Introduction to NumPy, SciPy and Matplotlib

Scientific Computation With Python

Overview

- Important packages for scientific use:

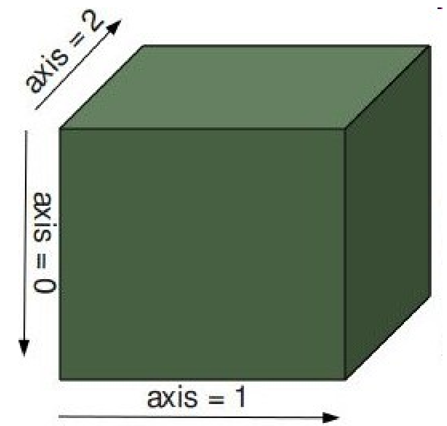
  - **NumPy** (handling and manipulation of large arrays)
  - **SciPy** (lots of user-friendly and efficient numerical routines)
  - **Matplotlib** (2D plotting library)

  => provide an **open source alternative to MATLAB**
  (http://www.scipy.org/NumPy_for_Matlab_Users)

- available at
  http://numpy.scipy.org/
  http://matplotlib.sourceforge.net/

NumPy Array

- provides a powerful N-dimensional array object:
  - table of items of **same type**
  - **more efficient** than python lists



- can be directly created from lists

```
>>> import numpy as np
>>> a = np.array ( [1,2,3,4] )
```

$$\vec{a} = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix}$$

```
>> import numpy as np
>> M = np.array ([ [1,2],[3,4] ] )
```

$$M = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

NumPy Array Creation

- arange() : improved range() – function

```
>> a = np.arange ( 0, 0.4, 0.1 )
```

$$\vec{a} = \begin{pmatrix} 0.0 \\ 0.1 \\ 0.2 \\ 0.3 \end{pmatrix}$$

- zeros(), ones() : fill with zeros or ones

```
>> M1 = np.ones ( (2,2) )
>> M2 = np.zeros ( (2,3) )
```

$$M1 = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} \quad M2 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

! the shape has to be specified !

- eye() : identity

```
>> M = np.eye ( 3 )
```

$$M = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

- rand(), randn() : random matrix

```
>> M = np.random.rand ( 2,2 )
```

$$M = \begin{pmatrix} 0.09833 & 0.94981 \\ 0.01581 & 0.34234 \end{pmatrix}$$

Indexing & Slicing

- 1-D arrays can be index, sliced and iterated like lists

```
>> a = np.arange ( 0, 0.4, 0.1 )

>> a[0]
0.0

>> a[1:3]
[0.1, 0.2]
```

$$\vec{a} = \begin{pmatrix} 0.0 \\ 0.1 \\ 0.2 \\ 0.3 \end{pmatrix}$$

- N-D arrays can have one index per axis

$$M = \begin{pmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{pmatrix}$$

```
>> M [0,2]
2

>> M [:,1]
[1,4,7]

>> M[:-1,::-1]
[  [2,1,0],
   [5,4,3] ]
```

- not specified axes considered complete slices

```
>> M[1] # eqv. To M[1,:]
[3,4,5]
```

Unary Operations

- many unary operations are implemented as methods of array class:

```
>> M = np.array ( [ [1,2], [3,4] ] )
```

$$M = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

```
>> M.sum()
10

>> M.mean()
2.5

>> M.max()
4
```

"handle array like lists"

```
>> M.sum( axis=0 )
array( [4,6] )

>> M.mean( axis=0 )
array ( [2, 3] )

>> M.max( axis=0 )
array ([3,4])
```

axis can be specified

More examples: argmax(), argsort(), conjugate(), cumsum(), conj(), imag(), real(), transpose(), …

Properties of Arrays

```
a = np.array( [ [0,1,2,3,4], [5,6,7,8,9] ] )

a.shape
(2,5)
```

$$a = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 & 9 \end{pmatrix}$$

- access attributes of numpy array:
  - a.shape (the dimensions) is (2,5)
  - a.ndim (number of axis) is 2
  - a.size (total number of elements) is 10
  - a.dtype (datatype of elements) is int64
  - a.itemsize (size of element in bytes) is 8
- additonal datatypes:
  - int8, int16, int32, int64
  - float32, float64, float96
  - complex64, complex128, complex192
  - object
- can be specified with dtype

```
a = np.array( [ [0,1,2,3,4], [5,6,7,8,9] ], dtype=np.complex64 )
```

Basic Operations

- arithmetic operations apply **elementwise**
- **new** array created

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

$$B = \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix}$$

$$c = 5$$

```
>> A * B
[ [5,12],
  [21,32] ]

>> A – c
[ [5,12],
  [21,32] ]
```

```
>> A ** B
[ [1, 64],
  [2187, 65536] ]

>> A<3
[ [True, False],
  [False, False] ]
```

matrix product:

```
>> np.dot (A,B)
[ [19,22],
  [43,50] ]
```

some operators act in place (similar to C++):

```
>> A *= 2
>> print A
[ [2,4],
  [6,8] ]
```

Reshaping

- reshaping an array (usually **no copy**):

$$\vec{x} = \left(\ 0, 1, \ldots, 9\ \right)$$

$$M = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 & 9 \end{pmatrix}$$

$$N = \begin{pmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \\ 6 & 7 \\ 8 & 9 \end{pmatrix}$$

```
>> x = np.arange (10)

>> M = x.reshape(2,5)

>> N = x.reshape(5,-1)

>> O = x.reshape(3,-1)
==> returns an error
```

“-1” means whatever needed

modifying x, M or N modifies all objects

working with copies:

$$O = \begin{pmatrix} 40 & 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 & 9 \end{pmatrix}$$

```
>> x = np.arange (10)

>> O = x.copy().reshape(2,5)

>> O[0,0] = 40
```

x is unchanged

Resizing

- resize an array with resize():

```
>> X = np.arange(10)
>> X.resize(3,3)
```

```
>> Y = np.arange(7)
>> Y.resize(3,3)
```

$$X = \begin{pmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{pmatrix}$$

$$Y = \begin{pmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 0 & 0 \end{pmatrix}$$

