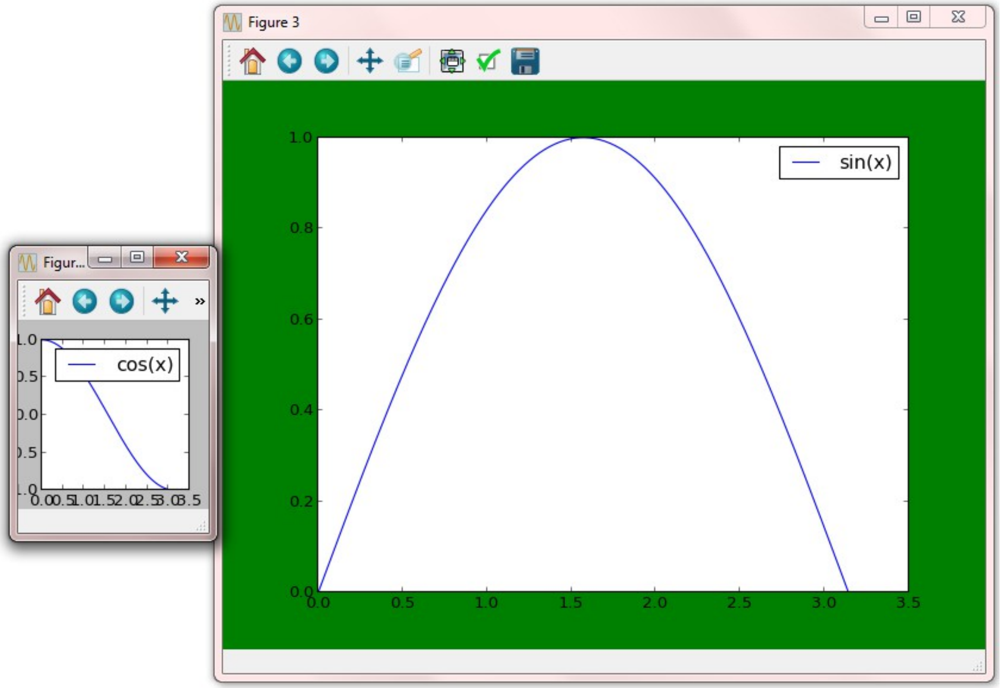
pylab mode: is convenient for interactive calculations and plotting.

pyplot mode: is generally preferred for non-interactive plotting

Figure

When the figure object, is defined, some properties such as dimensions and resolution, borders colour, etc can be set

```
>>> x=arange(0,pi,0.01)
>>> y=sin(x)
>>> y2=cos(x)
>>> figure(facecolor='g')
>>> plot(x,y,label='sin(x)')
>>> legend()
>>> figure(figsize=[3,3])
>>> plot(x,y2,label='cos(x)')
>>> legend()
>>> close(1)
>>> close('all')
```



Creating a 2D plot

- The function `plot()` is highly customizable, accommodating various options, including plotting lines and/or markers, line widths, marker types and sizes, colors, and legend to associate with each plot.

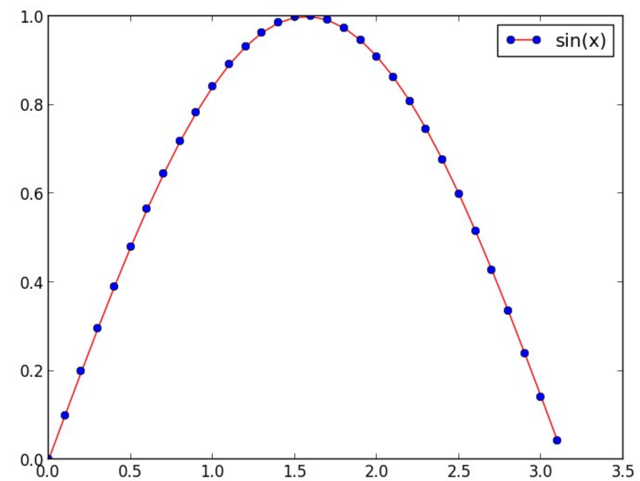
plot(line2d , [properties line2d])

color	**keyword color**: 'b' blue, 'r' red, 'g' green, 'y' yellow, 'k' black, 'w' white, 'c' cyan, 'm' magenta
label	line label used for legends
linestyle	line style: '' no line, '--' dashed, '-' continuous, ':' doted, '.-' dash-dot
linewidth	line width: float value in pixels
marker	marker type: '.' Point, 'o' circle, 'D' diamond, '^' triangle, 's' square, '*' star, '+' plus, 'h' hexagon,
markersize	marker size: float value in pixels
markeredgecolor	marker edge color: cf color
markerfacecolor	maker face color: cf color

Creating 2D Plot

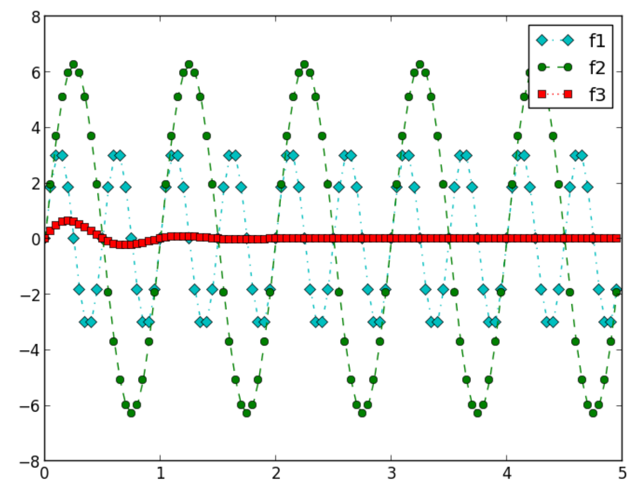
Setting line2D property

```
>>>x=arange(0,pi,0.1)
>>>plot(x,sin(x),marker='o',color='r',
markerfacecolor='b',label='sin(x)')
>>>legend()
```



