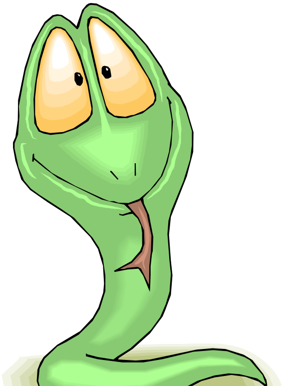
Introduction to NumPy, SciPy and Matplotlib

Scientific Computation With Python



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) )
```

! the shape has to be specified !

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

- 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

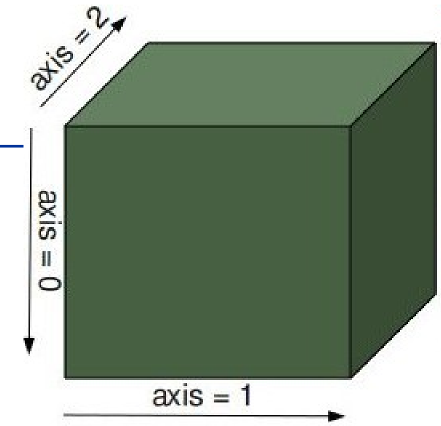
$$M = \begin{pmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{pmatrix}$$

```
>> 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

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
[ [0,1],
  [2,3] ]
```

```
>> 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] ]
```

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

- 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] ]
>> 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,

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

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

Finally I/O functions

- read & save text files

```
>> np.savetxt ( filename, variable) #format can be specified

>> data = np.loadtxt ( filename ) #adds an .npy to filename
```

- file format:
  - csv / tsv: comma/tab separated values

```
#comments
1.2    3.4    5.6    7.8    …
0.0    1.1    2.2    3.3    …
…      …      …      …      ...
```

- reading matlab files

```
>> import scipy.io as io

>> matdata = io.loadmat (filename.mat) #returns a dictonary

>> print matdata[ ‘data’ ] #if the matlabfile contains data struct
```

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: Basic 2D Plotting

- MATLAB like example:

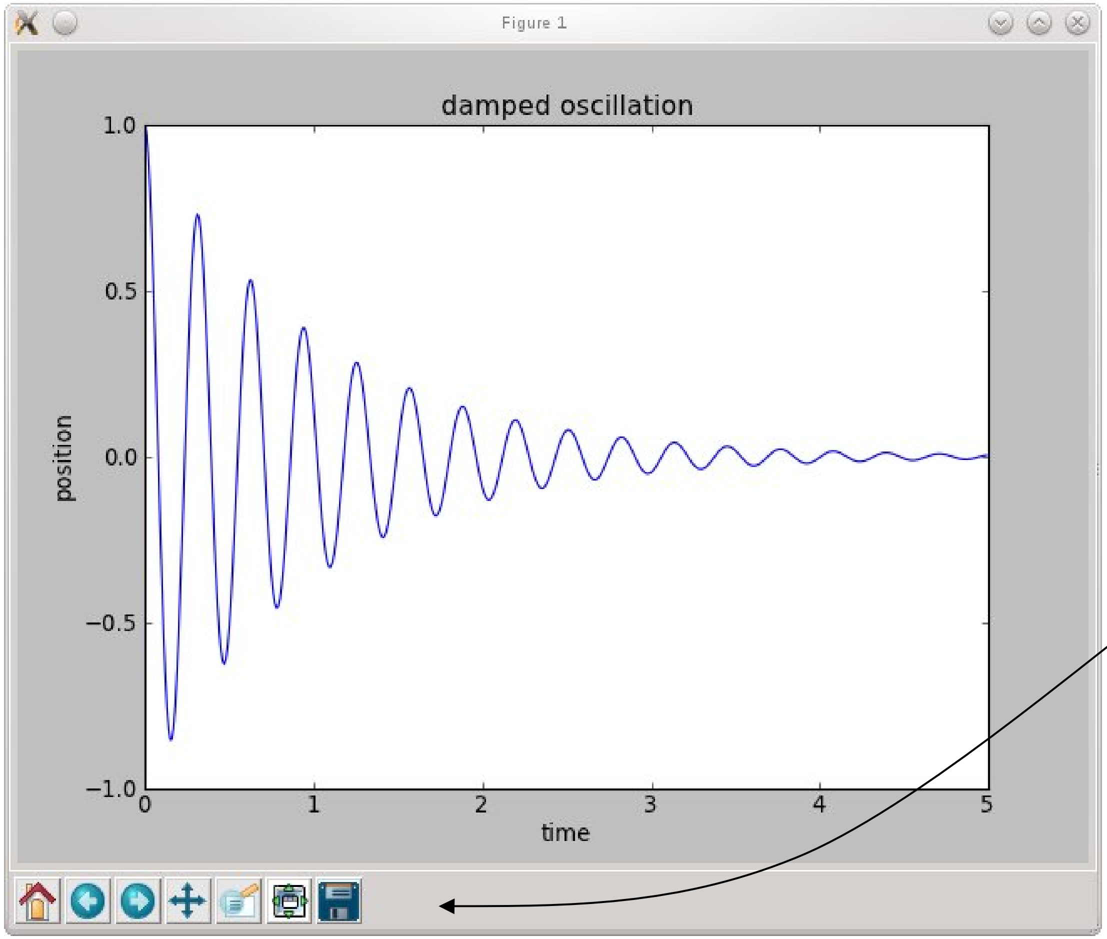
```
from pylab import *                # import pylab interface