- references may impede resizing:

```
>> x = np.arange (10)
>> M = x.reshape(2,5)

>> M.resize(3,3)
==> error: does not own data

>> x.resize(3,3)
==> error: referenced by other array

>> del M; x.resize(3,3)
==> this works
```

$$\vec{x} = \left(\; 0, 1, \ldots, 9 \; \right)$$

$$M = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 & 9 \end{pmatrix}$$

alternatively use a copy:

```
>> y = x.copy().resize(3,3)
```

```
>> y = np.resize(x, (3,3) )
```

Fancy Indexing

- Indexing with arrays of indices

$$\vec{x} = \begin{pmatrix} 0 \\ 2 \\ 4 \\ 6 \\ 8 \end{pmatrix}$$

```
>> x = np.arange ( 0, 10, 2)

>> idx = np.array ( [0,4,4,2])

>> y = x [ idx ]

>> print y
[0, 8, 8, 4 ]
```

$$\vec{y} = \begin{pmatrix} 0 \\ 8 \\ 8 \\ 4 \end{pmatrix}$$

- also works with N-dimensional index arrays

```
>> x = np.arange ( 0, 10, 2)

>> idx = np.array ( [[0,4] , [4,2] ])

>> M = x [ idx ]

>> print M
[ [0, 8],
  [8, 4 ] ]
```

$$M = \begin{pmatrix} 0 & 8 \\ 8 & 4 \end{pmatrix}$$

Fancy Indexing II

- also can present higher dimensional index

$$A = \begin{pmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{pmatrix}$$

```
>> A = np.arange (9).reshape(3,3)
>> i = np.array ( [0, 2] )  # defines rows
>> j = np.array ( [1,2] )   # defines columns
>> x = A [ i, j ]
>> B = A [ i,: ]
```

$$\vec{x} = \begin{pmatrix} 1 \\ 8 \end{pmatrix}$$

$$B = \begin{pmatrix} 0 & 1 & 2 \\ 6 & 7 & 8 \end{pmatrix}$$

$$i = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

$$j = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$$

```
>> i = np.eye (2, dtype = int )
>> j = np.ones( (2,2), dtype = int)
>> C = A[ i, j]
```

$$C = \begin{pmatrix} 4 & 1 \\ 1 & 4 \end{pmatrix}$$

Fancy Indexing III

- Boolean indexing, explicitly choose the elements

```
>> A = np.arange(1,5).reshape(2,2)

>> idx = np.array ( [ [True, False], [True, True] ])
>> x = A [ idx ]

>> idx = [ [True, False], [True, True] ]  # this is a list
>> y = A[idx]                             # be careful!

>> idx = A<3
>> z = A[idx]
```

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

$$\vec{x} = \begin{pmatrix} 1 \\ 3 \\ 4 \end{pmatrix} \qquad \vec{y} = \begin{pmatrix} 4 \\ 2 \end{pmatrix} \qquad \vec{z} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

Universal Functions

- Numpy provides a useful set of mathematical functons. They are called „universal functions“ and work **elementwise**.

```
>> x = np.arange(5)

>> np.sqrt (x)
[0., 1., 1.41421, 1.730225]
```

```
>> x = np.arange(5)

>> np.exp (x)
[1., 2.71828, 7.3891, 20.0855]
```

- Fast and very usefull for data processing,
  - eg. consider you have a list with data

```
>> absdata = [abs(i) for i in data  ] #have to iterate over list

>> absdata = np.abs(data) #can direclty manipulate array
```

arccos, arctan, ceil, conjugate, cos, exp, fabs, floor, fmod, log, log10, sin, sinh, sqrt, …

Is everything installed correctly?

- Open iPython notebook and run the following commands:

```
>>> import numpy as np
>>> import scipy as sc
>>> import pylab as pl
```

- If nothing happens, everything is fine :)

Example: Efficiency of NumPy

Your task:

Go through the numbers from one to one million and count the numbers that are dividable by 7.

(Yes, please really test the numbers ☺)

(a) Implement a function using a for loop to do the task.
(b) Use list comprehension to do the task.
(c) Use numpy and avoid loops at all.

To compare the runtimes of each version, use the “magic function” **`%timeit`.**

SciPy Package

- Based on the *numpy* package *scipy* provides advanced methods for science and engineering:
  - Constants (scipy.constants)
  - Fourier transforms (scipy.fftpack)
  - Integration and ODEs (scipy.integrate)
  - Interpolation (scipy.interpolate)
  - Linear algebra (scipy.linalg)
  - Orthogonal distance regression (scipy.odr)
  - Optimization and root finding (scipy.optimize)
  - Signal processing (scipy.signal)
  - Special functions (scipy.special)
  - Statistical functions (scipy.stats)
  - C/C++ integration (scipy.weave)
  - And more …

- Check: http://docs.scipy.org/doc/

Matplotlib Package

- „make easy things easy and hard things possible:“
  - create simple plots with just a few commands
  - “emulate” MATLABs plotting capabilities

- matplotlib is conceptually divided into three parts
  - ***Pylab interface :*** MATLAB like plotting
  - ***Matplotlib API :*** abstract interface
  - ***Backends :*** managing the output

- available at (including many examples)