Creating Multi-line plot

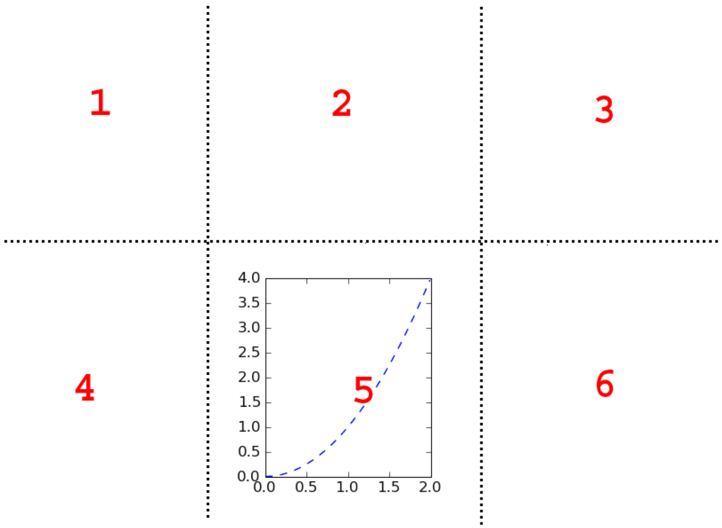
```
>>> t=arange(0,5,0.05)
>>> f=2*pi*sin(2*pi*t)
>>> f2=sin(2*pi*t)*exp(-2*t)
>>> plot(t,f,'g--o',t,f2,'r:s‛)
>>> hold(True)
>>> f3=2*pi*sin(2*pi*t)*cos(2*pi*t)
>>> plot(t,f3,'c-.D',label='f3')
>>> legend(('f1','f2‛,’f3’))
```



Creating sub-plot

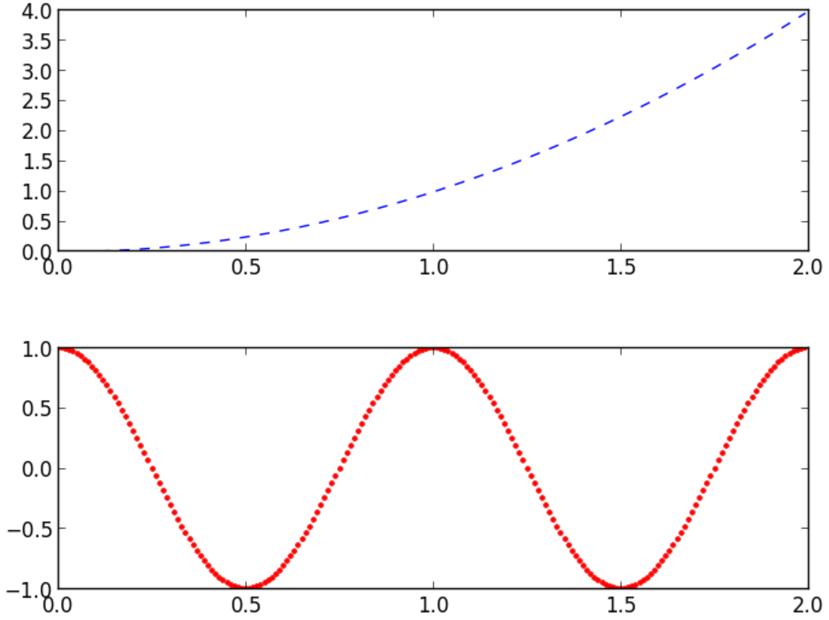
subplot() allows to divide the figure in a grid with specified number of columns and rows. Then we can place our plot in the desired zone.

```
subplot(numRows, numColumns, PlotIndex)
```



Creating sub-plot

```
from pylab import *
x = arange (0, 2.0, 0.01)

subplot(2, 1, 1)
plot(x, x ** 2, 'b--')

subplot(2, 1, 2)
plot(x, cos(2*pi*x), 'r.')

subplots_adjust(hspace = 0.5)
show()
```

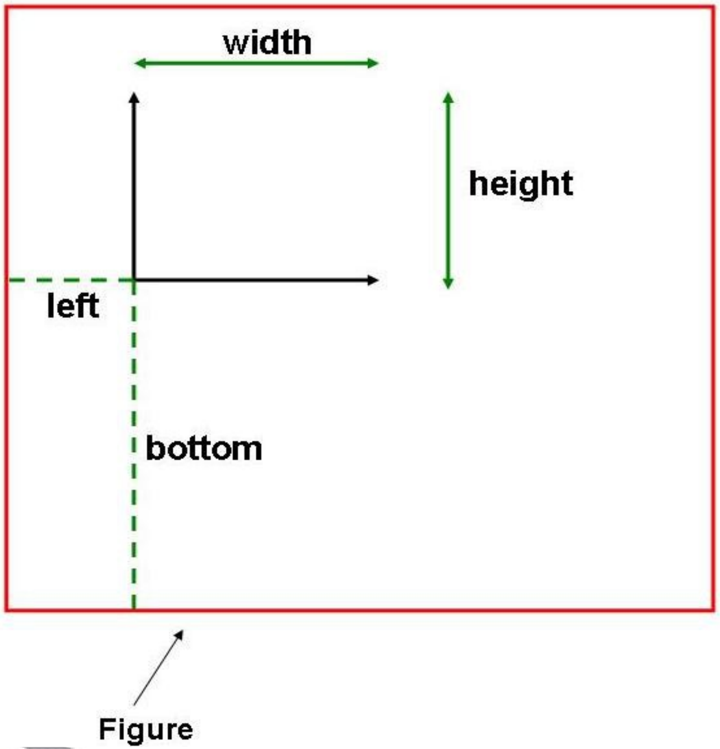


Axes

When you create a subplot, an axis instance is automatically created. The axes can be defined as follows: ax = subplot(111)

To create an axis:

```
axes([bottom_left_corner_x, bottom_left_corner_y, width, height])
```

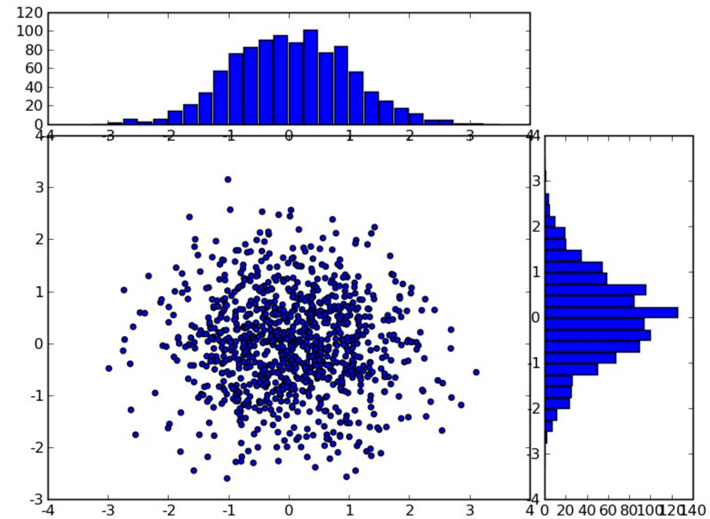


It is possible to modify axes with:

```
axis([xmin,xmax,ymin,ymax])
grid()
xticks(location,label)
```

