

# Image processing and computer graphics

Part 2

# Introduction to NumPy

- NumPy is the fundamental package for scientific computing with Python. It contains among other things:
  - a powerful N-dimensional array object
  - sophisticated (broadcasting) functions
  - tools for integrating C/C++ and Fortran code
  - useful linear algebra, Fourier transform, and random number capabilities
- To import whole NumPy module type:

**from numpy import \***

- This line will import whole NumPy. If you want to just import one object, for example arrays, type:

**from numpy import array**

- It is recommended to use this convention of import:

**import numpy**

**import numpy as np**

# Array operations

- Create two arrays:

```
a = array([1,2,3,4])
```

```
b = array([2,3,4,5])
```

- and try few simple math operations:

```
a + b
```

```
a * b
```

```
a **b
```

- To automatically create array in some range use **arange()** function. Use **help (arange)**. To make array from 0 to 10 type:

```
x = arange(11.,)
```

# Array operations

- In NumPy, there are predefined some mathematical constants like **pi** or **e**. Type:

$$c = (2 * \pi) / 10$$

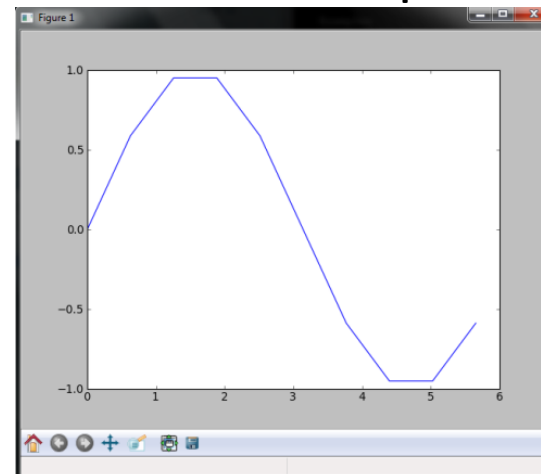
- To multiply whole **x** array by **c** try:

$$x *= c \quad \text{or} \quad (x = x * c)$$

- Now we can get to know another important part of NumPy which is plotting. Create simple function and plot it:

$$y = \sin(x)$$

$$\text{plot}(x, y)$$



# Array operations

- Another way to create array is by using **linspace()**. This NumPy function returns evenly spaced samples calculated over specified interval. Function takes three parameters: from where to start, where is the end and how many samples you need from that interval. Type:

```
x = linspace(0, 2*pi, 50)
```

```
plot (sin(x))
```

- Try different number of samples to see the difference. To find out more about **linspace()** check:

```
help (linspace)
```

# Plotting

- To plot multiple data sets type:

When you put ',' at the end of line, next line will be treated as part of previous

```
plot(x, sin(x),  
      x, sin(2*x))
```

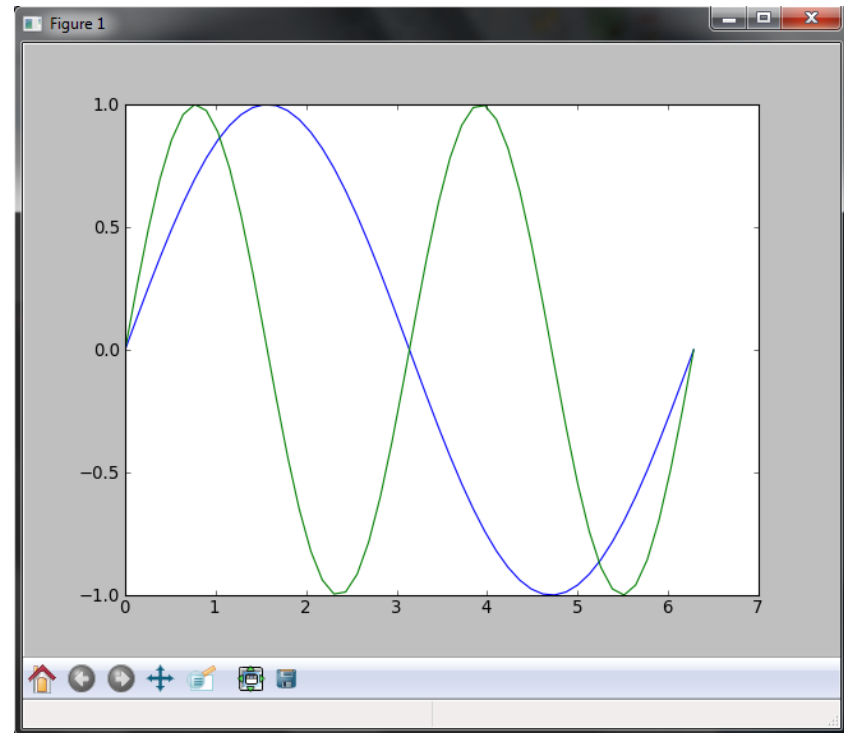
- or

```
plot(x, sin(x))  
plot(x, sin(2*x))
```