times = arange ( 0, 5, 0.01 )    # define x-vector
def fun(x) :
        return  cos (20 *x) * exp (- abs(x) )
                          # define some function fun (x)

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

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

Matplotlib: Basic 2D Plotting

- More “pythonic” style:

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



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

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

plt.title ( 'damped oscillation')             # setting the title
plt.show()                        # show the plot, not necessary in notebook
```

Basic 2D Plotting



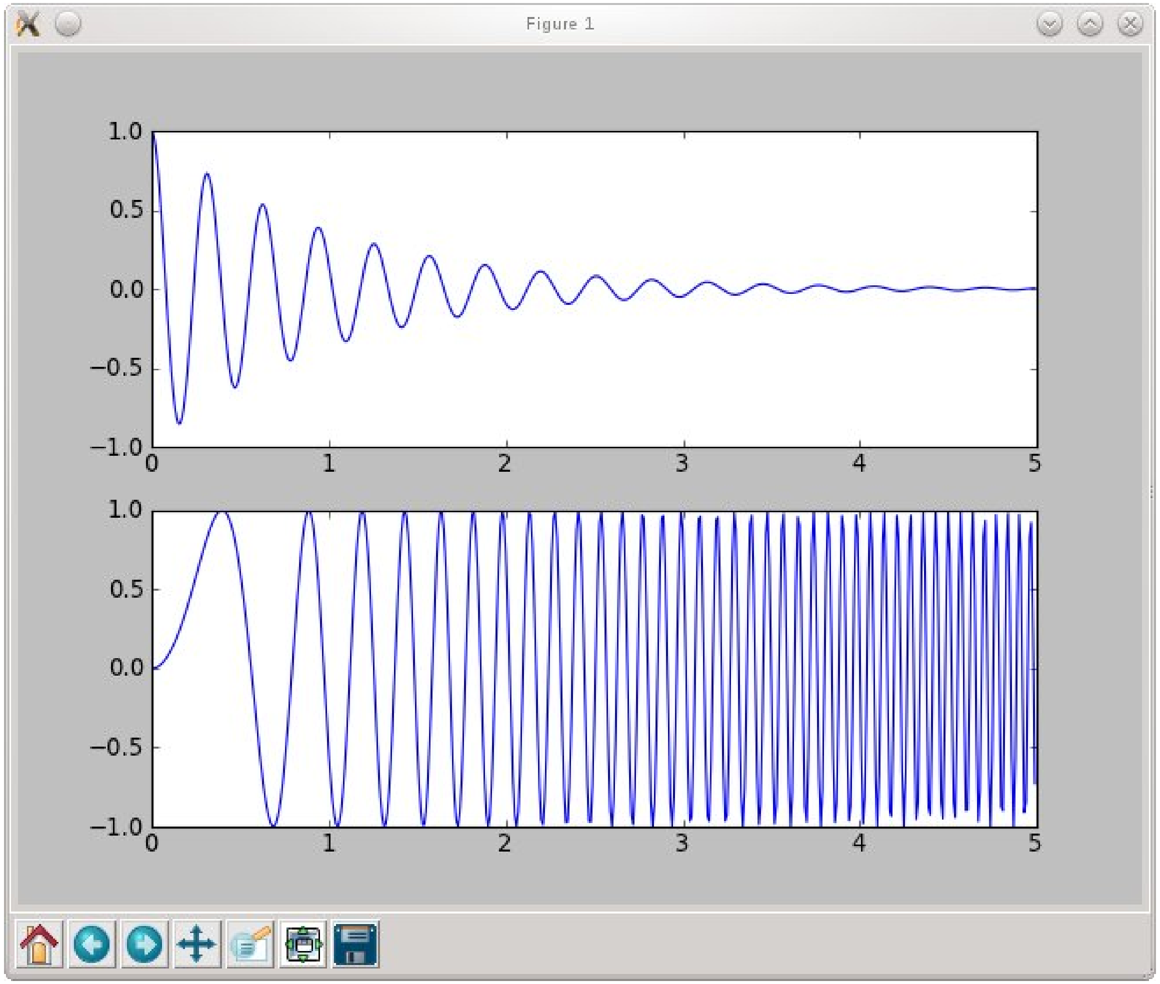
toolbar for zooming, saving/exporting etc.

appearance depends on backend

Interactive Plotting

- In the Jupyter Notebook use **%matplotlib inline** to make plots appear as figures in the browser.
- For Python scripts you use plt**.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 plt**.ion()** to start the interactive mode.
  - With plt**.draw()** you can update the figure after changing it.

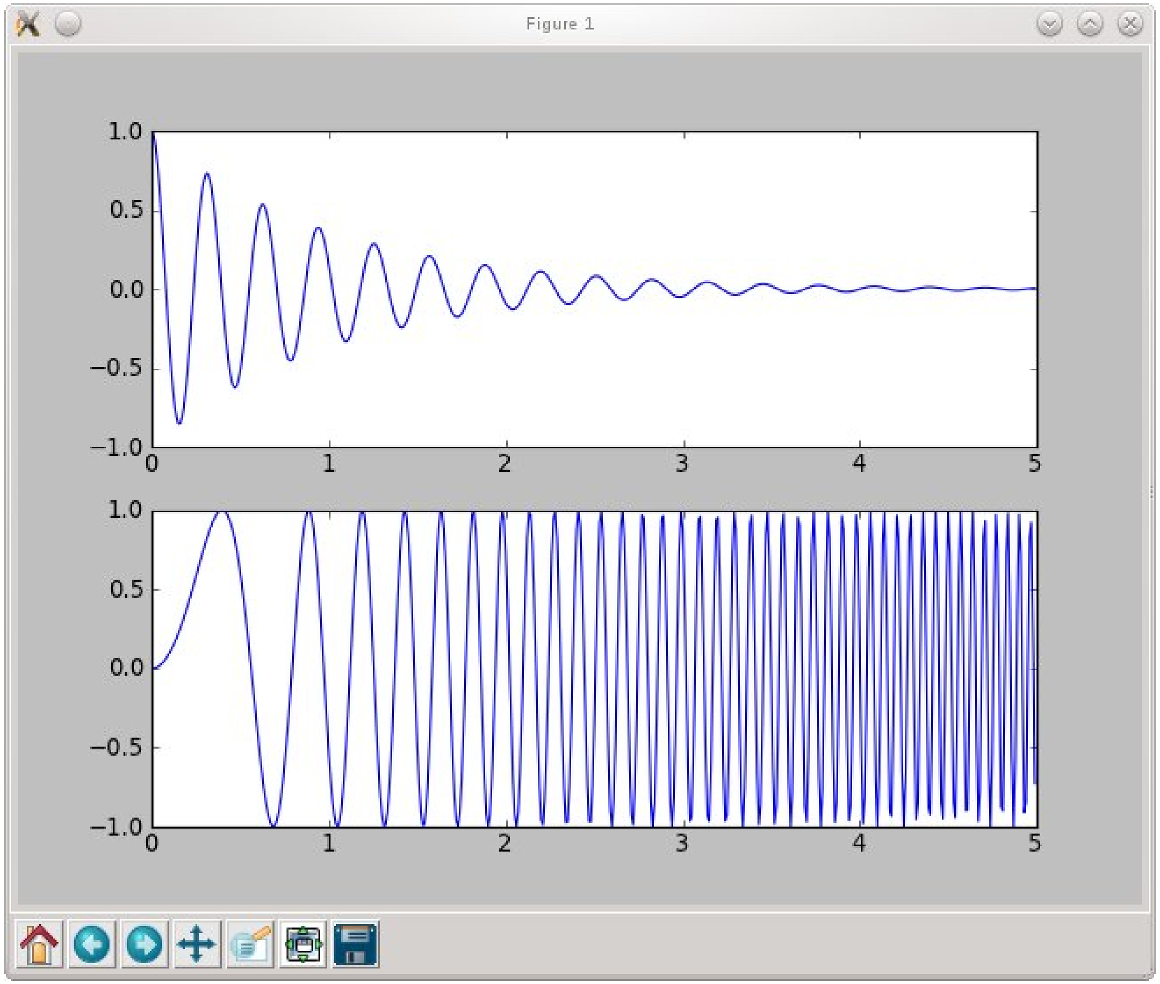
Subplots



Subplots