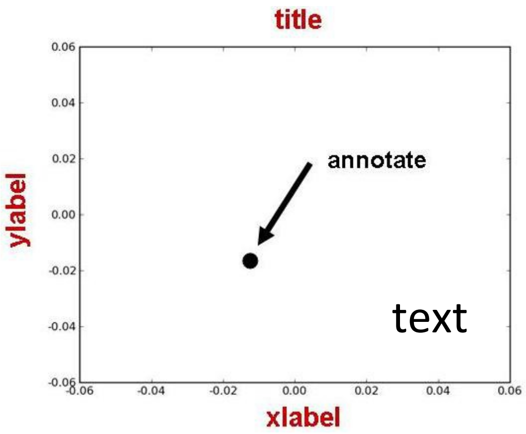
Axes

```
x = numpy.random.randn(1000)
y = numpy.random.randn(1000)
axscatter = axes([0.1,0.1,0.65,0.65])
axhistx = axes([0.1,0.77,0.65,0.2])
axhisty = axes([0.77,0.1,0.2,0.65])

axscatter.scatter(x, y)
draw()
binwidth = 0.25
xymax = max( [max(fabs(x)), max(fabs(y))] )
lim = ( int(xymax/binwidth) + 1) * binwidth
bins = arange(-lim, lim + binwidth,
binwidth)
axhistx.hist(x, bins=bins)
draw()
axhisty.hist(y, bins=bins,
orientation='horizontal')
draw()
```



Text

There are several option to annotate a graph with text.



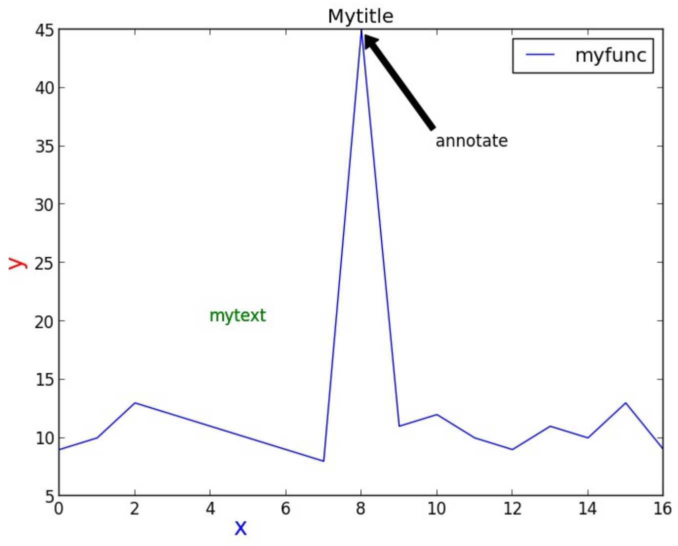

```
xlabel (s, *args, **kwargs)
ylabel (s, *args, **kwargs)
title (s, *args, **kwargs)
annotate(s, xy, xytext=None,
textcoords='data',arrowprops=None,**props)
text(x, y, s, fontdict=None,**kwargs)
```

Is is possible to create text object with several options

fontsize	[size in points | 'xx-small' | 'x-small' | 'small' | 'medium' | 'large' | 'x-large' | 'xx-large']
fontfamily	[FONTNAME | 'serif' | 'sans-serif' | 'cursive' | 'fantasy' | 'monospace']
fontstyle	['normal' | 'italic' | 'oblique']
fontweight	[a numeric value in range 0-1000 | 'ultralight' | 'light' | 'normal' | 'regular' | 'book' | 'medium' | 'roman' | 'semibold' | 'demibold' | 'demi' | 'bold' | 'heavy' | 'extra bold' | 'black']
fontstretch	[a numeric value in range 0-1000 | 'ultra-condensed' | 'extra-condensed' | 'condensed' | 'semi-condensed' | 'normal' | 'semi-expanded' | 'expanded' | 'extra-expanded' | 'ultra-expanded']
color	[matplotlib color]
position	[(x,y)], in range 0-1
rotation	[angle in degrees | 'vertical' | 'horizontal']
verticalalignment	['top' | 'center' | 'bottom']
horizontalalignment	['left' | 'center' | 'right]

Text

```
>>> x=[9,10,13,12,11,10,9,8,45,11,12,10,9,
11,10,13,9]
>>> plot(x,label='myfunc')
>>> legend()
>>> title('Mytitle')
>>> ylabel('y',fontsize='medium',color='r')
>>> xlabel('x',fontsize='x-
large',color='b',position=(0.3,1))
>>> text(4,20,'mytext',
color='g',fontsize='medium')
>>> annotate('annotate',xy=(8,45),xytext=(10,
35),arrowprops=dict(facecolor='black',shrink=0.
05))
```



To render mathematical expressions, use a raw string and enclose your mathematical expression with signs $. For Greek letters, start with a slash followed by the name of the letter.

xlabel(r'$y_i=2\pi \sin(2\pi x)$') is equal to $y_i = 2\pi \sin(2\pi x)$

Images File

There are several ways you can use matplotlib:

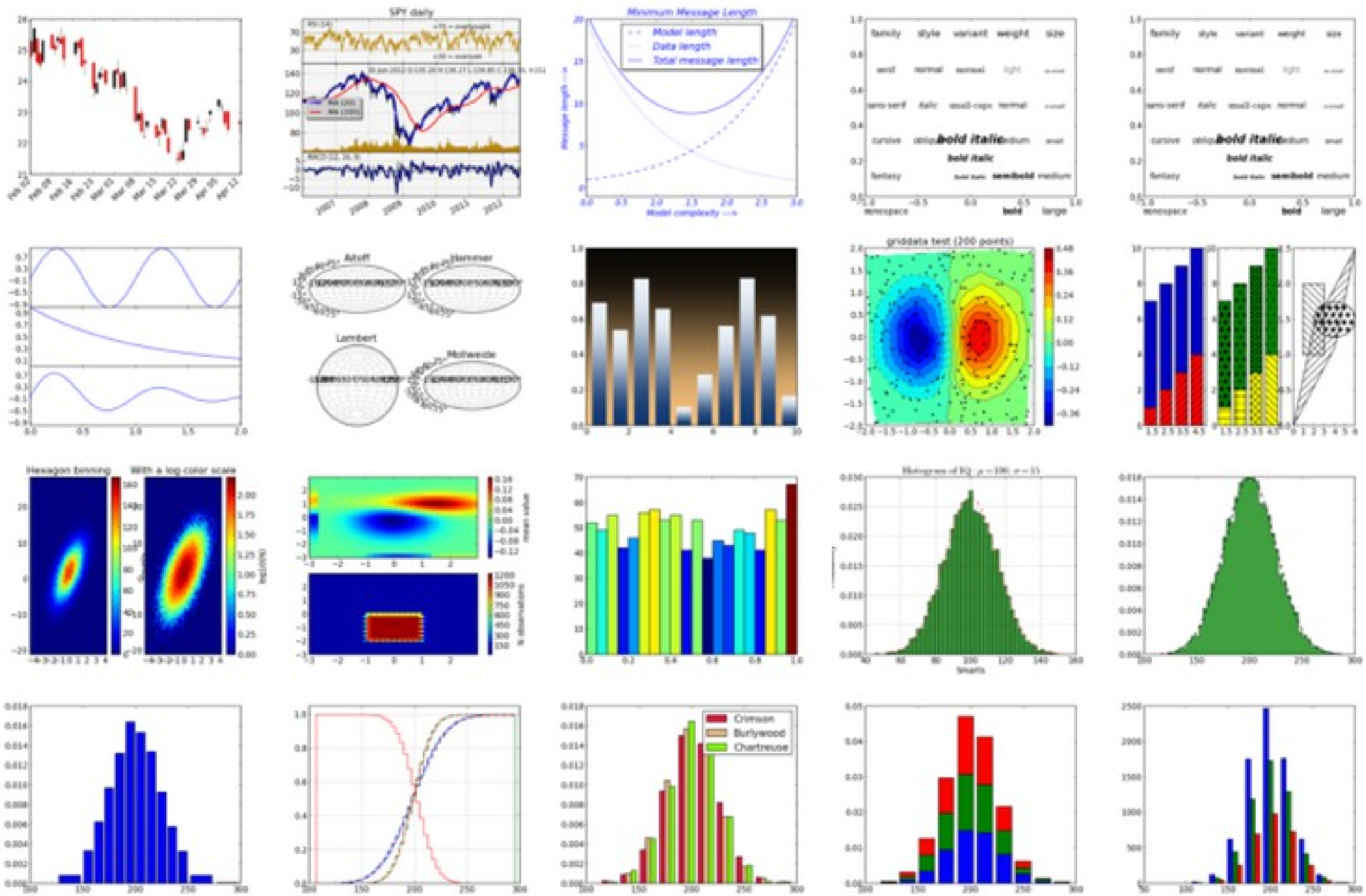
- Run it interactively with the Python shell
- Automatically process data and generate output in a variety of file format
- Embed it in a graphical user interface, allowing the user to interact with an application to visualize data.

Displaying a plot can be time consuming, especially for multiple and complex plots. Plots can be saved without being displayed using the **savefig()** function:

```
x = arange(0,10,0.1)
plot(x, x ** 2)
savefig('C:/myplot.png')
```

Matplotlib Gallery

- http://matplotlib.sourceforge.net/gallery.html



mplot3d

- The mplot3d toolkit adds simple 3D plotting capabilities to matplotlib by supplying an axes object that can create a 2D projection of a 3D scene. The resulting graph will have the same look and feel as regular 2D plots.

Lorentz attractor

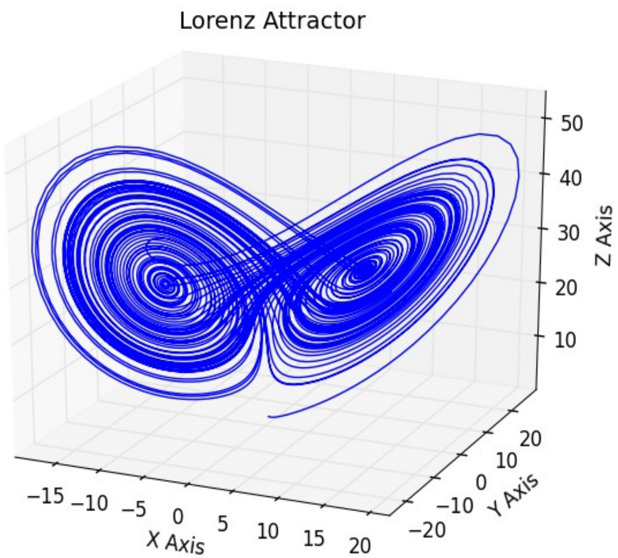
$$\frac{\mathrm{d}x}{\mathrm{d}t} = \sigma(y - x),$$
$$\frac{\mathrm{d}y}{\mathrm{d}t} = x(\rho - z) - y,$$
$$\frac{\mathrm{d}z}{\mathrm{d}t} = xy - \beta z.$$

mplot3d

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

def lorenz(x, y, z, s=10, r=28, b=2.667) :
    x_dot = s*(y - x)
    y_dot = r*x - y - x*z
    z_dot = x*y - b*z
    return x_dot, y_dot, z_dot

dt = 0.01
stepCnt = 10000
# Need one more for the initial values
xs = np.empty((stepCnt + 1,))
ys = np.empty((stepCnt + 1,))
zs = np.empty((stepCnt + 1,))
# Setting initial values
xs[0], ys[0], zs[0] = (0., 1., 1.05)
```

mplot3d

