Scientific Programming

Lecture A08 – Numpy

Andrea Passerini

Università degli Studi di Trento

2020/10/20

Acknowledgments: Alberto Montresor, Stefano Teso, Numpy Documentation

Table of contents

What is Numpy?

Numpy is a freely available library for performing efficient numerical computations over scalars, vectors, matrices, and more generally N-dimensional tensors.

Features

- Flexible indexing and manipulation of arbitrary dimensional data.
- Many numerical routines (linear algebra, Fourier analysis, local and global optimization, etc.) using a variety of algorithms.
- Clean Python interface to many underlying libraries, which are used as efficient implementations of numerical algorithms.
- Support for random sampling from a variety of distributions.

Some links

Official Numpy website

http://www.numpy.org/

Official documentation

https://docs.scipy.org/doc/numpy/

Source code

https://github.com/numpy/numpy

Importing numpy

The customary idiom to import numpy is:

```
import numpy as np
```

Importing specific sub-modules also makes sense, so feel free to write (in this case for the linear algebra module):

```
import numpy.linalg as la
```

The array class ndarray

An ndarray is an n-dimensional array (a.k.a. tensor)

- Concepts encountered in analysis and algebra can be implemented as arrays: vectors are 1D arrays, matrices are 2D arrays.
- Given an array, its dimensionality and type can be ascertained by using the shape, ndim and dtype attributes:

```
import numpy as np
x = np.zeros(10)
print(x)            [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
print(x.shape)      (10,)
print(x.ndim)       1
print(x.dtype)      float64
```

Creating arrays with various shapes

Creating arrays from scratch can be useful to start off your mathematical code or for debugging purposes.

- Creating an all-zero array with `np.zeros()`
- Creating an all-one array with `np.ones()`

```
import numpy as np
print( np.zeros(2) )                   [ 0.  0.]

print( np.ones( (2,2) ) )              [[ 1.  1.]
                                        [ 1.  1.]]

print( np.zeros( (2,2,2) ) )           [[[ 0.  0.]
                                         [ 0.  0.]]

                                        [[ 0.  0.]
                                         [ 0.  0.]]]
```

Creating arrays with various shapes

To create a diagonal matrix, use the `diag()` and `ones()` methods together.

```python
import numpy as np
print( np.diag( np.ones(3) )
```

```
[[ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]]
```

Creating arrays with various shapes

The numpy (more powerful) analogue of `range()` is `np.arange()`.

```
import numpy as np
print( np.arange(10,100,10) )
print( np.arange(0.0,1.0,0.1))
print( np.linspace(0.0,1.0,11))
print( np.diag( np.arange(5) ) )
```

```
10 20 30 40 50 60 70 80 90]
[ 0.   0.1  0.2  0.3  0.4  0.5  0.6  0.7  0.8  0.9]
[ 0.   0.1  0.2  0.3  0.4  0.5  0.6  0.7  0.8  0.9  1. ]
[[0 0 0 0 0]
 [0 1 0 0 0]
 [0 0 2 0 0]
 [0 0 0 3 0]
 [0 0 0 0 4]]
```

Creating random arrays

- 5-element vector from a uniform distribution in $[0, \ldots, 10[$
- 5-element vector from a normal distribution with $\mu = 1$ and $\sigma = 0.2$
- A 3x3 matrix from a uniform distribution over $\{0, 1\}$