```
import numpy as np                         # import numpy
import matplotlib.pyplot as plt            # import pylab interface

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

def fun(x):
   return np.cos (20 *x) * np.exp (- plt.abs(x) )

def fun2(x):
  return np.sin (10 *x**2)        # define two functions

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

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

plt.show()
```

Subplots



subplot (2,1,1) :

- 2 columns, 1 row
- choose first subplot

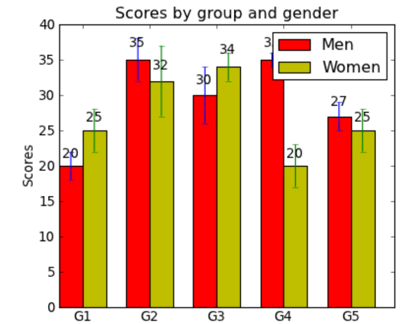
! Indexing starts with 1

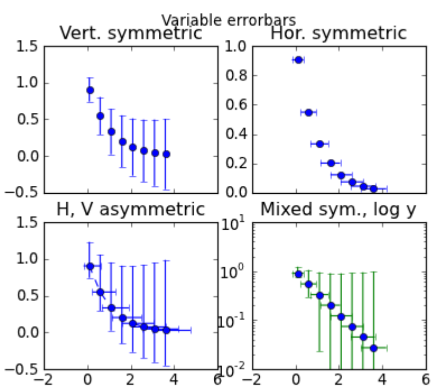
subplot (2,1,2) :

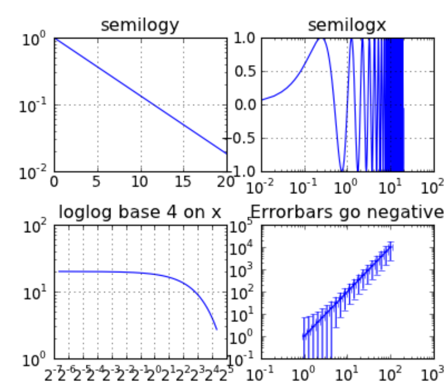
- 2 columns, 1 row
- choose second subplot

Other basic plotting commands

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



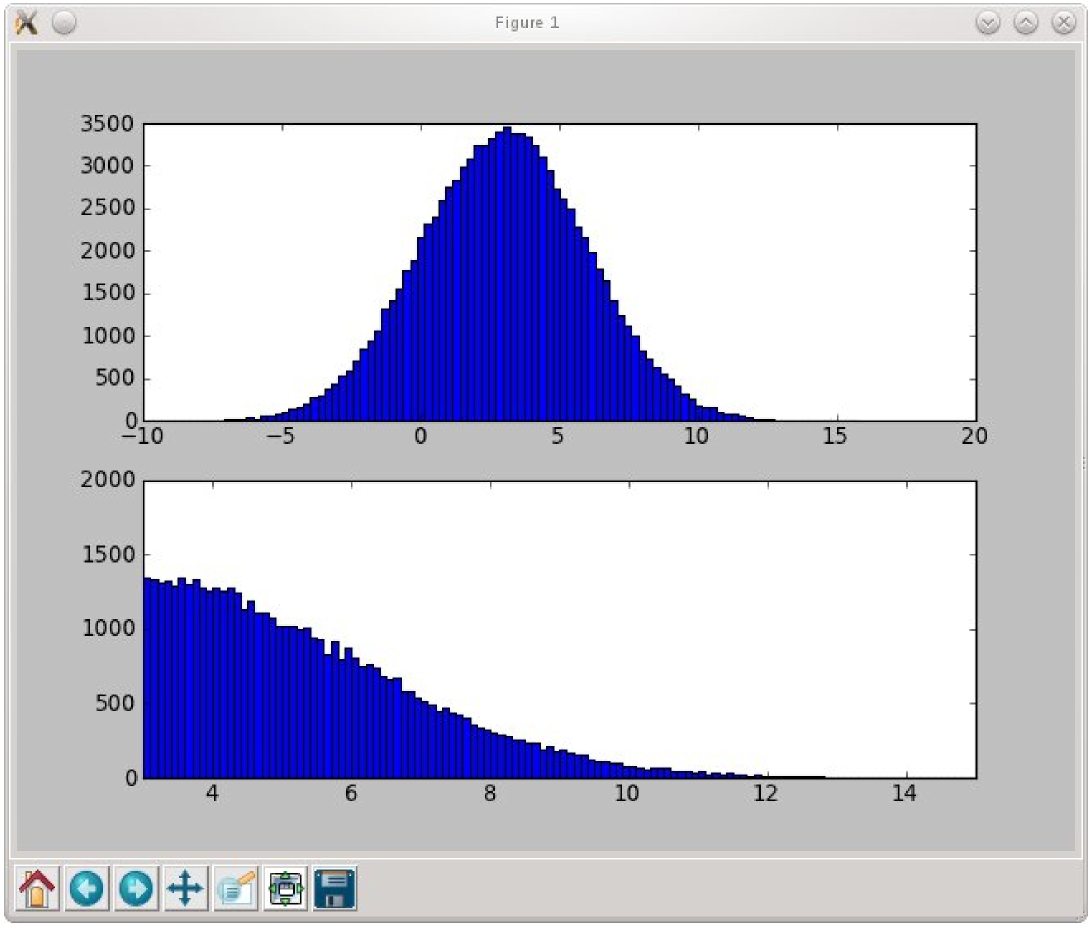




Histograms

```
import numpy as np                              # import numpy
import matplotlib.pyplot as plt                 # import pylab interface