```
# Stepping through "time".
for i in
(stepCnt) :
    # Derivatives of the X, Y, Z state
    x_dot, y_dot, z_dot = lorenz(xs[i], ys[i], zs[i])
    xs[i + 1] = xs[i] + (x_dot * dt)
    ys[i + 1] = ys[i] + (y_dot * dt)
    zs[i + 1] = zs[i] + (z_dot * dt)
fig = plt.figure()
ax = fig.gca(projection='3d')
ax.plot(xs, ys, zs)
ax.set_xlabel("X Axis")
ax.set_ylabel("Y Axis")
ax.set_zlabel("Z Axis")
ax.set_title("Lorenz Attractor")
plt.show()
```



More on Matplotlib

It is possible to create animated graph:

simple_animation.py

lorentz_animation.py

It is possible to interact with the object graph:

mouse_event.py

picker_example.py

It is possible to customize the plot widget enabling action

matplotlib_radiobutton.py

matplotlib_checkbutton.py

More on Scientific Visualization in Python

http://wiki.python.org/moin/NumericAndScientific/Plott ing

Plotting Tools

- Matplotlib is an Open Source plotting library designed to support interactive and publication quality plotting with a syntax familiar to Matlab users. Its interactive mode supports multiple windowing toolkits (currently: GTK, Tkinter, Qt, and wxWindows) as well as multiple noninteractive backends (PDF, postscript, SVG, antigrain geometry, and Cairo). Plots can be embedded within GUI applications or for non-interactive uses without any available display in batch mode. Matplotlib provides both a Matlab-like functional interface as well as an object oriented interface. IPython has a "pylab" mode which is specifically designed for interactive plotting with matplotlib.
- Veusz is a GPL scientific plotting package written in Python and PyQt, designed to create publication-quality output. Graphs are built up from simple components, and the program features an integrated command-line, GUI and scripting interface. Veusz can also be embedded in other Python programs, even those not using PyQt.
- Visvis is a pure Python library for visualization of 1D to 4D data in an object oriented way. Essentially, visvis is an object oriented layer of Python on top of OpenGl, thereby combining the power of OpenGl with the usability of Python. A Matlab-like interface in the form of a set of functions allows easy creation of objects (e.g. plot(), imshow(), volshow(), surf()).
- Chaco is a device-independent 2D plotting package based on a DisplayPDF API. It supports **fast** vector graphics rendering for **interactive data analysis** (read: fast live updating plots) and custom plot construction. Chaco is easy to embed in python GUI applications (wxWindows, Qt) and provides nice abstractions for overlays and tools (select regions, zoom/pan, cross-hairs, labels, data inspectors, etc.). Chaco is able to output to any raster format supported by PIL, as well as PDF, PostScript and SVG backends. See the gallery for screenshots and code examples.