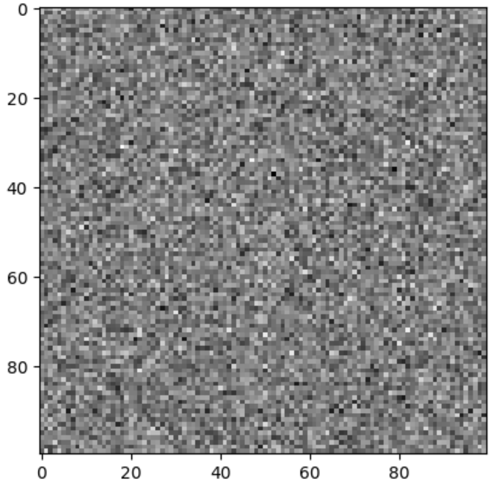
```
import numpy as np
print(np.random.uniform(0, 10, size=5))
print(np.random.normal(1, 0.2, size=5))
print(np.random.randint(0, 2, size=(3, 3)))
```

```
[ 7.91700684  7.41652128  8.1393401   0.8123227   5.50427964]

[ 1.14191823  0.89203955  1.09505607  0.8081311   0.82282836]

[[0 0 0]
 [0 1 1]
 [0 1 1]]
```

Creating random arrays

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

np.random.seed(0)

rm = np.random.normal(0, 1, size=(100, 100))

plt.imshow(rm, cmap="gray", interpolation="nearest")
plt.savefig("random-matrix.png")
```

Creating random arrays



From pandas to numpy and back

It is very easy to "convert" a `Series` or `DataFrame` into an array: as a matter of facts, these Pandas types are built on numpy arrays: their underlying array is accessible through the `values` attribute.

```
import pandas as pd
import numpy as np
iris = pd.read_csv("iris.csv")
print(type(iris.PetalWidth.values))
```

```
<type 'numpy.ndarray'>
```

Re-shaping, un-raveling

The arrangement of entries into rows, columns, tubes, etc. can be changed at any time using the `reshape()` and `ravel()` methods:

- `reshape()` reshapes an array
- `ravel()` flattens an array into a vector

```
import numpy as np
x = np.arange(9)
print(x)                           [0 1 2 3 4 5 6 7 8]
y = x.reshape((3, 3))
print(y)                           [[0 1 2]
                                    [3 4 5]
                                    [6 7 8]]

z = y.ravel()
print(z)                           [0 1 2 3 4 5 6 7 8]
```

Iterating over an array

It is possible to iterate over arrays in several ways:

- Iteration over all elements of an `ndarray`

```
for element in A.flat:
    print(element)
```

- Iteration over multidimensional arrays is done on slices in the first dimension

```
for row in A:
    print(row)
```

Indexing

- `x[i,j]` extracts the element in row `i`, column `j` – just like in standard matrix algebra.
- You can also use the colon `:` notation to select specify all the elements on an axis (e.g. a column, a row, a sub-array).

```python
import numpy as np
x = np.arange(9).reshape((3, 3))
print(x)
print(x[0,:]) # First row
print(x[:,0]) # First column
print(x[1:3, :]) # Subarray
```

```
[[0 1 2]
 [3 4 5]
 [6 7 8]]
[0 1 2]
[0 3 6]
[[3 4 5]
 [6 7 8]]
```

Indexing

The same syntax applies to the n-dimensional case

```
import numpy as np
x = np.arange(5**5).reshape((5,)*5)
print(x.shape)
print(x[0,0,:,:,0])
```

```
(5, 5, 5, 5, 5)
[[  0   5  10  15  20]
 [ 25  30  35  40  45]
 [ 50  55  60  65  70]
 [ 75  80  85  90  95]
 [100 105 110 115 120]]
```

Multi-dimensional arrays

It can be difficult to conceptualize operations over n-dimensional arrays, when n is larger than 3, unless they have concrete semantics.

Example

Assume that a 4D array holds the performance of several sequence alignment algorithms over multiple sequence clusters in multiple DBs:

- The first axis is the index of the algorithm
- The second axis is the index of the database
- The third axis is the index of a cluster of sequences
- The fourth axis is one of three measures of performance: precision, recall, and accuracy

Multi-dimensional arrays

Your tensor looks like this:

`performances[alg][db][cluster][measure]`

You may define symbolic constants to identify the indexes of the columns. E.g.,

```
NEEDLEMAN = 0 # Needleman-Wunsch
SMITH = 1     # Smith-Waterman

PRECISION = 0
RECALL = 1
ACCURACY = 2
```

To extract the accuracy of the Needleman-Wunsch algorithm over all databases and sequence clusters, you can just do:

```
print(performances[NEEDLEMAN,:,:,ACCURACY])
```

Arithmetic broadcasting

Operations between scalars and arrays are broadcast, like in Pandas (and more generally in linear algebra)

```
import numpy as np
x = np.arange(5)*10
print(x)                                  [ 0 10 20 30 40]
y = -x
print(y)                                  [  0 -10 -20 -30 -40]
z = x+y
print(z)                                  [0 0 0 0 0]
```

Arithmetic broadcasting

Operations can update the sub-arrays automatically.