- Simple GUI allows to pan, zoom, plot parameters and save image



- **plot()** is part of **matplotlib** module



# Line formatting

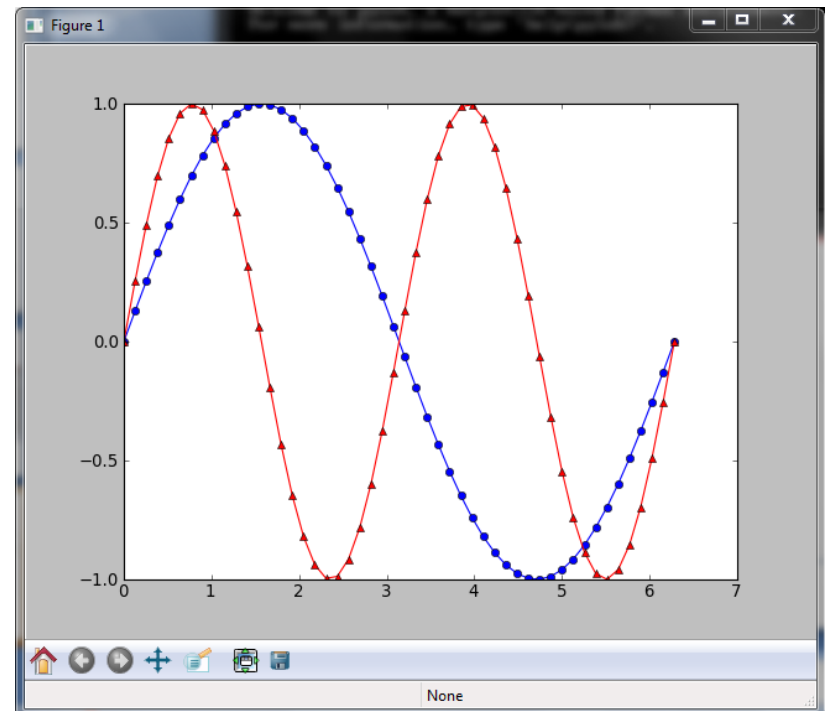
- Type **help(plot)** to see other possibilities of plot function. You can format your data by changing color, adding shaped markers. Let's make  **$\sin(x)$**  in blue and add circles,  **$\sin(2*x)$**  will be red with triangles:

```
plot(x, sin(x), 'b-o',  
     x, sin(2*x), 'r-^')
```