- diaGrabber is based on PyQtGraph and allows you to read, filter, process, interpolate and plot n-dimensional values from different sources (like libreOffice- or csv-files) and variable size. Through interactive reading it's also possible to evaluate streams in a kind of 'software-oscilloscope'.
- KonradHinsen has some plotting support in his ScientificPython package, for example TkPlotCanvas.
- Michael Haggerty has a Gnuplot module that interfaces with the GNUPLOT package.
- plot_wrap A module by Mike Miller which wraps the functions in the GNU plotutils package.
- BLT BLT is an extension to the tk widgets that can produce X/Y plots and bar charts. The BLT package can be used through the Pmw package, a framework for the creation of megawidgets built on top of Tkinter.
- PyQwt is a set of Python bindings for the Qwt C++ class library which extends the Qt framework with widgets for scientific and engineering applications.
- GUIQwt is a Python library based on Qwt providing efficient 2D data-plotting features (curve/image visualization and related tools) for interactive computing and signal/image processing application development.
- DISLIN DISLIN is a high-level and easy to use graphics library for displaying data as curves, bar graphs, pie charts, 3D-color plots, surfaces, contours and maps. The software is available for several C, Fortran 77 and Fortran 90 compilers. For some operation systems, the programming languages Python and Perl are also supported by DISLIN. DISLIN is free for the Linux and FreeBSD operating systems and for the MS-DOS and Windows 95/NT compilers GCC, G77 and ELF90. Other DISLIN versions are available at low prices and can be tested free of charge.
- Mayavi Starting from Mayavi2, the 3D data visualization program Mayavi is fully scriptable from Python, can be integrated in larger applications, and exposes a simple pylab/matlab-like interface for plotting arrays.
- gdmodule GD is a graphics library for the creation of GIF pictures, written by Thomas Boutell. gdmodule is an Python extension for this library. It can do lines, arcs, fills, fonts and can also manipulate other GIF pictures. Included in the gdmodule is a graphing module,

A brief introduction to Mayavi

Mayavi2 seeks to provide easy and interactive visualization of 3D data, or 3D plotting. It does this by the following:

- an (optional) rich user interface with dialogs to interact with all data and objects in the visualization.