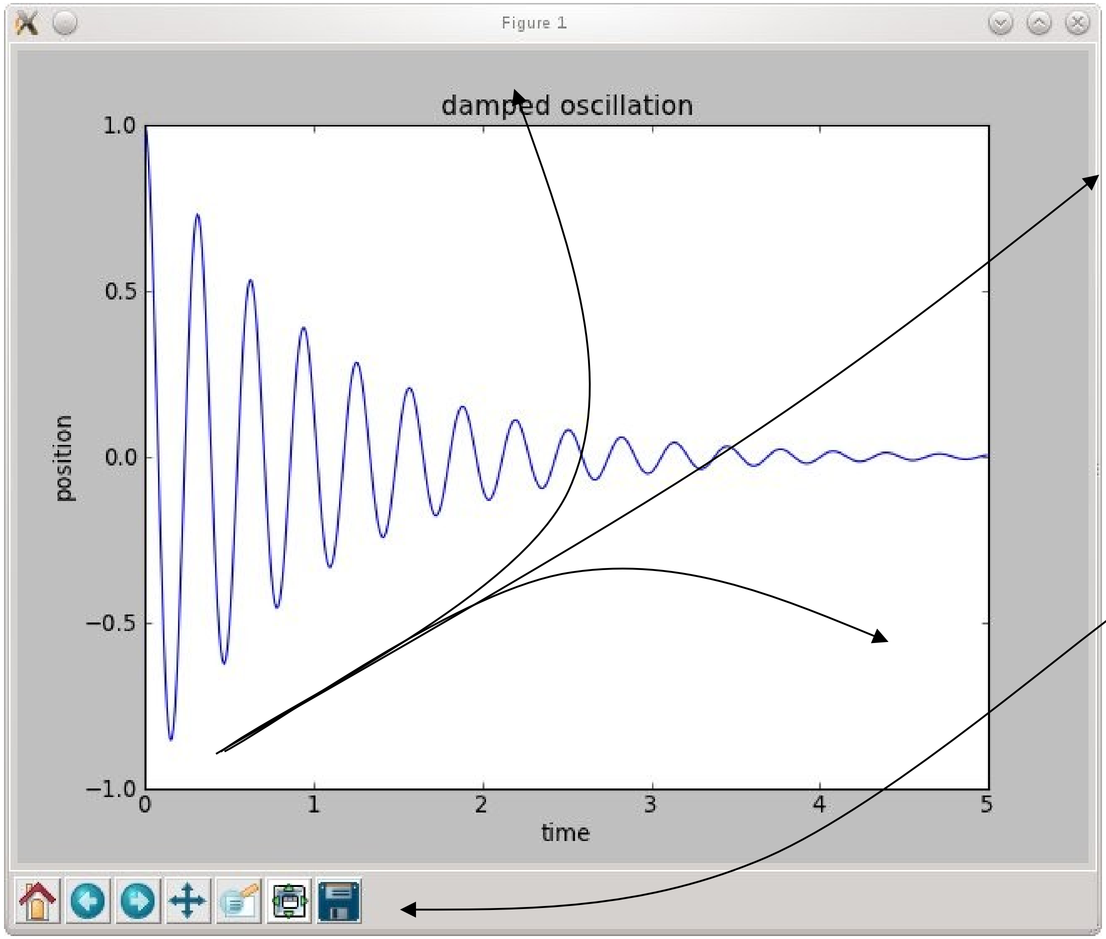
  http://matplotlib.sourceforge.net/

Basic 2D Plotting

- MATLAB like example:

```
import numpy as np                  # import numpy
import pylab as pl                  # import pylab interface



times = np.arange ( 0, 5, 0.01 )             # define x-vector
fun  = lambda x : np.cos (20 *x) * np.exp (- pl.absolute(x) )
                              # define some function fun (x)

pl.plot ( times, fun(times) )     # plot fun (t) vs. t
pl.xlabel ('time' )                 # creating x-label
pl.ylabel ('position')              # creating y-label

pl.title ( 'damped oscillation')             # setting the title
pl.show()                                    # show the plot
```

Basic 2D Plotting



line plot represents data

title and labels

toolbar for zooming, saving/exporting etc.

appearance depend on backend

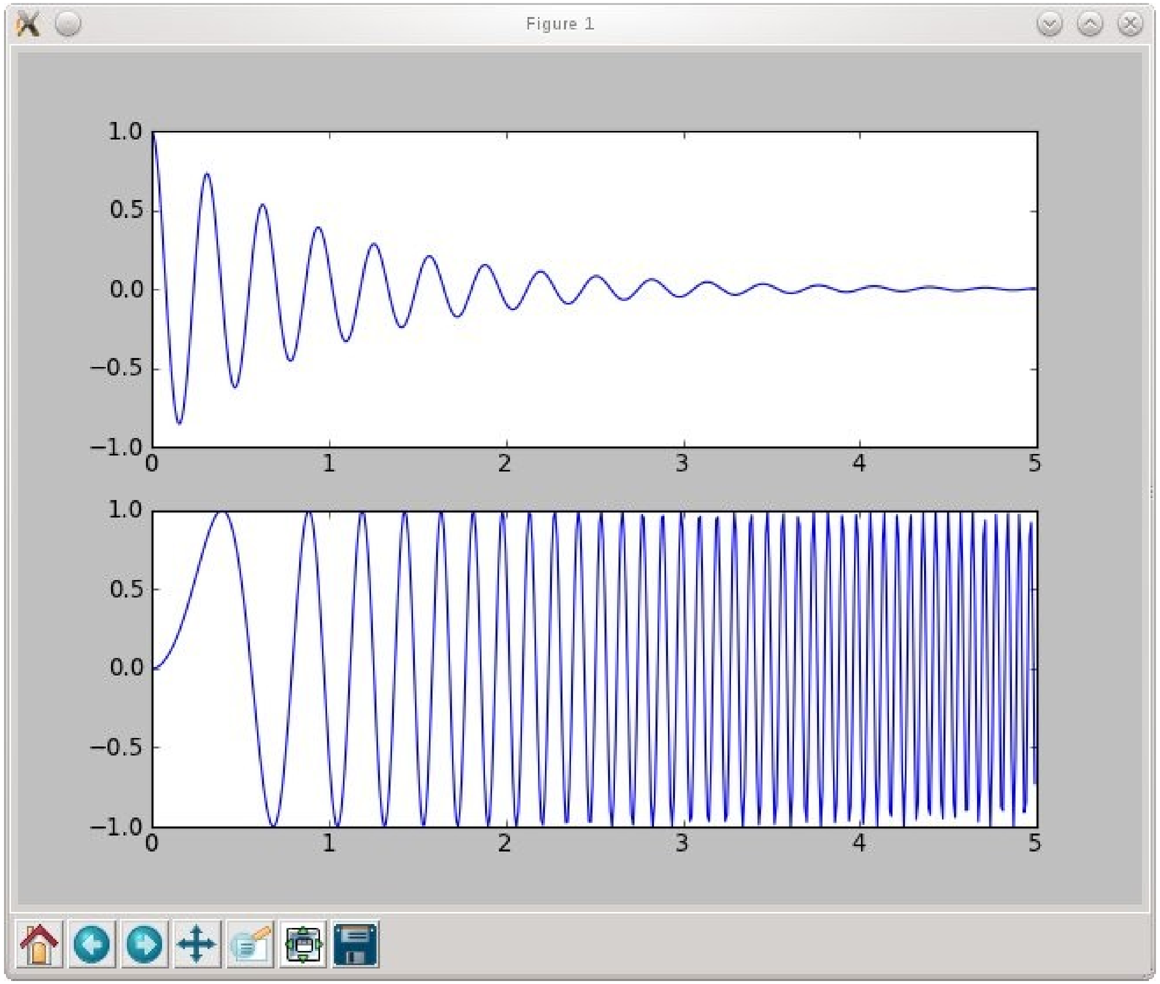
Interactive Plotting

- For Python scripts you use **pylab.show()** to show the plot after setting the parameters.
- Working in the Python shell you usually want to know how things look like immediately. Therefor you can use **pylab.ion()** to start the interactive mode.
- With **pylab.draw()** you can update the figure after changing it.