- or

```
plot(x, sin(x), 'b-o')  
plot(x, sin(2*x), 'r-^')
```

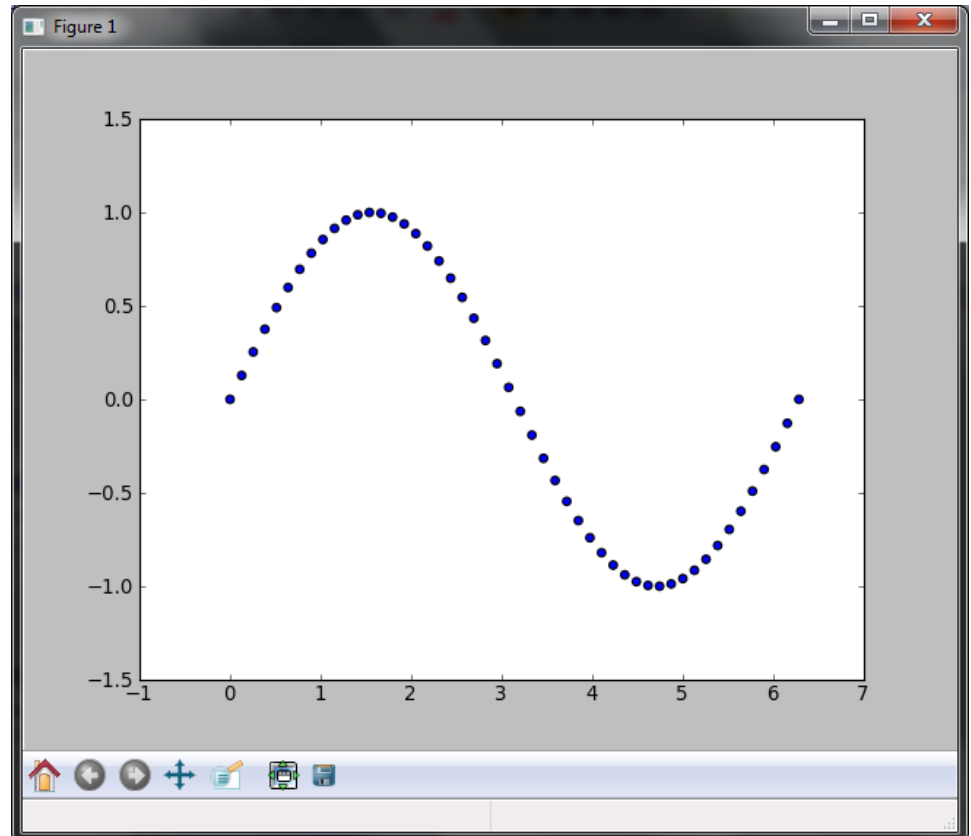
- Try different formatting



# Scatter plot

- To plot some data without interpolation use **scatter()** function:

**scatter(x, sin(x))**





# Multiple figures

- If you want to display more than one plot at the same time you need to create separate objects called **figure()**:

```
t = linspace(0, 2*pi, 50)
```

```
x = sin(t)
```

```
y = cos(t)
```

```
fig1 = figure()
```

window pops out

```
plot(x)
```

plot appears in window

```
fig2 = figure()
```

new window pops out

```
plot(y)
```

plot appears in new window

# Subplot

- **subplot()** function is used to show multiple plots in single window. Plots can be situated in rows or columns, or both. First parameter of **subplot()** is how many rows your window will display, second – how many columns and third one shows which plot will be active at the time:

**a= array([1, 2, 3, 2, 1])**

**b = array([1, 3, 2, 3, 1])**

**subplot (2, 1, 1)**

window pops out, draws first row

**plot(a)**

display plot in first row

**subplot(2, 1, 2)**

draws second row

**plot (b)**

display plot in second row

# Legend

- To add a legend to your plots type:

```
plot (x, label='sin')
```

```
plot (y, label='cos')
```

