Lab – Numpy Scipy

**Introducción**

Python es un gran lenguaje de programación de propósito general por sí mismo, pero con la ayuda de algunas bibliotecas populares (numpy, scipy, matplotlib) se convierte en un poderoso entorno para la computación científica.

Este laboratorio servirá como un rápido curso intensivo del uso de Python para la computación científica

1. **Numpy**

Numpy es la biblioteca principal para la computación científica en Python. Proporciona un objeto de matriz multidimensional de alto rendimiento y herramientas para trabajar con estas matrices. Si ya estás familiarizado con MATLAB, puede que este laboratorio te resulte útil para empezar a utilizar Numpy.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Para utilizar Numpy, primero tenemos que importar el paquete numpy:

[ ]

import numpy as np

**Arrays**

Un array de numpy es una matriz de valores, todos del mismo tipo, y está indexado por una tupla de enteros no negativos. El número de dimensiones es el rango del array; la forma de un array es una tupla de enteros que da el tamaño del arreglo a lo largo de cada dimensión.

Se puede inicializar matrices numpy a partir de listas anidadas de Python, y acceder a los elementos utilizando corchetes:

[ ]

a=np.array([1,2,3])#Create a rank1 array  
print(type(a),a.shape,a[0],a[1],a[2])  
a[0]=5#Change an element of the array  
print(a)

**out**

<class 'numpy.ndarray'> (3,) 1 2 3

[5 2 3]

[ ]

b=np.array([[1,2,3],[4,5,6]])#Create a rank2 array  
print(b)

**out**

[[1 2 3]

[4 5 6]]

[ ]

print(b.shape)  
print(b[0,0],b[0,1],b[1,0])

**out**

(2, 3)

1 2 4

Numpy también proporciona muchas funciones para crear matrices:

[ ]

a=np.zeros((2,2))#Create an array of all zeros  
print(a)

**out**

[[0. 0.]

[0. 0.]]

[ ]

b=np.ones((1,2))#Create an array of all ones  
print(b)

**out**

[[1. 1.]]

[ ]

c=np.full((2,2),7)#Create a constant array  
print(c)

**out**

[[7 7]

[7 7]]

[ ]

d=np.eye(2)#Create a 2x2 identity matrix  
print(d)

**out**

[[1. 0.]

[0. 1.]]

[ ]

e=np.random.random((2,2))#Create an array filled with random values  
print(e)

**out**

[[0.8690054 0.57244319]

[0.29647245 0.81464494]]

**Tensores**

[]

from numpy import array

T = array([

[[1,2,3], [4,5,6], [7,8,9]],

[[11,12,13], [14,15,16], [17,18,19]],

[[21,22,23], [24,25,26], [27,28,29]],

])

print(T.shape)

print(T)

**out**

3, 3, 3)

[[[ 1 2 3]

[ 4 5 6]

[ 7 8 9]]

[[11 12 13]

[14 15 16]

[17 18 19]]

[[21 22 23]

[24 25 26]

[27 28 29]]]

[]

# tensor addition

from numpy import array

A = array([

[[1,2,3], [4,5,6], [7,8,9]],

[[11,12,13], [14,15,16], [17,18,19]],

[[21,22,23], [24,25,26], [27,28,29]],

])

B = array([

[[1,2,3], [4,5,6], [7,8,9]],

[[11,12,13], [14,15,16], [17,18,19]],

[[21,22,23], [24,25,26], [27,28,29]],

])

C = A + B

print(C)

**out**

[[[ 2 4 6]

[ 8 10 12]

[14 16 18]]

[[22 24 26]

[28 30 32]

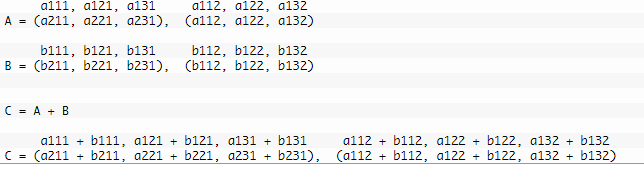
[34 36 38]]

[[42 44 46]

[48 50 52]

[54 56 58]]]

La operación realizada se vería, en notación tensor, de la siguiente manera:



**Array indexing**

Numpy ofrece varias formas de indexar en arrays.

Slicing: Al igual que las listas de Python, los arrays de numpy pueden ser cortados. Como los arrays pueden ser multidimensionales, debes especificar un corte para cada dimensión del array:

[ ]

import numpy as np  
  
#Create the following rank 2 array with shape(3,4)  
#[[1234]  
#[5678]  
#[9101112]]  
a=np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12]])  
  
#Use slicing to pull out the subarray consisting of the first 2 rows  
#and columns 1 and 2;bis the following array of shape(2,2):  
#[[23]  
#[67]]  
b=a[:2,1:3]  
print(b)

**out**

[[2 3]

[6 7]]

Una porción de una matriz es una vista de los mismos datos, por lo que al modificarla, provocara que se modifique la matriz original.