- a simple and clean scripting interface in Python, including ready to use 3D visualization functionality similar to matlab or matplotlib or an object-oriented programming interface.
- use the power of VTK without forcing you to learn it.

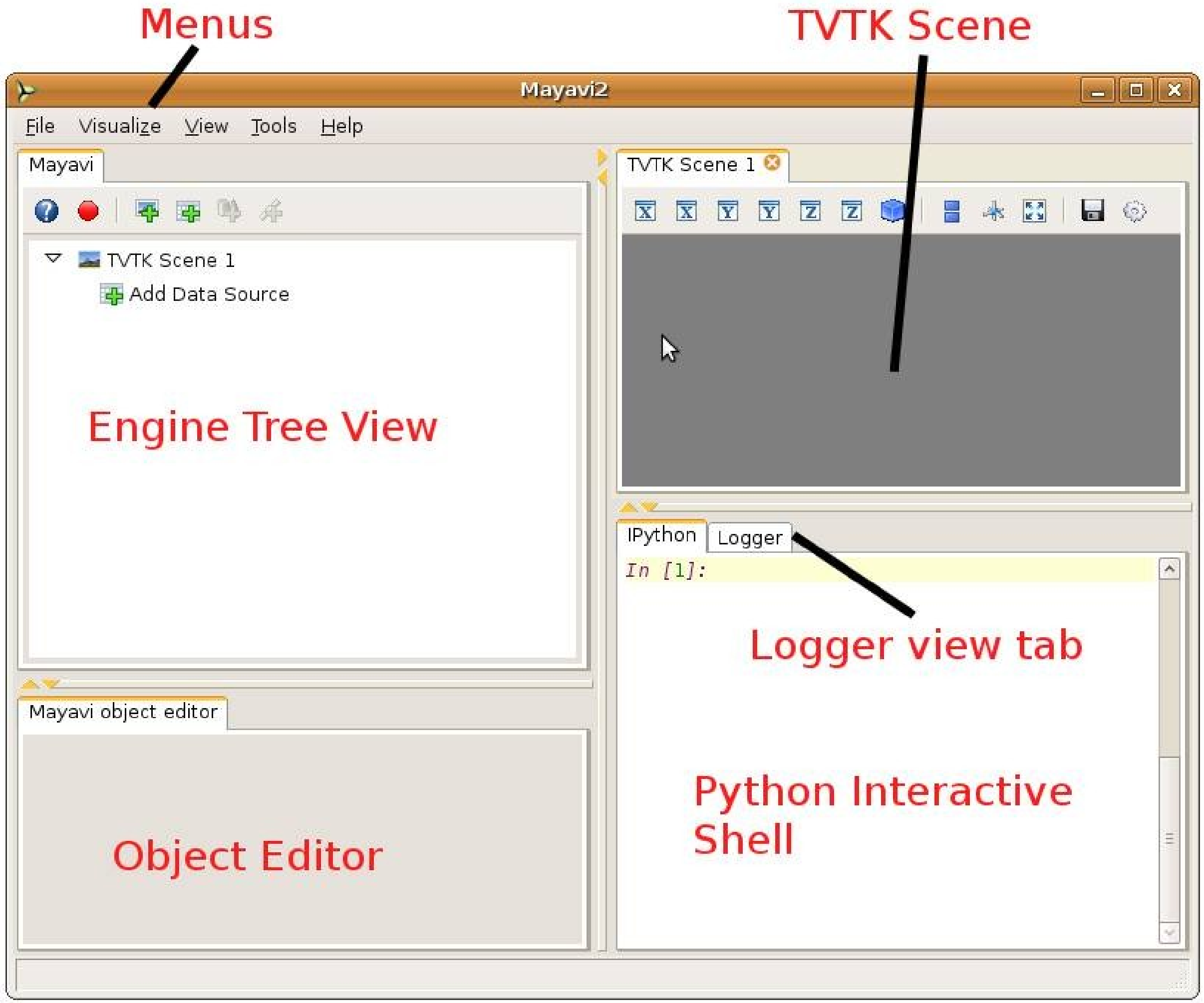
A brief introduction to Mayavi

So the user can choose three different ways to use Mayavi:

- Use the mayavi2 application completely graphically.
- Use Mayavi as a plotting engine from simple Python scripts, for example from Ipython, in combination with numpy.
- (Advanced) Script the Mayavi application from Python. The Mayavi application itself features a powerful and general purpose scripting API that can be used to adapt it to your needs.

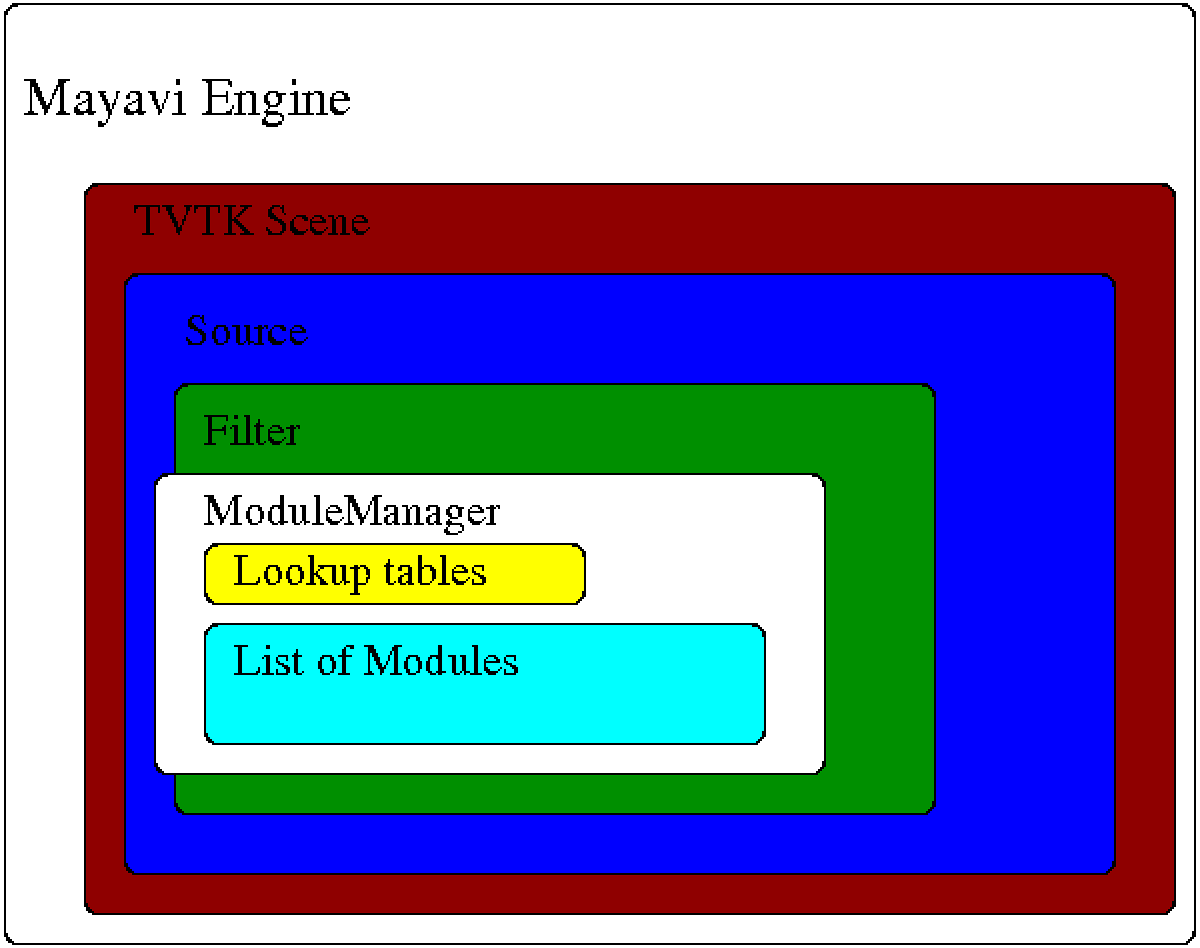
Mayavi Interface

- The interactive application, mayavi2, is an end-user tool that can be used without any programming knowledge
- Mayavi presents a simplified pipeline view of the visualization.
- The application displays an interactive Python shell, where Python commands can be entered for immediate execution.



Mayavi Engine

- The Engine manages a collection of Scene.
- In each Scene, a user may have created any number of Source
- A Source object can further contain any number of Filter or ModuleManager objects

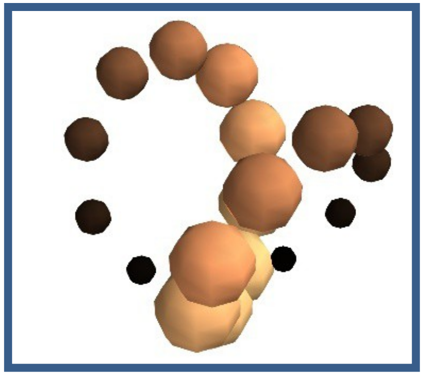


Simple Python Scripting with Mayavi

- Mayavi can also be used through a simple and yet powerful scripting API, providing a workflow similar to that of MATLAB or Mathematica.
- Mayavi's `mlab` scripting interface is a set of Python functions that work with numpy arrays and draw some inspiration from the MATLAB and matplotlib plotting functions. It can be used interactively in IPython, or inside any Python script or application.
- There are a lot of parallels between matplotlib and mayavi:
  - there exists huge object-oriented library, allowing you to control even the smallest detail in a plot.
  - there exists a module around that library called mlab, similar (and in fact inspired by) pylab.

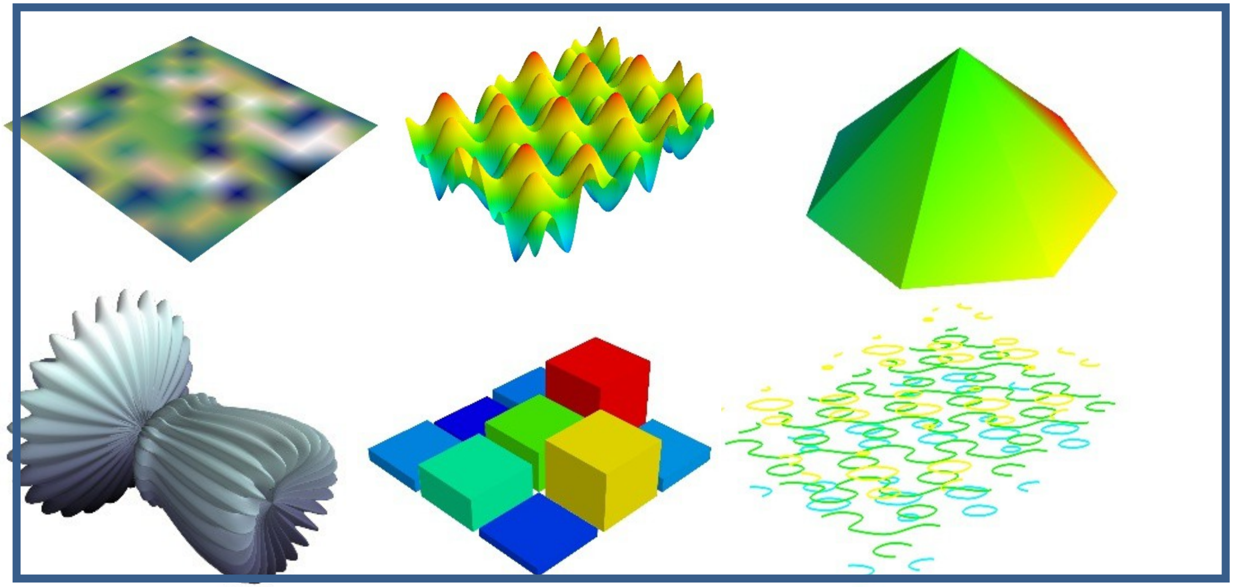
mlab

0D

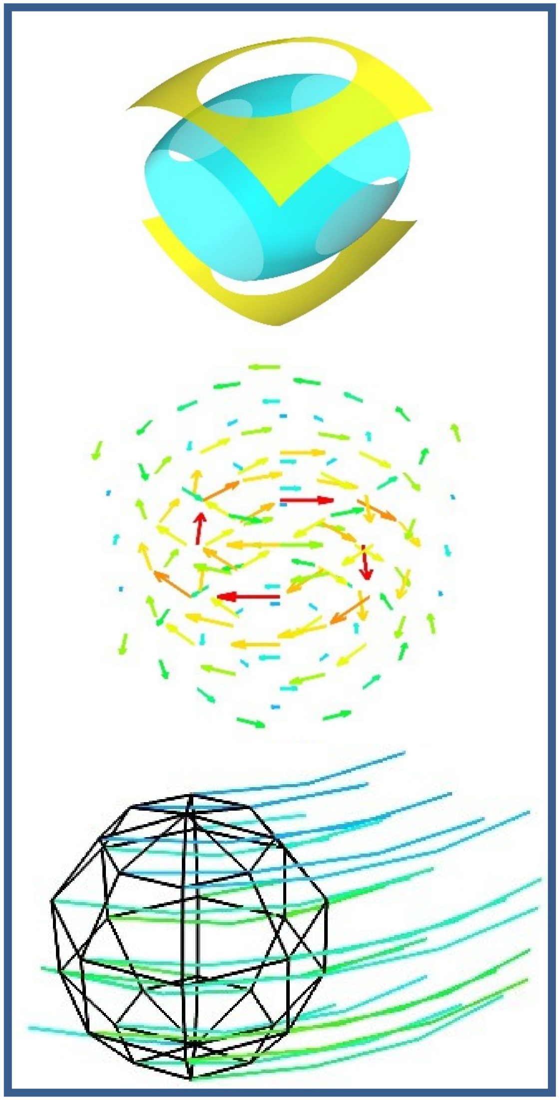


1D

2D



3D

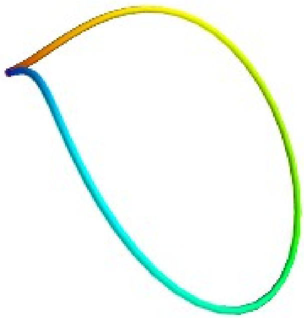


mlab

Simple problems should have simple solutions

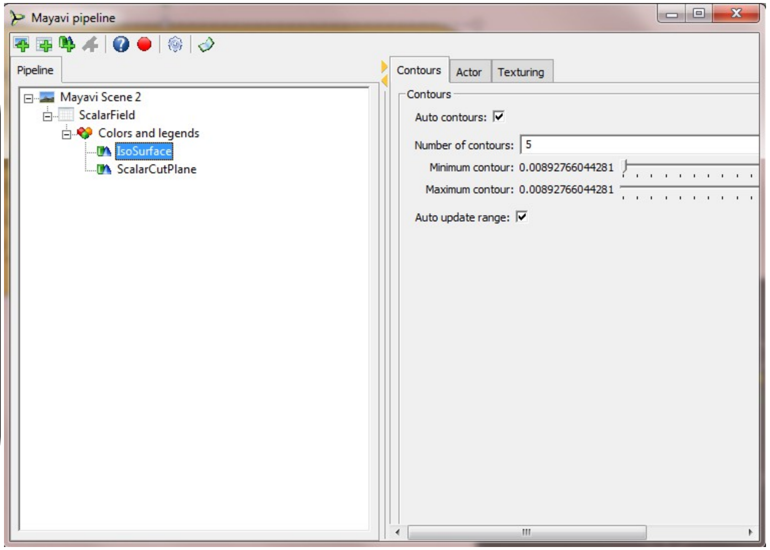
```
>>> from numpy import *
>>> t = linspace (0,2*pi,50)
>>> u = cos(t)* p i
>>> x,y,z =sin(u),cos(u),sin(t)
>>> mlab.points3d(x, y, z)
```

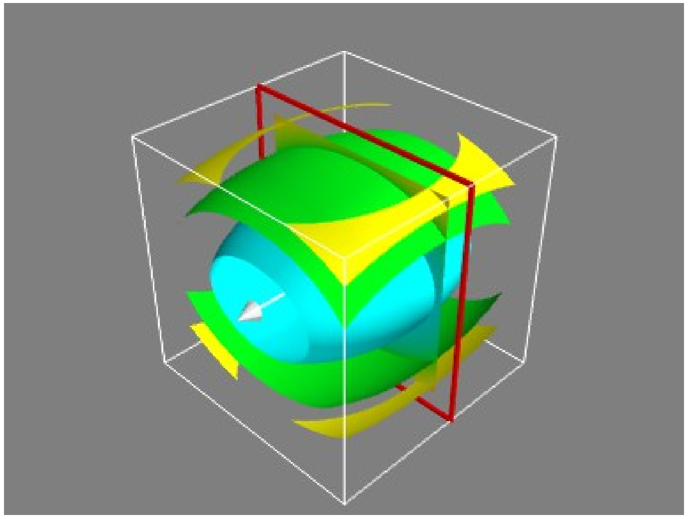
```
>>> from numpy import *
>>> t = linspace (0,2*pi,50)
>>> u = cos(t)* p i
>>> x,y,z =sin(u),cos(u),sin(t)
>>> mlab.plot3d(x, y, z, t )
```



mlab managing the pipeline