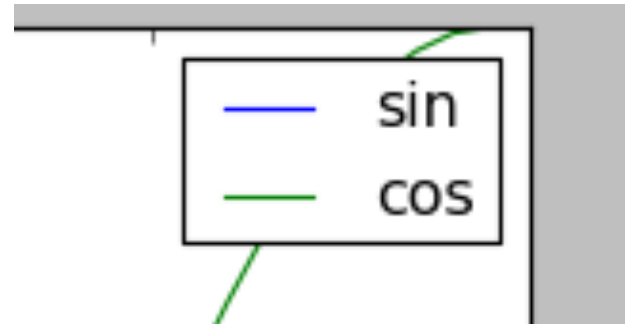
```
legend()
```

- Or

```
plot(x)
```

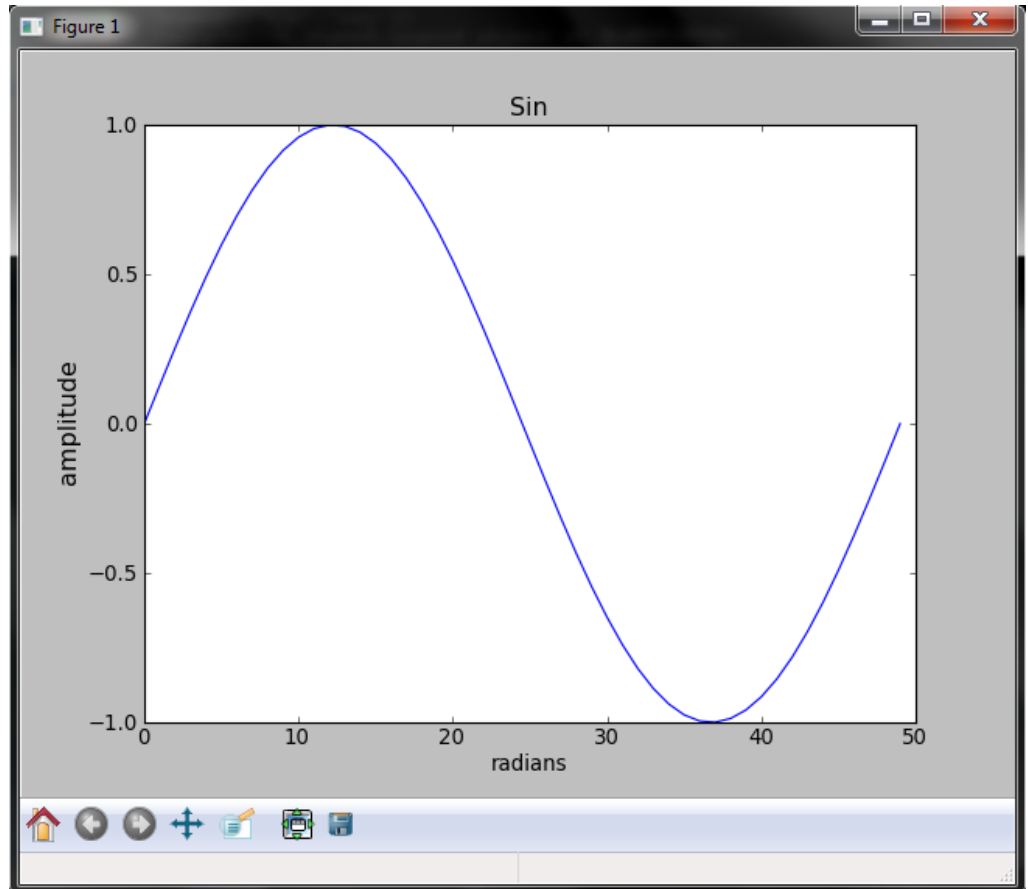
```
plot(y)
```

```
legend(['sin', 'cos'])
```



# Title and axis labels

**plot(x)**  
**xlabel ('radians')**  
**ylabel('amplitude')**  
**title ('Sin')**



# Clearing and closing plots

- Few commands to manage windows with plots:
  - clf()**                clears current figure
  - close()**            close currently active plot window
  - close('all')**        close all plot windows
- If you want to display plots from your script you must type **show()** at the end

# Histogram

- **randn()** return samples from the „standard normal” distribution. Lets try to view a histogram of that array:

**hist(randn(10))**

**hist(randn(100))**

**hist(randn(1000))**

- Default number of bins in **hist()** function is 10. Try more:

**hist(randn(1000), 20)**

**hist(randn(1000), 30)**

# Arrays

- Now we will learn some basic functions of arrays

**a = array([0, 1, 2, 3]) b=zeros((3,5)), c=ones((3,5,2))**

**type(a)**            checking type of array

**a.dtype**            checking numeric type of elements

**a.itemsize**        show number of bytes per element

**a.shape**            returns a tuple listing the length of the array long each dimension

**a.size**             reports the entire number of elements

**a.ndim**            return number of dimensions

**a.nbytes**          return bytes number stored in memory

# Arrays

- Changing element of array is similar like in list:

**`a[0] = 10`**

- Filling whole array with some value

**`a.fill(0)`**

- Second way also works but may be slower with large arrays:

**`a[:] = 1`**

- Slicing an array is the same like slicing a list



# Multi-Dimensional arrays

- Create two dimensional array:

```
a = array([[0, 1, 2, 3], [10, 11, 12, 13]])
```

- Check **a.shape**, **a.size**, **a.ndim**
- Our array has two rows and four columns. To get or set an element you need to address which row and column is he at. For example lets check element in second row and forth column(numeration goes from 0),or whole second row:

```
a[1,3]          a[1]
```