Subplots

```
import numpy as np                # import numpy
import pylab as pl                # import pylab interface

times = np.arange ( 0, 5, 0.01 )           # define x-vector

fun   = lambda x : np.cos (20 *x) * np.exp (- pl.absolute(x) )
fun2  = lambda x : np.sin (10 *x**2)     # define two functions



pl.subplot (2,1,1)                # choose a subplot ( rows, colums, idx)
pl.plot ( times, fun(times) )     # plot fun(t)

pl.subplot (2,1,2)                # choose a subplot ( rows, colums, idx)
pl.plot ( times, fun2(times) )    # plot fun2(t)

pl.show()
```

Subplots



subplot (2,1,1) :

- 2 columns, 1 row
- choose first subplot

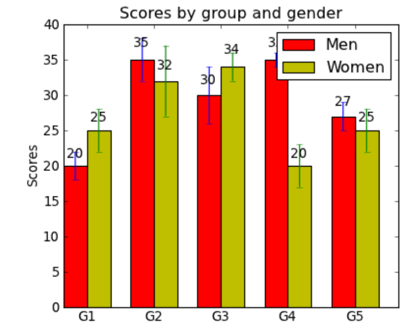
! Indexing starts with 1

subplot (2,1,2) :

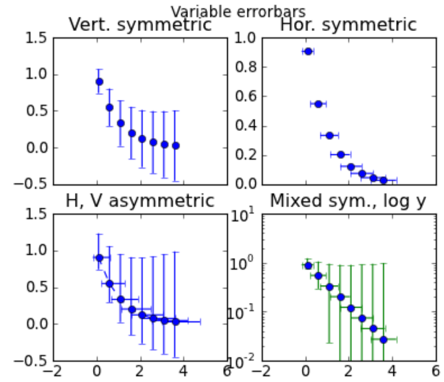
- 2 columns, 1 row
- choose second subplot

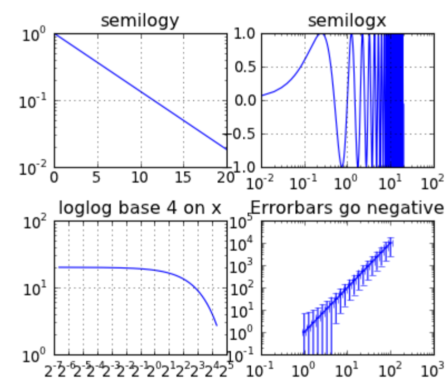
Other basic plotting commands

- pl.bar () # box plot



- pl.errorbar() # plot with errorbars
- pl.loglog() # logarithmically scaled axis
- pl.semilogx () # x-axis logarithmically scaled
- pl.semilogy () # y-axis logarithmically scaled