```
x,y,z = ogrid [ -5:5:100 j , -5:5:100 j
, -5:5:100 j ]
scalars = x*x*0.5 + y*y + z*z*2.0
obj = mlab.contour3d( scalars ,
opacity=0)
mlab.pipeline.scalar_cut_plane(obj )
mlab.show_pipeline()
```

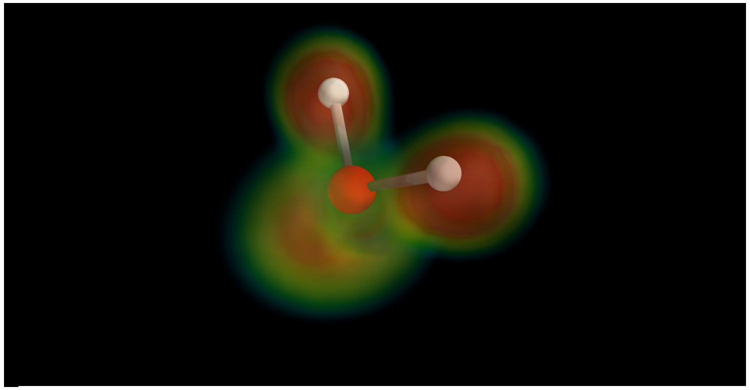




mlab managing pipeline

`chem.py`

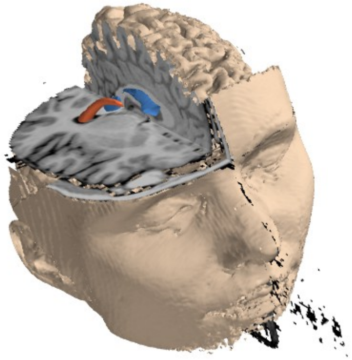
we display the H_2O molecule, and use volume rendering to display the electron localization function



`mri.py`

Viewing MRI data with cut plane and iso surface

We read an MRI scan, we turn it into a 3D numpy array and we visualize it



Brief Exercise on Matplotlib

In this exercise we'll plot some weather data read from a .csv file. Each row rapresents one day, and there are columns for min/mean/max temperature, dew point, wind speed, etc. We'll be plotting temperature and weather event data.

- read .csv file with numpy `loadtxt` function populating a numpy `array` only with min/max/mean temperature and weather event data.
- plot on the same figure using subplot function, max,min and mean temperature, add axis labels and title
- plot on the same figure using subplot function a trend line for mean/max/min temperature. Use numpy's polyfit function to add a trend line.
- plot on a new figure an event histogram counting occurred events per month as display in figure 2

Brief Exercise on Matplotlib

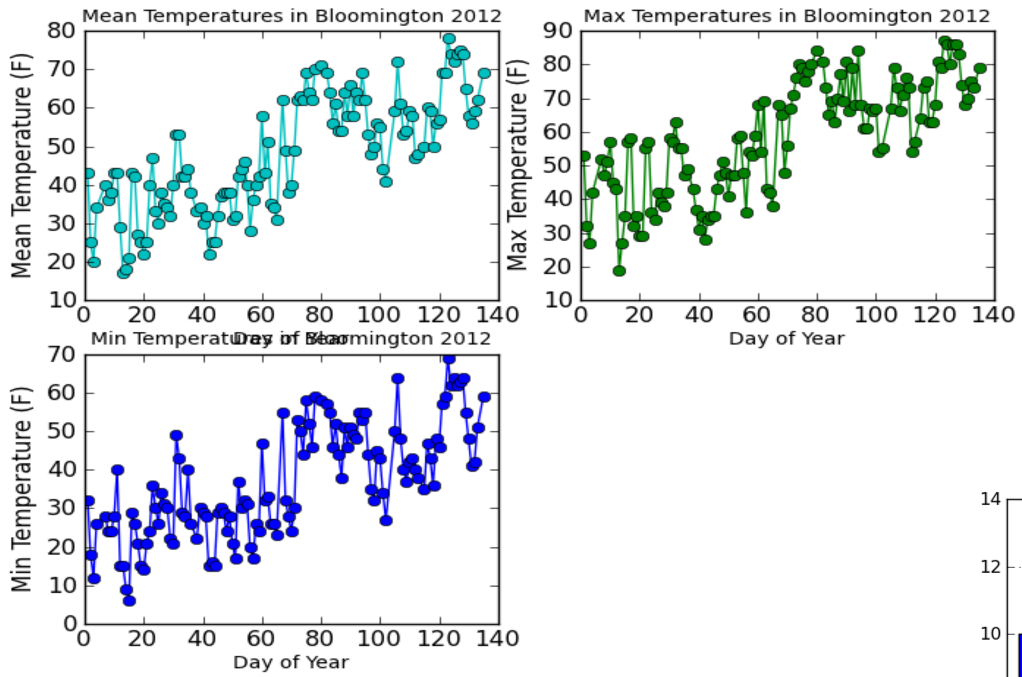


Figure 1

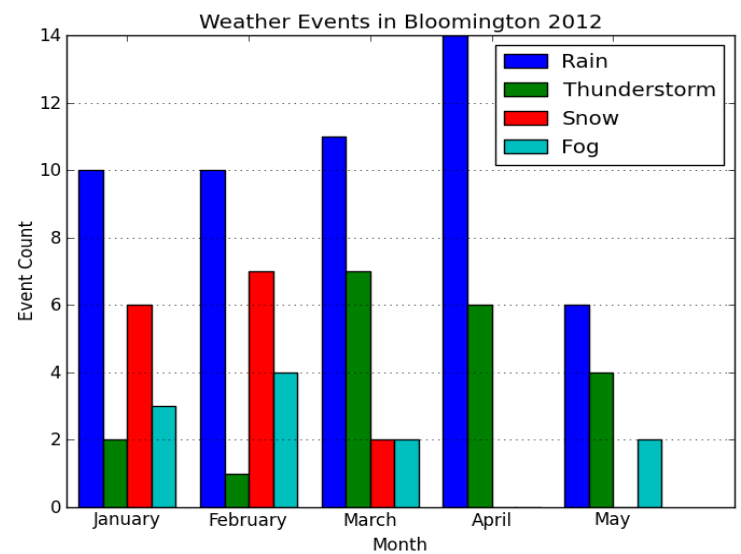


Figure 2