- Now replace value:

```
a[1,3] = -1
```

- Check if that worked

# Multi-Dimensional slicing

`a[0, 3:5 ]`

`a[4::, 4:]`

`a[:, 2]`

`a[2::2, ::2]`

0	1	2	3	4	5
10	11	12	13	14	15
20	21	22	23	24	25
30	31	32	33	34	35
40	41	42	43	44	45
50	51	52	53	54	55

# Slices are references

- Slices are references to memory in the original array.  
Changing values in a slice also changes the original array:

```
a = array((0, 1, 2, 3, 4))
```

```
b = a[2:4]
```

```
b[0] = 10
```

- Now check **a** array for changes

# Fancy indexing

- Instead of slicing you can also use some list which will have desired indexes you want to copy:

```
a = arange(0,80,10)
```

```
list = [1, 2, -3]
```

```
b = a[list]
```

- Another way is to create mask as an array:

```
mask = ([0, 1, 1, 0, 0, 1, 0, 0],dtype = bool)
```

```
b = a[mask]
```

- Unlike slicing, fancy indexing creates copies instead of a view into original array

# Fancy indexing

`a[(0, 1, 2, 3, 4), (1, 2, 3, 4, 5)]`

`a[3:, [0, 2, 5]]`

`mask=array([1, 0, 1, 0, 0, 1],  
dtype=bool)`

`a[mask, 2]`

0	1	2	3	4	5
10	11	12	13	14	15
20	21	22	23	24	25
30	31	32	33	34	35
40	41	42	43	44	45
50	51	52	53	54	55

# Where

- Another way to slice an array is condition. Type:

**a = array([0, 12, 5, 20])**

**a > 10**

- It will return array of the same size, but with boolean type. True if condition is fulfilled and false if it's not. If you want to receive tuple with only values higher than 10, type:

**where(a>10)**

# Newaxis

- **newaxis** is a special index that inserts a new axis in the array at the specified direction. Each **newaxis** increases array's dimensionality by 1:

```
a = array([0, 1, 2])
```

```
shape(a)
```

- Try different axis adding and check **shape(y)** after each one:

```
y = a[newaxis, :]           (1x3)
```

```
y = a[:, newaxis]          (3x1)
```

```
y = a[:, newaxis, newaxis] (3x1x1)
```

# Flattening arrays

- **a.flatten()** converts a multi-dimensional array into a 1-D array. The new array is a **copy of the original data**. Create a 2-D array:, and check **b** value:

```
a = array([[0, 1], [2, 3]])
```

```
b = a.flatten()
```

- **Changing b does not change a**
- **a.flat** is an attribute that returns an iterator object that accesses the data in the M-D array data as a 1-D array. **It references the original memory**. Try:

```
a.flat[:]
```

and check **a** value



# Changing shape

- **a.shape** method used with parameters can change shape of an array it reshapes array **in place**:

```
a = arange(6)
```

```
a.shape
```

```
a.shape = (2, 3)
```

```
a.shape
```

- **reshape** will return **a new array** with a different shape:

```
a.reshape(3, 2)
```

- **Reshape cannot change the number of elements in an array**

# Transpose

- Transpose swaps the order of axes. For 2-D this swaps rows and columns:

```
a = arange(6)
```

```
a.shape
```

- Now to transpose array use **transpose()** method:

```
a.transpose()
```

```
a.shape
```

- Or as a shortcut:

```
a.T()
```

```
a.shape
```

# Squeeze

- **squeeze** removes any dimension with length == 1. To see how it works, first create an array and insert an „extra” dimension in it:

```
a = array([[1, 2, 3, 4, 5, 6]])
```

```
a.shape
```

```
a.shape = (2, 1, 3)
```

- now to remove that „extra” dimension, type:

```
a = a.squeeze()
```

```
a.shape
```

# Diagonals

- Thanks to **diagonal()** you can extract diagonal from an array:  
**a = array ([[11, 21, 31], [12, 22, 32], [13, 23, 33]])**  
**a.diagonal ()**
- You can also use offset to move off the main diagonal:  
**a.diagonal (offset = 1)**  
**a.diagonal (offset = -1)**