```python
import numpy as np
x = np.arange(16).reshape((4, 4))
print(x)
x[1:3,1:3] += 100
print(x)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]

[[  0   1   2   3]
 [  4 105 106   7]
 [  8 109 110  11]
 [ 12  13  14  15]]
```

Compatible sizes

Operations between arrays of different shapes but compatible sizes are broadcast by "matching" the last (right-most) dimension. e.g, addition between a matrix `x` and a vector `y` broadcasts `y` over all rows of `x`:

```
import numpy as np
x = np.arange(9).reshape((3, 3))
print(x)
y = np.arange(3)
print(y)
z = x + y
print(z)
```

```
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]]

[0 1 2]

[[ 0  2  4]
 [ 3  5  7]
 [ 6  8 10]]
```

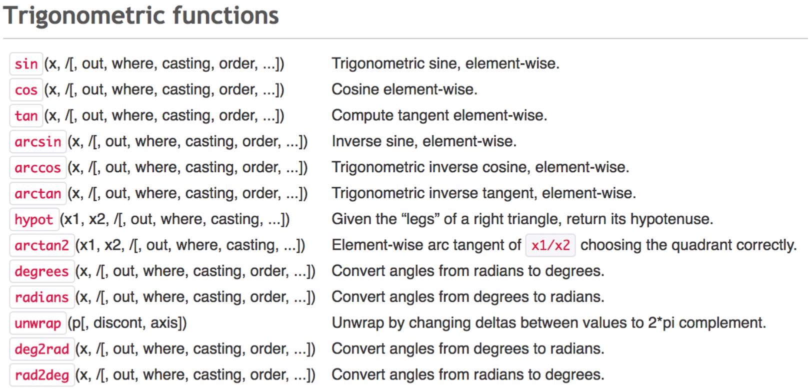
Incompatible sizes

Operations between arrays of incompatible sizes raise an error.

```
>>> np.arange(3) + np.arange(2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast together with
shapes (3,) (2,)
```

Mathematical functions

Numpy contains several mathematical functions, that can be applied to tensors (besides being applicable to scalar values).
`https://docs.scipy.org/doc/numpy/reference/routines.math.html`



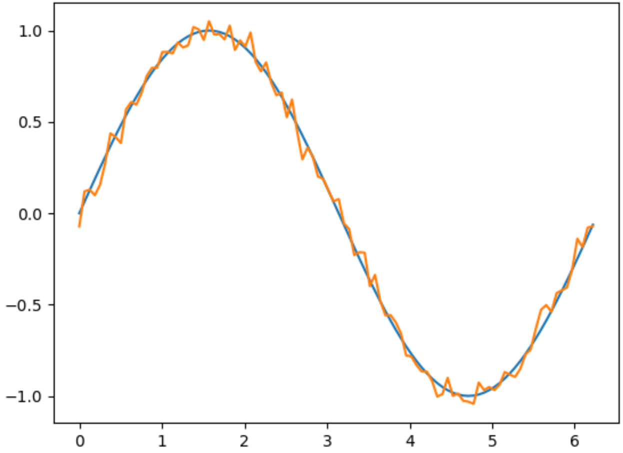
Trigonometric functions

`sin` (x, /[, out, where, casting, order, ...])	Trigonometric sine, element-wise.
`cos` (x, /[, out, where, casting, order, ...])	Cosine element-wise.
`tan` (x, /[, out, where, casting, order, ...])	Compute tangent element-wise.
`arcsin` (x, /[, out, where, casting, order, ...])	Inverse sine, element-wise.
`arccos` (x, /[, out, where, casting, order, ...])	Trigonometric inverse cosine, element-wise.
`arctan` (x, /[, out, where, casting, order, ...])	Trigonometric inverse tangent, element-wise.
`hypot` (x1, x2, /[, out, where, casting, ...])	Given the "legs" of a right triangle, return its hypotenuse.
`arctan2` (x1, x2, /[, out, where, casting, ...])	Element-wise arc tangent of `x1/x2` choosing the quadrant correctly.
`degrees` (x, /[, out, where, casting, order, ...])	Convert angles from radians to degrees.
`radians` (x, /[, out, where, casting, order, ...])	Convert angles from degrees to radians.
`unwrap` (p[, discont, axis])	Unwrap by changing deltas between values to 2*pi complement.
`deg2rad` (x, /[, out, where, casting, order, ...])	Convert angles from degrees to radians.
`rad2deg` (x, /[, out, where, casting, order, ...])	Convert angles from radians to degrees.

Mathematical functions

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

# The x values
x = 2 * np.pi /100 * np.arange(100)

# The y values, uncorrupted
y = np.sin(x)
plt.plot(x, y)

# The y values, now corrupted by Gaussian noise
y += np.random.normal(0, 0.05, size=100)
plt.plot(x, y)

plt.savefig("sin.png")
```