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

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

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

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

plt.show()
```

Histograms



(automatic) histogram with 100 bins

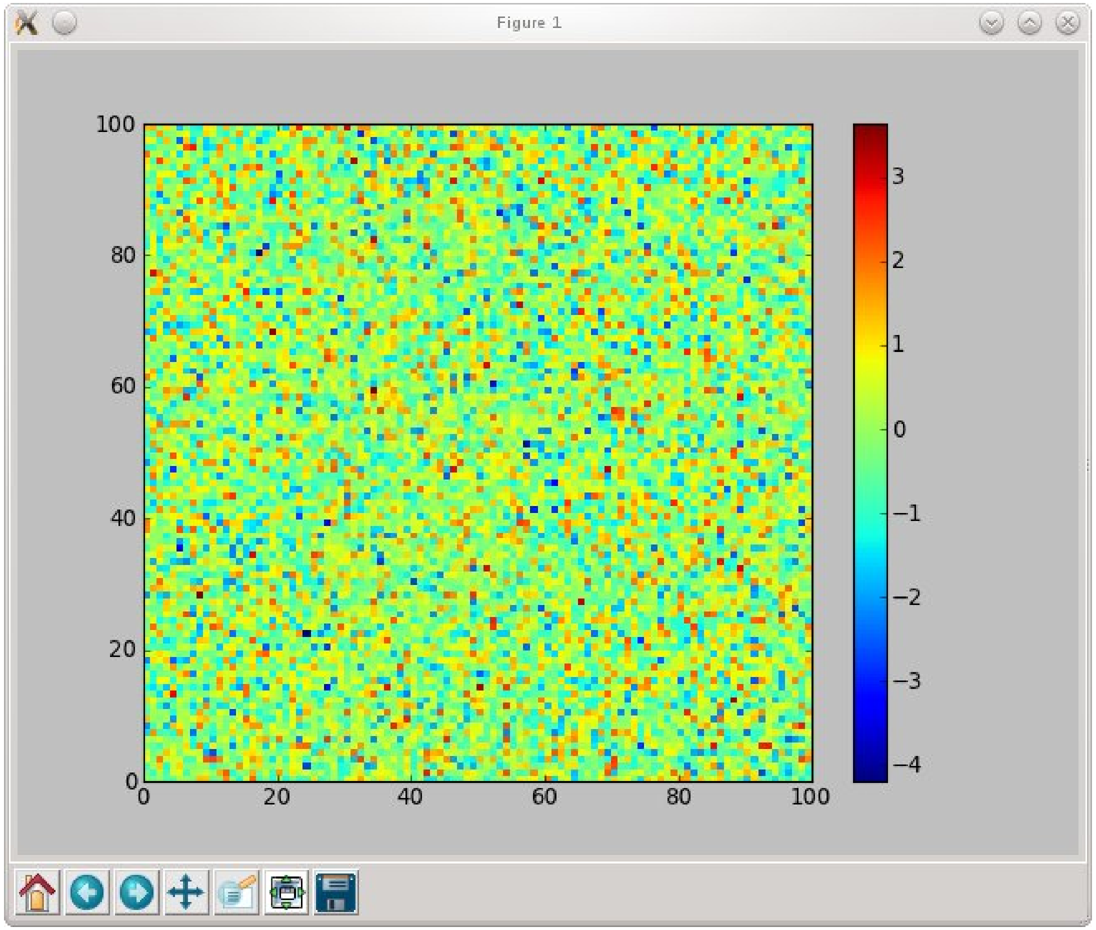
histogram for data between 3. and 25. with binsize 0.1

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

Basic Matrix Plotting

```
import numpy as np                    # import numpy
import matplotlib.pyplot as plt       # import pylab interface