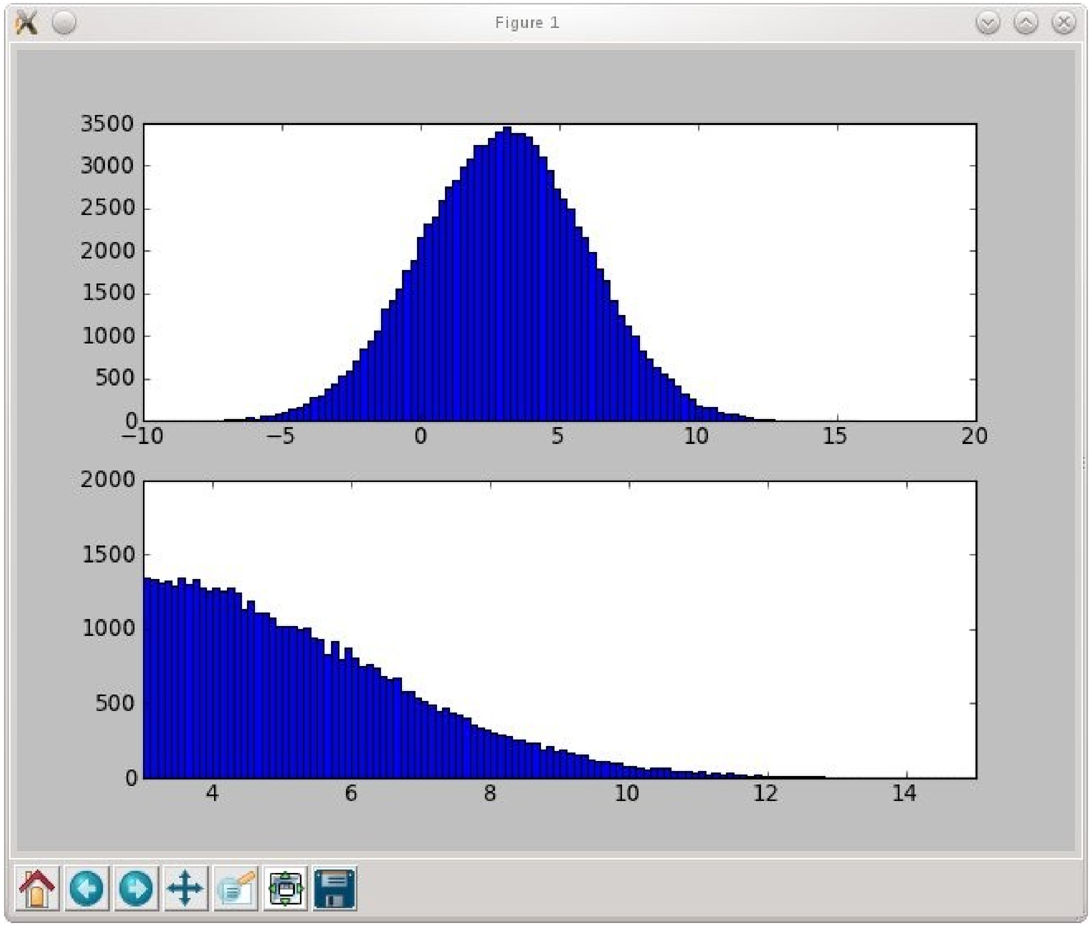




Histograms

```
import numpy as np                # import numpy
import pylab as pl                # import pylab interface

data = 3. + 3. * np.random.randn (100000)
           # generate normally distributed randonnumbers

pl.subplot (2,1,1)
pl.hist (data, 100)    # make histogram with 100 bins

pl.subplot (2,1,2)
pl.hist ( data, bins = np.arange(3, 25, 0.1) )
                      # make histogram with given bins

pl.axis ( (3, 15,0,2000 ))        # specify axis (x1,x2,y1,y2)

pl.show()
```

Histograms



(automatic) histogram with 100 bins

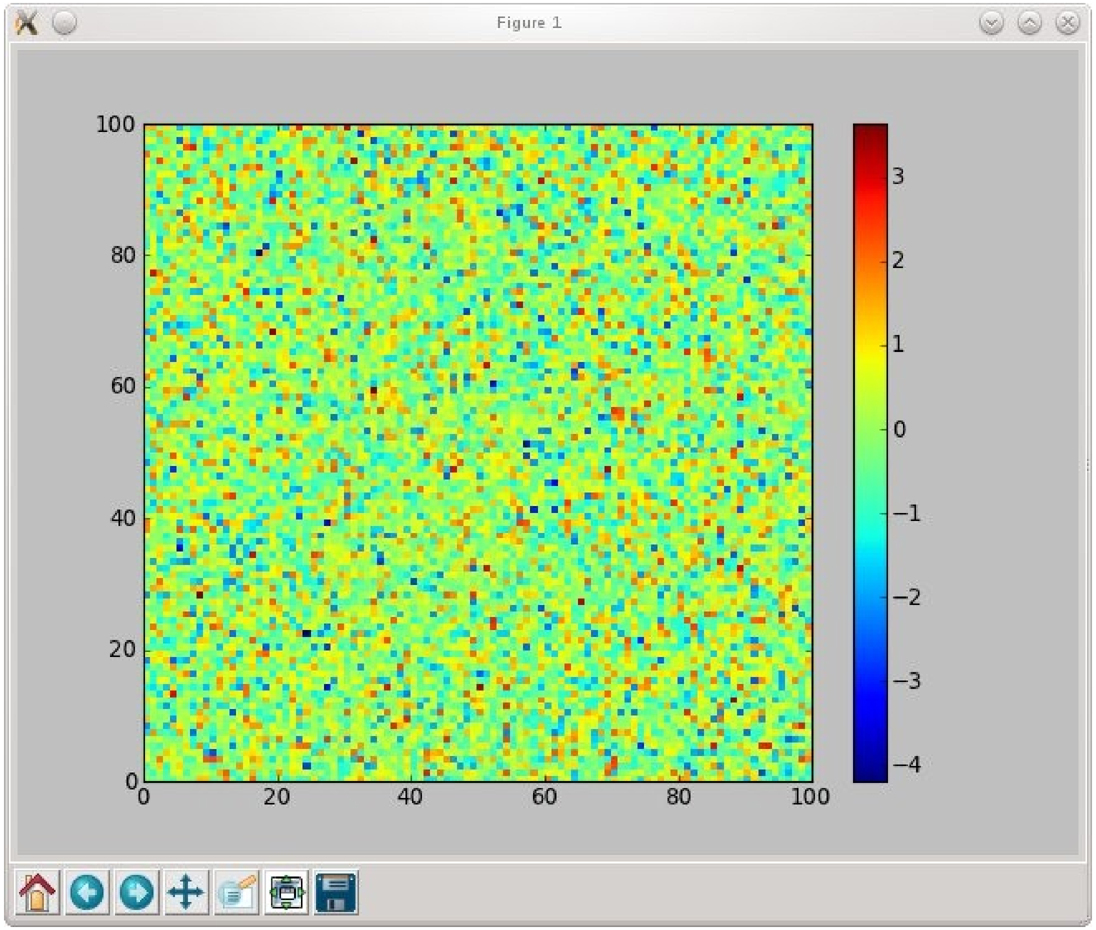
histogram for data between 3. and 25. with binsize 0.1

axis set to (3,15,0,2000)

Basic Matrix Plotting

```
import numpy as np              # import numpy
import pylab as pl              # import pylab interface

data = np.random.randn (100,100)
            # generate random data

pl.pcolor (data)       # plot data
pl.colorbar()          # show a colorbar

pl.show()
```