Mathematical functions



Linear Algebra

Numpy provides a number of linear algebraic functions, including dot products, matrix-vector multiplication, and matrix-matrix multiplication.

```
# dot product between vectors
x = np.array([0, 0, 1, 1])
y = np.array([1, 1, 0, 0])
print(np.dot(x, x))                      2
print(np.dot(y, y))                      2
print(np.dot(x, y))                      0

# Matrix-vector product
a = 2 * np.diag(np.ones(4))
print(np.dot(a, x))                      [ 0.  0.  2.  2.]
print(np.dot(a, y))                      [ 2.  2.  0.  0.]

# Matrix-matrix product                  [[-2.  2.  2.  2.]
b = np.ones((4, 4)) - a                   [ 2. -2.  2.  2.]
print(np.dot(a, b))                       [ 2.  2. -2.  2.]
                                          [ 2.  2.  2. -2.]]
```

Linear Algebra

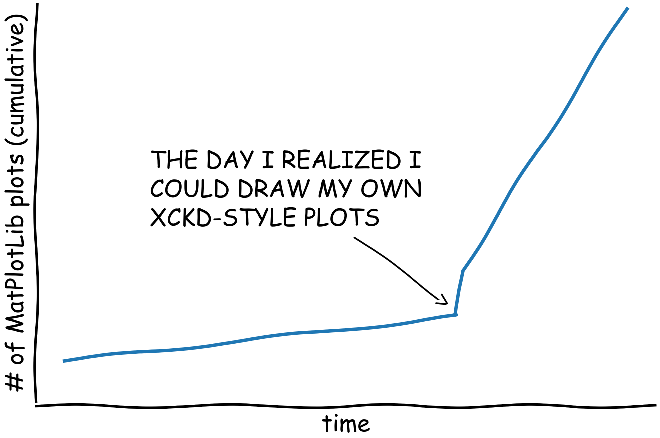
Numpy also provides routines for dealing with eigenvalues/eigenvects, singular value decompositions, and other decompositions.

```
import numpy as np
import numpy.linalg as la
eigenvalues, eigenvectors =
  la.eig(np.ones((4, 4)))
print(eigenvalues)
print(eigenvectors)
```

```
[  0.00000000e+00   4.00000000e+00   0.00000000e+00   2.80731443e-32]

[[-0.8660254  -0.5        -0.8660254  -0.64641535]
 [ 0.28867513 -0.5         0.28867513 -0.32788993]
 [ 0.28867513 -0.5         0.28867513  0.48715264]
 [ 0.28867513 -0.5         0.28867513  0.48715264]]
```

Table of contents

Why use MatPlotLib?