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

plt.imshow (data)    # plot data
plt.colorbar()       # show a colorbar

plt.show()
```

Basic Matrix Plotting



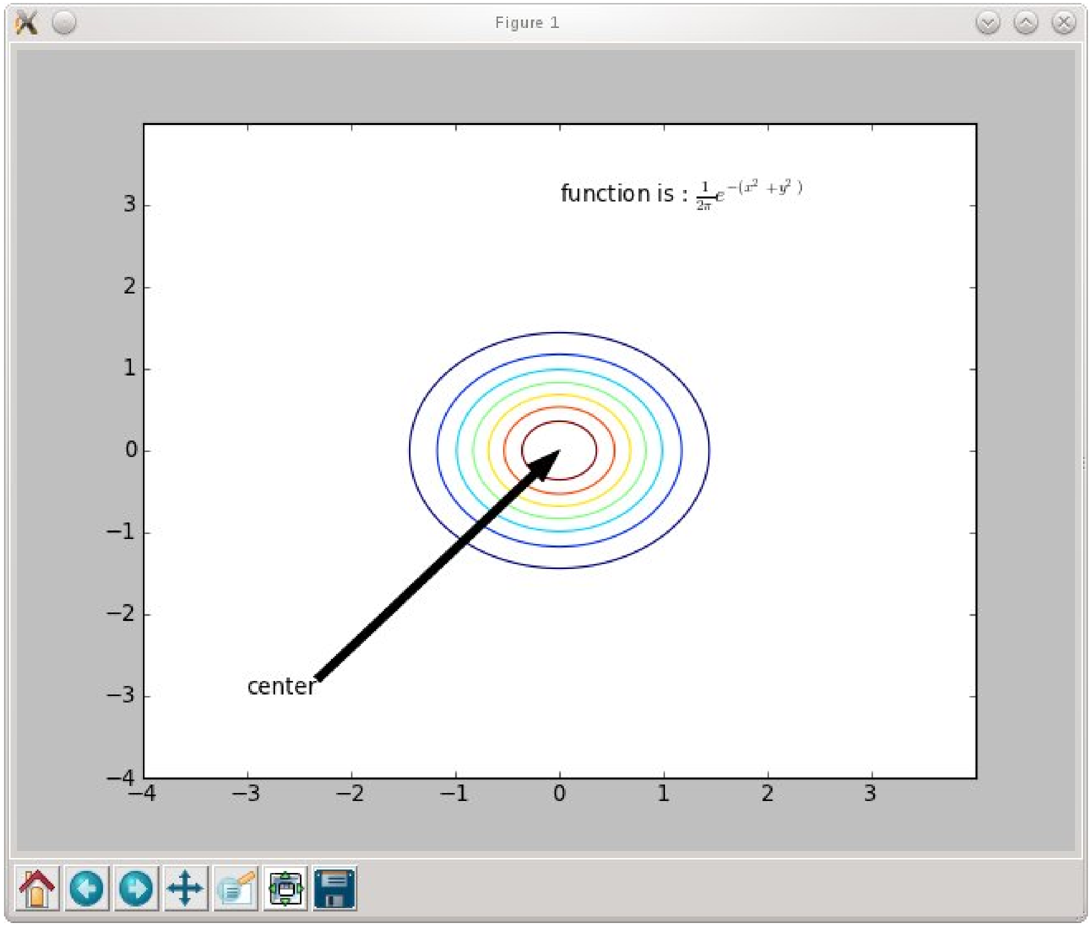
entries of matrix are translated to a color code

Working with text

- Include text with text() or annotate()
  - you can use LaTeX (enclosed in $...$)

```
import numpy as np                        # import numpy
import matplotlib.pyplot as plt           # import pylab interface

[...]