Basic Matrix Plotting



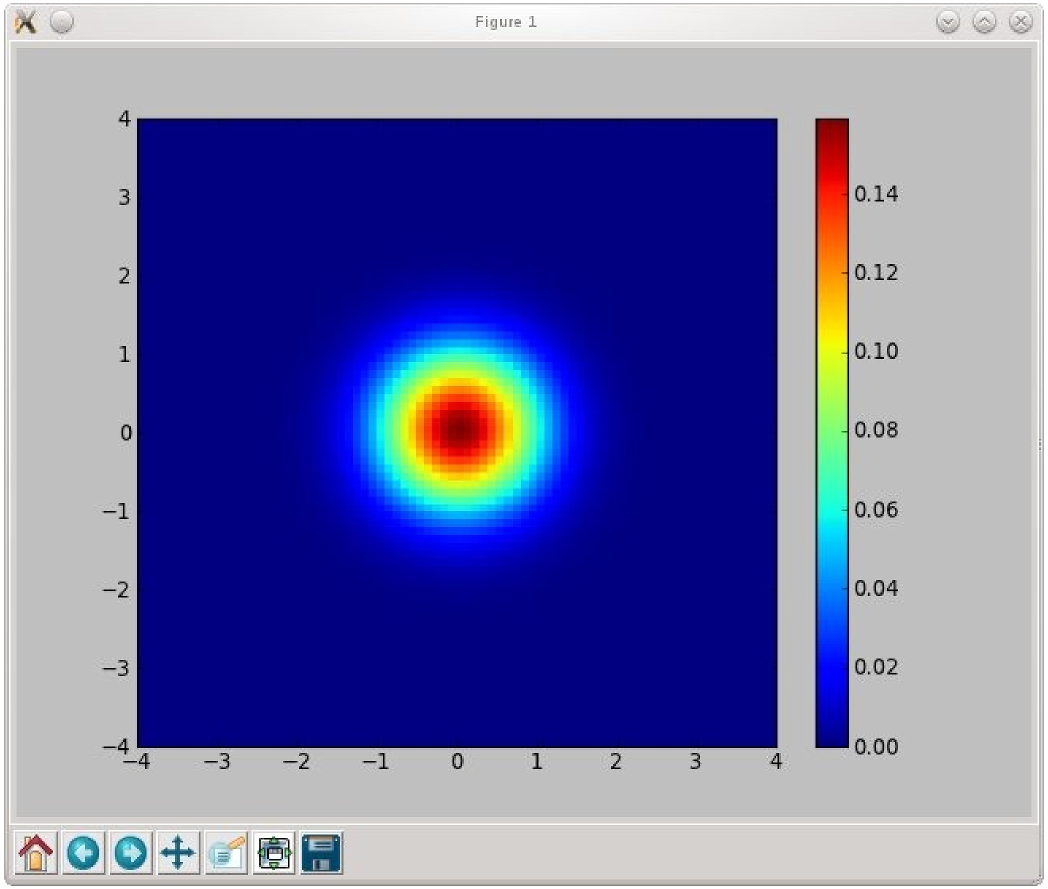
dimensions of matrix used as coordinates

entries are translated to a color code

2D - Functions

```
import numpy as np                  # import numpy
import pylab as pl                  # import pylab interface

x = np.arange ( -4, 4.01, 0.1)  # x-values
y = np.arange ( -4, 4.01, 0.1)  # y-values

X,Y = np.meshgrid(x,y)          # Create a meshgrid !
Z = np.zeros ( X.shape )        # Matrix for function values
Z = 1./2/np.pi * np.exp ( - (X**2 + Y**2) )

pl.pcolor(X,Y,Z)                # plot function
pl.axis ( (-4,4,-4,4) )         # set axis
pl.colorbar()                   # show colorbar

pl.show()
```

2D - Functions



$$X = \begin{pmatrix} x_1 & x_1 & x_1 & \cdots \\ x_2 & x_2 & x_2 & \cdots \\ \cdots & \cdots & \cdots & \end{pmatrix}$$

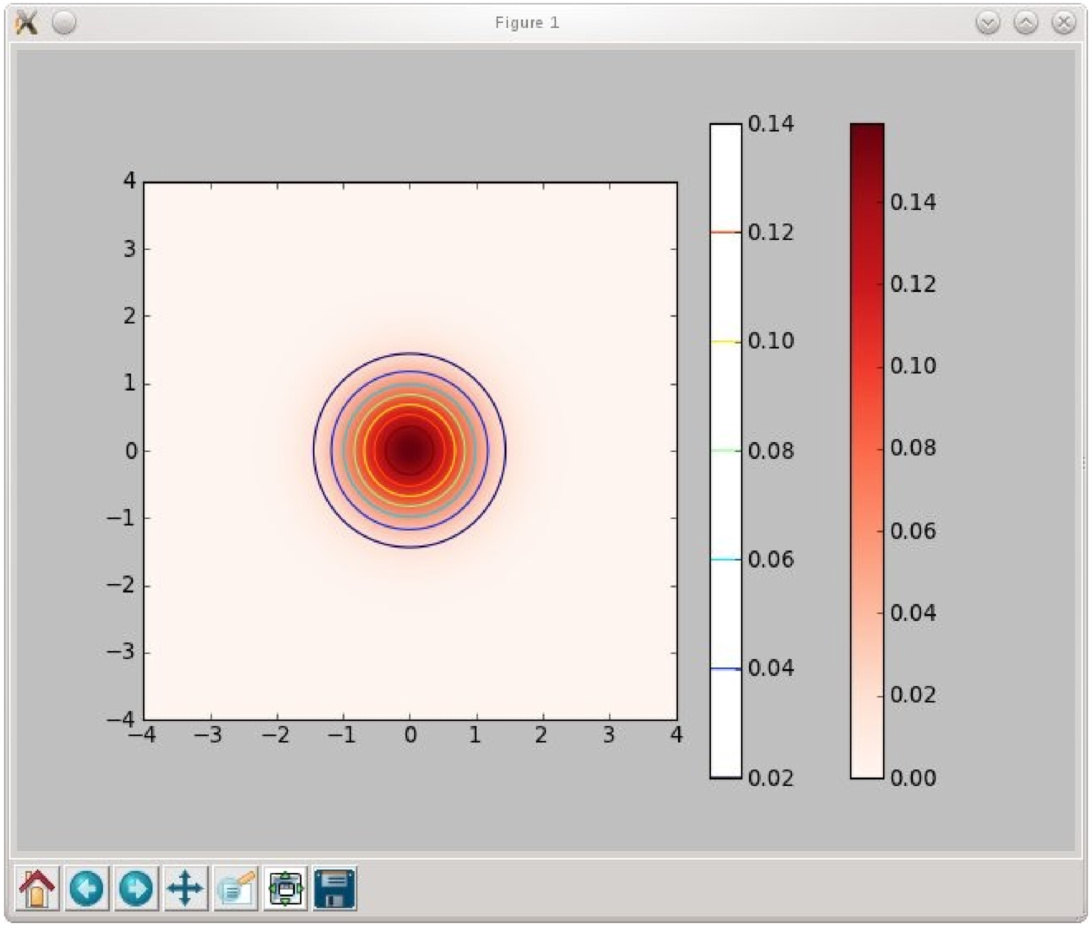
$$Y = \begin{pmatrix} y_1 & y_2 & y_3 & \cdots \\ y_1 & y_2 & y_3 & \cdots \\ \cdots & \cdots & \cdots & \end{pmatrix}$$

$$Z = \begin{pmatrix} f(x_1, y_1) & f(x_1, y_2) & \cdots \\ f(x_2, y_1) & f(x_2, y_2) & \cdots \\ \cdots & \cdots & \cdots \end{pmatrix}$$