What is MatPlotLib?

MatPlotLib is a plotting library for the Python programming language and its numerical mathematics extension NumPy

Two approaches

- The underlying general purpose MatPlotLib provides the general framework for plotting several types of figures
- The MatPlotLib module `pyplot` provides a declarative way of plotting data, really similar to the syntax of MatLab

`pyplot` is simpler than the general framework, and allows you to produce figures very quickly.
However, the full power is given by the entire framework.

MatPlotLib plotting script

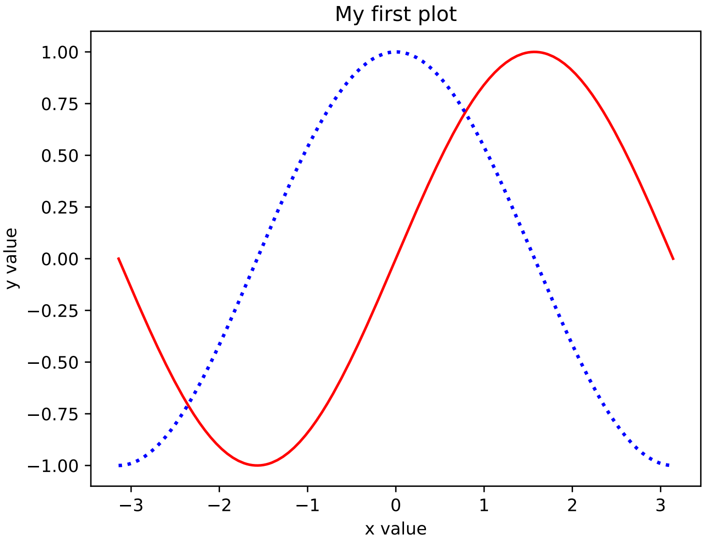
General structure

- import the required modules (`numpy as np` and `matplotlib.pyplot as plt`),
- load or generate some data,
- (optional) customize the appearance of your figure,
- generate a figure (`plot()`, `bar()`, `pie()` etc),
- display it or save it in a file (many formats: PNG, PDF, SVG etc)

Example 1

```
import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(-np.pi, np.pi, 200)
y1 = np.sin(x)
y2 = np.cos(x)
plt.title("My first plot")
plt.plot(x, y1, "r-")
plt.plot(x, y2, color="blue", linewidth=2.0, linestyle=":")
plt.ylabel('y value')
plt.xlabel('x value')
plt.savefig("figure1.pdf", bbox_inches='tight')
```

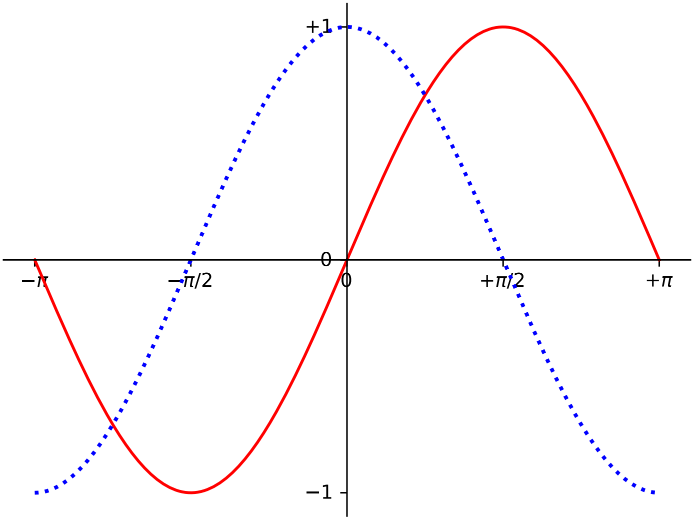
Example 1


My first plot
y value
x value
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−3
−2
−1
0
1
2
3