# Sum

- Summing arrays can be done by function or method. Both ways work the same. Create an array and sum all elements:

```
a = array([[1, 2, 3, 4, 5, 6]])
```

```
sum(a)
```

or

```
a.sum()
```

- To sum array by columns - type:

```
sum(a, axis = 0)
```

or

```
a.sum(axis = 0)
```

- To sum arrays by rows - type:

```
sum(a, axis = -1)
```

or

```
a.sum(axis = -1)
```

- In similar way you can calculate product:

```
prod(a, axis = 0)
```

or

```
a.prod(axis = 0)
```

# Min/Max

- Similar to **sum()** and **product()** you can calculate minimum and maximum value or it's index in an array:

**amin(a, axis = 0)**      or      **a.min(axis = 0)**

**amax(a, axis = 0)**      or      **a.max(axis = 0)**

- There are also Python's build-in functions **min()** and **max()** but they are slower when performed on multi-dimensional arrays
- To find index of maximum or minimum value - type:  
**argmin(a, axis = 0)**      or      **a.argmin(axis = 0)**  
**argmax(a, axis = 0)**      or      **a.argmax(axis = 0)**

# Statistics

- mean value:

**mean(a, axis = 0)**      or      **a.mean(axis = 0)**

- **average()** does the same thing, but when used with weights, can calculate a weighted average:

**average(a, weights = [1, 2], axis = 0)**

- standard deviation:

**a.std(axis = 0)**

- variance:

**var(a, axis = 0)**      or      **a.var(axis = 0)**

# Other methods

- **clip** method allows you to limit values in an array to a range:  
**a.clip(3, 5)**
- this line will change values > 3 equal to 3, and values > 5 to 5
- **round** method rounds vales in an array. NumPy rounds to even value. You can also use it with **decimals** parameter  
**a = array([1.35, 2.5, 1.5])**  
**a.round()**  
**a.round(decimals = 1)**
- **ptp** (peak to peak)method will calculate **max – min** value along axis:  
**a.ptp(axis = 0)**



# Image display

- To show image in new window use **imshow()** function. First lets import example image from SciPy(famous Lena image):

```
from scipy.misc import lena
```

```
img = lena()
```

```
imshow (img)
```

- You can also add some parameters to **imshow()**. Change color map(if image is in grayscale use **cm.gray**). Values in arrays can also be changed by using **extent** parameter:

```
imshow(img,extent=[-25, 25, -25, 25], cmap=cm.gray)
```

- And add a color bar:

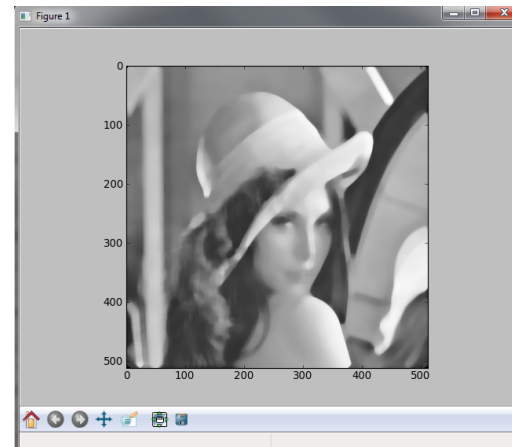
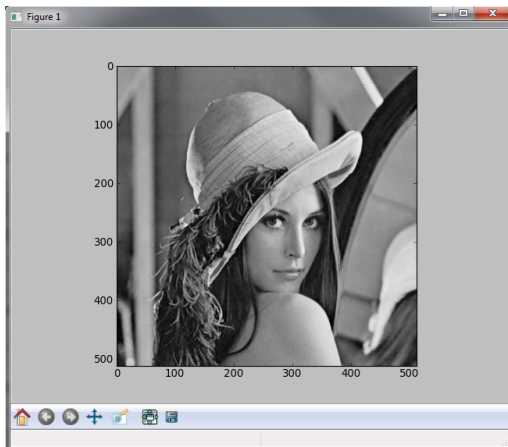
```
colorbar()
```

# Signal module

- To process an image - import **signal** module from SciPy:  
**from scipy.misc import lena**  
**from scipy import signal**
- now we will create an object to store image(it needs to be float 32 because many filters require it):  
**lena = lena().astype(float32)**  
**im.show(lena, cmap = cm.gray)**
- to see what filters you can use from **signal** module - type **signal.** and then hit *TAB* key
- to see filter description (for example gaussian filter) type:  
**help (signal.gaussian)**