Contour Plots

```
import numpy as np              # import numpy
import pylab as pl              # import pylab interface

x = np.arange ( -4, 4.01, 0.01)                  # x-values
y = np.arange ( -4, 4.01, 0.01)                  # y-values

X,Y = np.meshgrid(x,y)          # Create a meshgrid !
Z = np.zeros ( X.shape )        # Matrix for function values
Z = 1./2/np.pi * np.exp ( - (X**2 + Y**2) )

pl.imshow(Z, extent=(-4,4,-4,4), cmap = pl.cm.Reds  )
pl.colorbar()

pl.contour(X,Y,Z) # Creates a contour plot
pl.colorbar()

pl.show()
```

Contour Plots

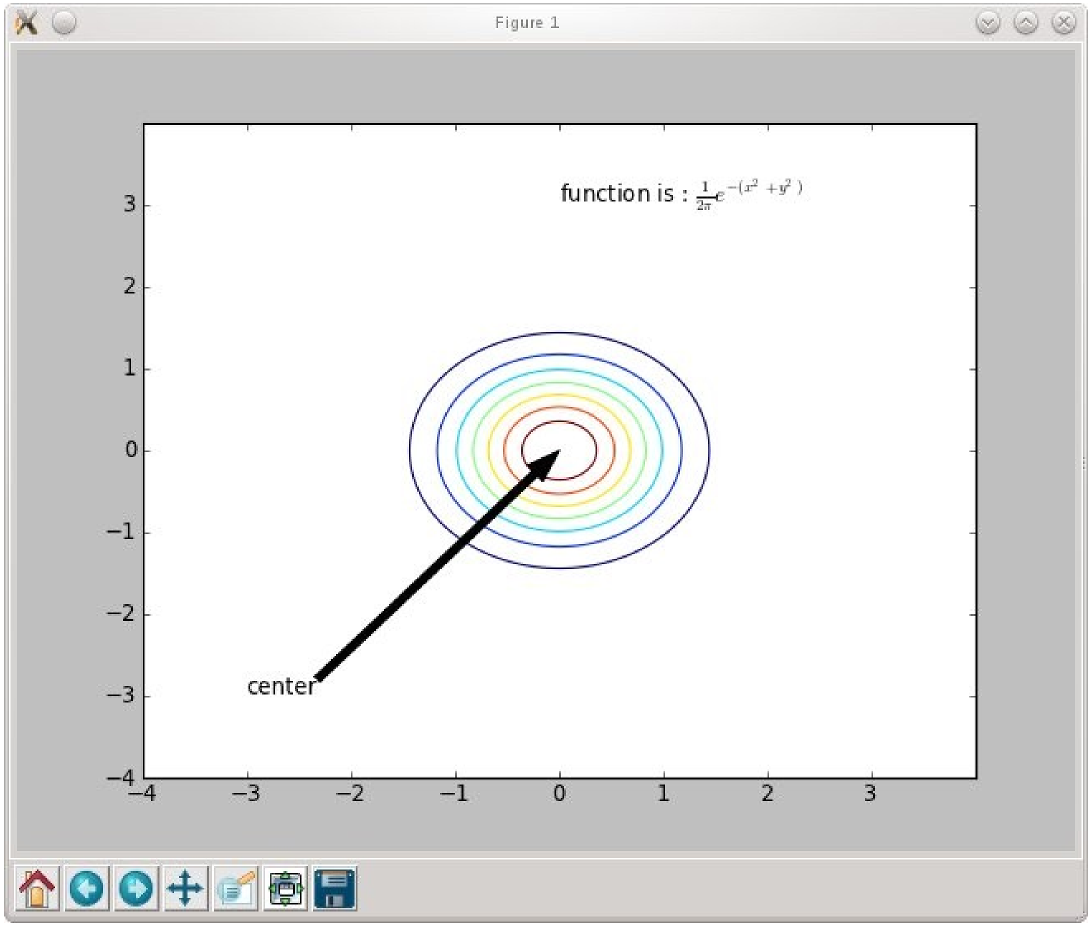


Working with text

- Include text with text() or annotate()
  - you can use TeX (translated by matplotlib itself)
  - you can use real LaTeX (matplotlib.rc (‘text’, usetex=‘true’))

```
import numpy as np                 # import numpy
import pylab as pl                 # import pylab interface
…
pl.contour(X,Y,Z)                  # make a contour plot

pl.text (0,3, 'function is : '+ r'$\frac{1}{2\pi} e^{- (x^2 + y^2) } $' )
              # (x,y, text);   r‘….‘ indicates rawtext
pl.annotate ( 'center', xy = (0,0), xytext = (-3,-3), arrowprops =
{'facecolor':'black'} )
              # xy <= where the arrow ends
              # xytext <= position of the text

pl.show()
```

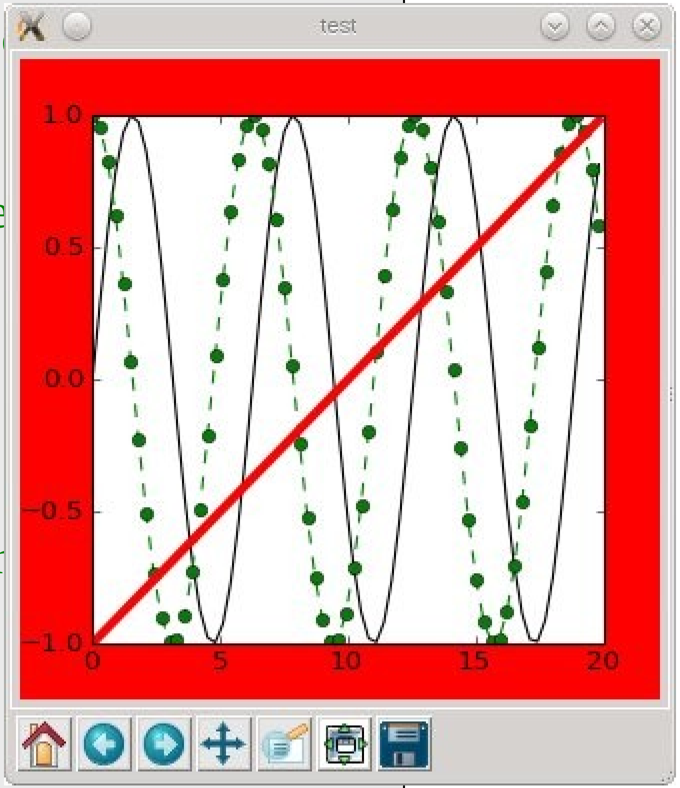
Working with text



Formatting Figures (Keywords)