Example 2

```
plt.xticks([-np.pi, -np.pi/2, 0, np.pi/2, np.pi],
  [r'$-\pi$', r'$-\pi/2$', r'$0$', r'$+\pi/2$', r'$+\pi$'])
plt.yticks([-1, 0, +1],
  [r'$-1$', r'$0$', r'$+1$'])
plt.plot(x, y1, "r-")
plt.plot(x, y2, color="blue", linewidth=2.0, linestyle=":")
ax = plt.gca()
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
ax.spines['bottom'].set_position(('data',0))
ax.yaxis.set_ticks_position('left')
ax.spines['left'].set_position(('data',0))
```

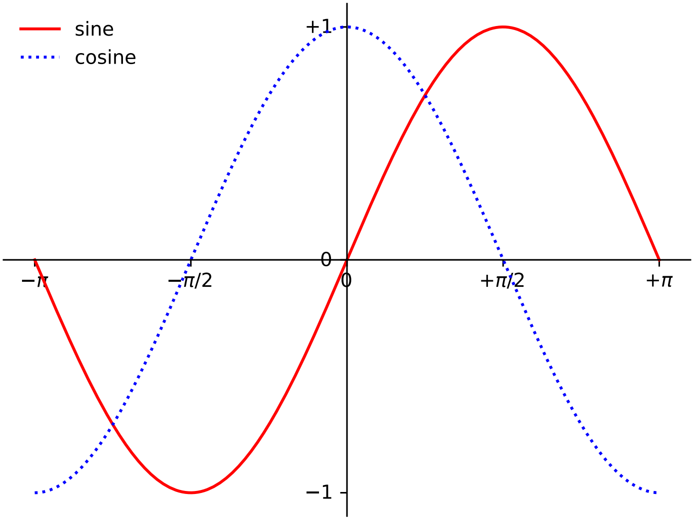
Example 2



Example 3

```
plt.plot(x, y1, "r-", label="sine")
plt.plot(x, y2, "b:", label="cosine")
plt.legend(loc='upper left', frameon=False)
```

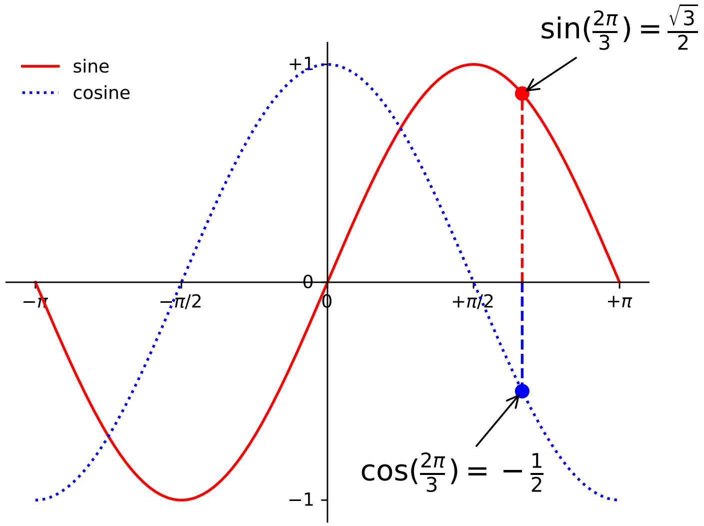
Example 3



Example 4

```
t = 2*np.pi/3
plt.plot([t,t],[0,np.cos(t)], "b--")
plt.scatter([t,],[np.cos(t),], 50, color ='blue')

plt.annotate(r'$\sin(\frac{2\pi}{3})=\frac{\sqrt{3}}{2}$',
  xy=(t, np.sin(t)), xycoords='data',
  xytext=(+10, +30), textcoords='offset points',
  fontsize=16, arrowprops=dict(arrowstyle="->"))

plt.plot([t,t],[0,np.sin(t)], "r--")
plt.scatter([t,],[np.sin(t),], 50, color ='red')

plt.annotate(r'$\cos(\frac{2\pi}{3})=-\frac{1}{2}$',
  xy=(t, np.cos(t)), xycoords='data',
  xytext=(-90, -50), textcoords='offset points',
  fontsize=16, arrowprops=dict(arrowstyle="->"))
```