# Median filter

- Blur the image by using median filter:  
**median = signal.medfilt2d (lena, [15, 15])**  
**imshow(median, cmap = cmgray)**
- **signal.medfilt2d** takes size of mask as an second parameter



# Noise removal

- Lets add and some noise to **lena** image. First we will need **stats** module from scipy. After that we will use **wiener** filter:

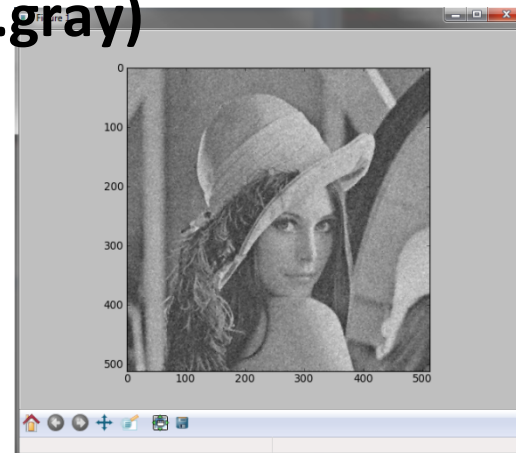
```
from scipy.stats import norm
```

```
lena_noise = lena + norm(0,32) .rvs(lena.shape)
```

```
imshow (lena_noise, cmap = cm.gray)
```

```
cleaned = signal.wiener(lena_noise)
```

```
imshow (cleaned, cmap = cm.gray)
```



# Ndimimage module

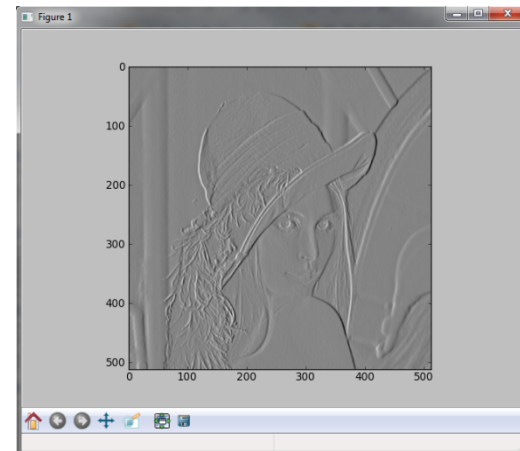
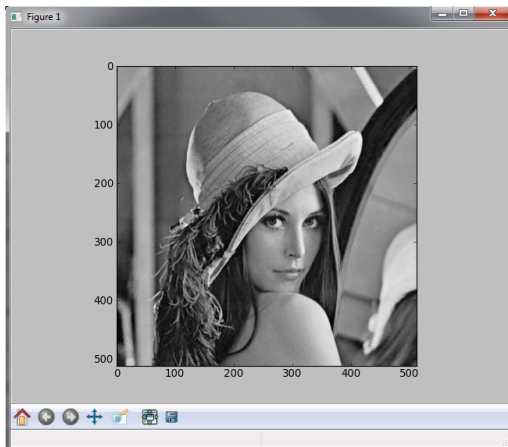
- Another powerful module which contains many filters useful in image processing is **ndimage**.
- In this module you can find ordinary filters, fourier filters, interpolation, measurements and morphology.
- If you want to get access to all those methods simply type:

**from scipy.ndimage import \***

- To see detailed list of filters go to <http://docs.scipy.org/doc/scipy/reference/ndimage.html>

# Edge detection

- Import **sobel** filter from:  
**from scipy.ndimage.filters import sobel**
- Create an object to **sobel** filter and show images:  
**imshow (lena, cmap = cm.gray)**  
**edges = sobel(lena)**  
**imshow (edges, cmpa = cm.gray)**



# Image module

- The **Image** module provides a class with the same name which is used to represent a PIL image. The module also provides a number of factory functions, including functions to load images from files, and to create new images.
- To import module type :  
**import Image**
- and after that type  
**Image.**
- And hit *TAB* key. You can find here alternative way for opening images and different ways of interpolation