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

plt.show()
```

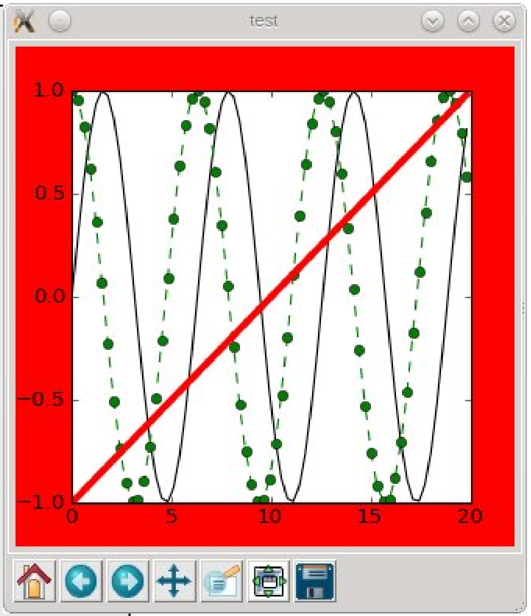
Working with text



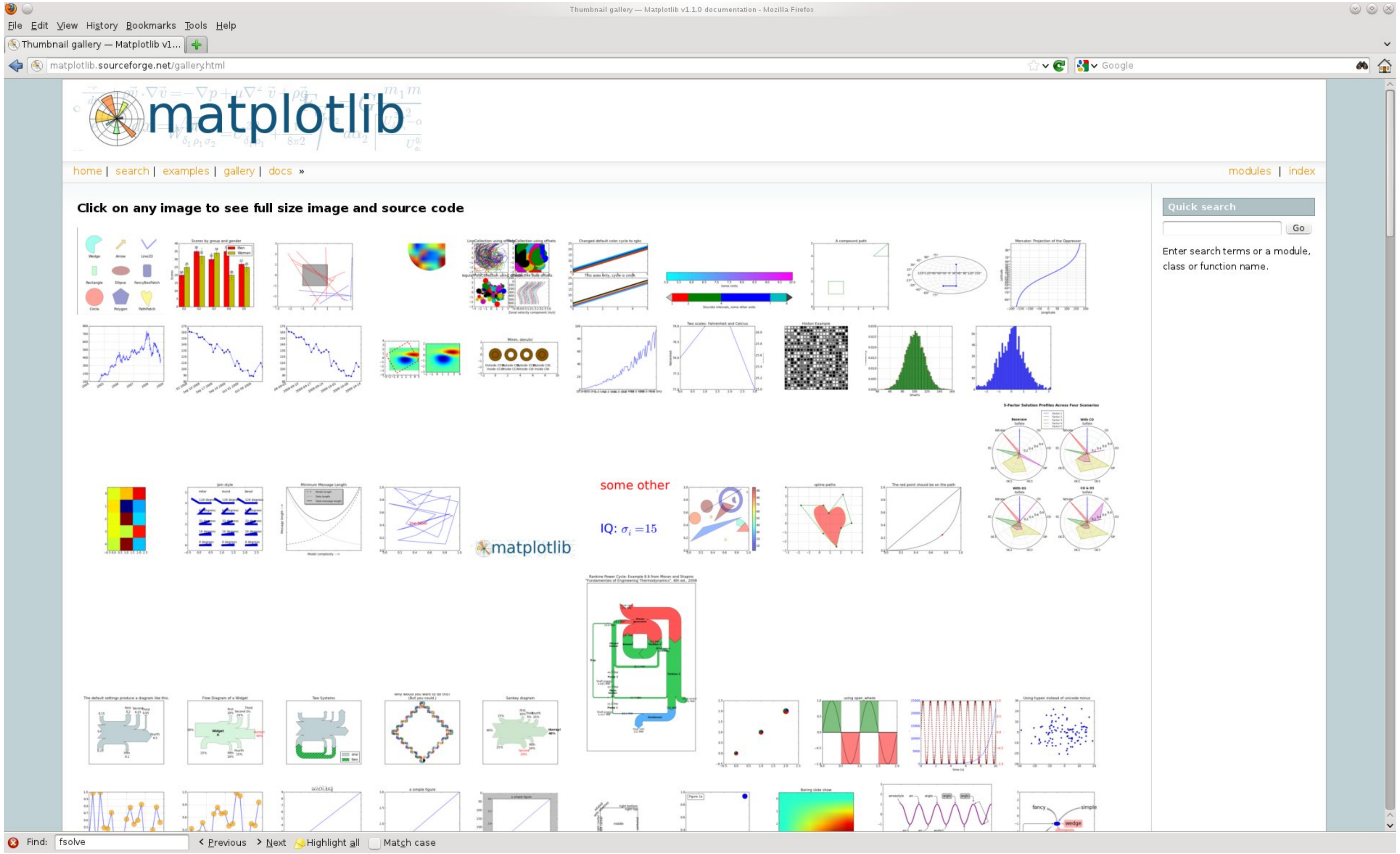
Formatting Figures (Keywords)

- Properties of plots can be set by keywords:

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

plt.figure( “test“ , figsize = (4,4), facecolor = 'r')
            # create figure with title test, 4x5 inches,
            # red backgound

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

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

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



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!