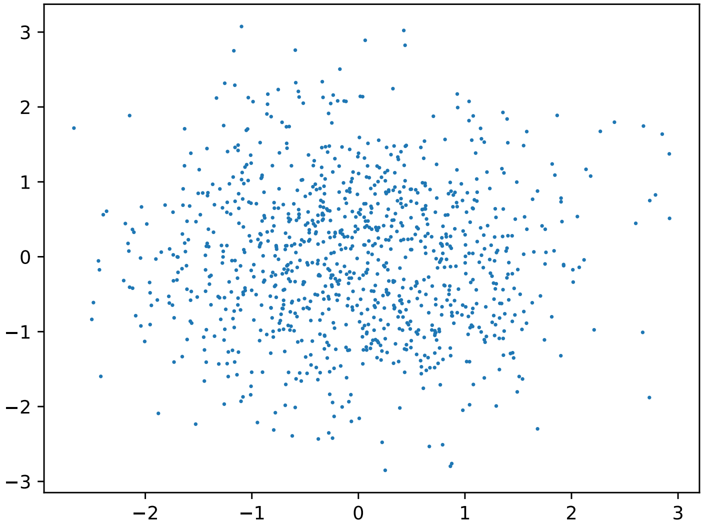
Example 4


sine
cosine
$\sin(\frac{2\pi}{3}) = \frac{\sqrt{3}}{2}$
$\cos(\frac{2\pi}{3}) = -\frac{1}{2}$
$+1$
0
-1
$-\pi$
$-\pi/2$
0
$+\pi/2$
$+\pi$

Example 5

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

n = 1024
X = np.random.normal(0,1,n)
Y = np.random.normal(0,1,n)

plt.scatter(X,Y,s=1)
plt.savefig("figure5.pdf")
```

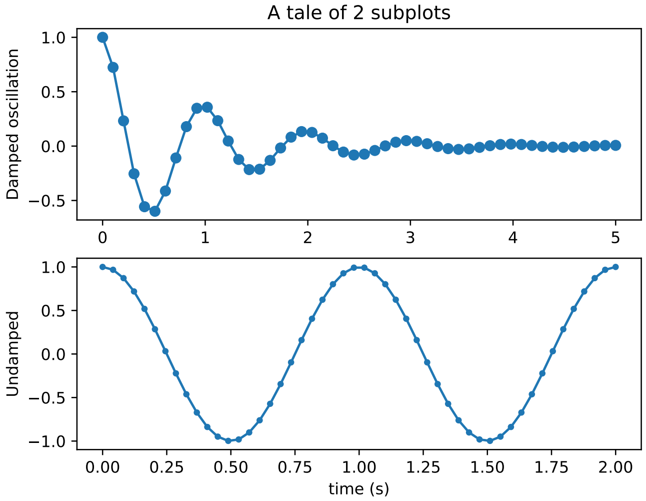
Example 5



Example 6

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

x1 = np.linspace(0.0, 5.0)
x2 = np.linspace(0.0, 2.0)
y1 = np.cos(2 * np.pi * x1) * np.exp(-x1)
y2 = np.cos(2 * np.pi * x2)

plt.subplot(2, 1, 1)
plt.plot(x1, y1, 'o-')
plt.title('A tale of 2 subplots')
plt.ylabel('Damped oscillation')

plt.subplot(2, 1, 2)
plt.plot(x2, y2, '.-')
plt.xlabel('time (s)')
plt.ylabel('Undamped')

plt.savefig("figure6.pdf")
```

Example 6


A tale of 2 subplots
Damped oscillation
1.0
0.5
0.0
−0.5
0
1
2
3
4
5
Undamped
1.0
0.5
0.0
−0.5
−1.0
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
time (s)

Tutorials

http://www.labri.fr/perso/nrougier/teaching/matplotlib/
http://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1003833