- Properties of plots can be set by keywords:

```
import numpy as np              # import numpy
import pylab as pl              # import pylab interfac

pl.figure( “test“ , figsize = (4,4), facecolor = 'r')
          # create figure with title test, 4x5 inches, re

x = np.arange ( 0, 20, 0.3 )    # x – values

# for basic properties: using formatstring
pl.plot (x, np.sin(x), 'k' )        # black line
pl.plot (x, np.cos(x), 'go--' )     # green dotted line with

# using keywords
pl.plot (x, x / 10. - 1, color = 'red', linewidth = 4)
pl.show()
```



For details: **help(command)** or **http://matplotlib.sourceforge.net/**

pylab.getp(), pylab.setp()

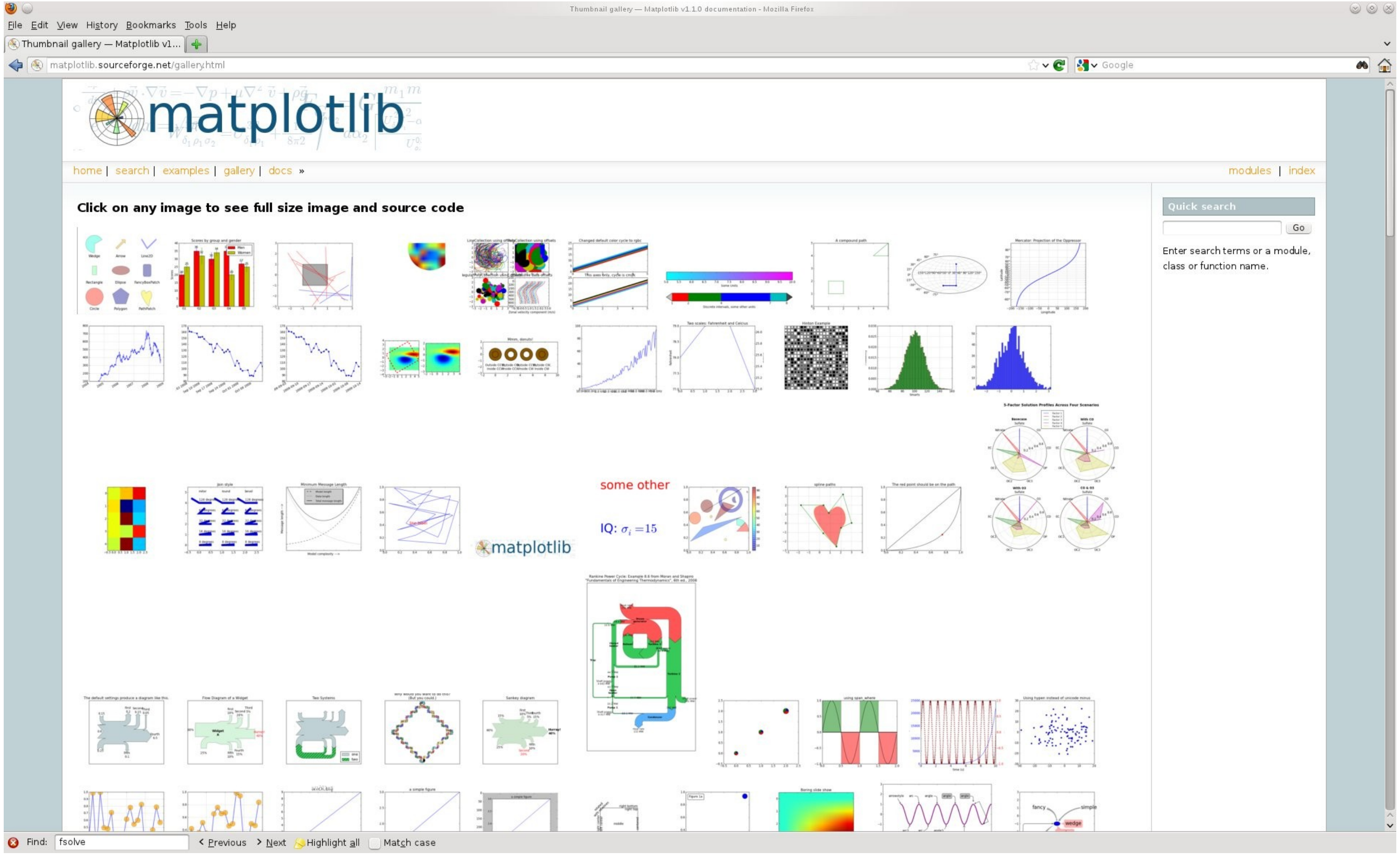
Once you have the objects, you can manipulate them via

- the methods provided by the class:
  - there are thousands of **object.get_xxx** and **object.set_xxx** methods

    use the <tab> or look at **http://matplotlib.sourceforge.net/api**

- **pylab.setp()** command:
  - **pylab.getp(obj)** returns all properties of the object
  - **pylab.getp(obj, property)** returns value of current property
  - **pylab.setp(obj, property)** returns possible values for property
  - **pylab.setp(obj, propety=value)**

The Gallery



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

For help take a look at the reference pages:

SciPy:

- http://docs.scipy.org/doc/scipy/reference/

NumPy:

- http://docs.scipy.org/doc/numpy/reference/

Matplotlib:

- http://matplotlib.org/contents.html

Have fun in the exercises!

One last tip:

For in-line plotting of figures, open your ipython notebook with:

```
ipython notebook --pylab=inline
```