[ ]

print(a[0,1])  
b[0,0]=77 #b[0,0]is the same piece of data as a[0,1]  
print(a[0,1])

**out**

2

77

También se puede mezclar la indexación de enteros con la indexación de trozos. Sin embargo, al hacerlo se obtendrá una matriz de menor rango que la matriz original. Tenga en cuenta que esto es bastante diferente de la forma en que MATLAB maneja el corte de matrices:

[ ]

#Create the following rank2 array with shape(3,4)  
a=np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12]])  
print(a)

**out**

[[ 1 2 3 4]

[ 5 6 7 8]

[ 9 10 11 12]]

Dos formas de acceder a los datos de la fila central del array. Si se combina la indexación de enteros con los slices se obtiene una matriz de menor rango, mientras que si se utilizan sólo slices se obtiene una matriz del mismo rango que la matriz original:

[ ]

row\_r1=a[1,:]#Rank1 view of thesecondrowofa  
row\_r2=a[1:2,:]#Rank2 view of the second row of a  
row\_r3=a[[1],:]#Rank2 view of the second row of a  
print(row\_r1,row\_r1.shape)  
print(row\_r2,row\_r2.shape)  
print(row\_r3,row\_r3.shape)

**out**

[5 6 7 8] (4,)

[[5 6 7 8]] (1, 4)

[[5 6 7 8]] (1, 4)

[ ]

#We can make the same distinction when accessing columns of an array:  
col\_r1=a[:,1]  
col\_r2=a[:,1:2]  
print(col\_r1,col\_r1.shape)  
print()  
print(col\_r2,col\_r2.shape)

out

[ 2 6 10] (3,)

[[ 2]

[ 6]

[10]] (3, 1)

**Indexación de arrays enteros**: Cuando indexas en arrays de numpy usando slicing, la vista del array resultante siempre será un subarray del array original. En cambio, la indexación de arrays enteros te permite construir arrays arbitrarios utilizando los datos de otro array. Aquí hay un ejemplo:

[ ]

a=np.array([[1,2],[3,4],[5,6]])  
  
#An example of integer array indexing.  
#There turned array will have shape(3,) and  
print(a[[0,1,2],[0,1,0]])  
  
#The above example of integer array indexing is equivalent to this:  
print(np.array([a[0,0],a[1,1],a[2,0]]))

**out**

[1 4 5]

[1 4 5]

[ ]

#When using integer array indexing,you can reuse the same  
#element from the source array:  
print(a[[0,0],[1,1]])  
  
#Equivalent to the previous integer array indexing example  
print(np.array([a[0,1],a[0,1]]))

**out**

[2 2]

[2 2]

Un truco útil con la indexación de matrices de números enteros es seleccionar o mutar un elemento de cada fila de una matriz:

[ ]

#Create a new array from which we will select elements  
a=np.array([[1,2,3],[4,5,6],[7,8,9],[10,11,12]])  
print(a)

**out**

[[ 1 2 3]

[ 4 5 6]

[ 7 8 9]

[10 11 12]]

[ ]

#Create an array of indices  
b=np.array([0,2,0,1])  
  
#Select one element from each row of a using the indices in b  
print(a[np.arange(4),b])#Prints"[16711]"

out

[ 1 6 7 11]

[ ]

#Mutate one element from each row of a using the indices in b  
a[np.arange(4),b]+=10  
print(a)

**out**

[[11 2 3]

[ 4 5 16]

[17 8 9]

[10 21 12]]

**Indexación booleana de matrices**: La indexación booleana de matrices permite seleccionar elementos arbitrarios de una matriz. Con frecuencia, este tipo de indexación se utiliza para seleccionar los elementos de una matriz que satisfacen alguna condición. He aquí un ejemplo:

[ ]

import numpy as np

a=np.array([[1,2],[3,4],[5,6]])  
  
bool\_idx=(a>2)#Find the elements of a that are bigger than 2;

#this returns a numpy array of Booleans of the same

#shape as a,where each slot of bool\_idx tells

#whether that element of a is>2.

print(bool\_idx)

**out**

[[False False]

[ True True]

[ True True]]

[ ]

#We use boolean array indexing to construct a rank1 array  
#consisting of the elements of a corresponding to the True values  
#of bool\_idx  
print(a[bool\_idx])  
  
#We can do all of the above in a single concise statement:  
print(a[a>2])

**out**

[3 4 5 6]

[3 4 5 6]

**Datatypes**

Todo array de numpy es una red de elementos del mismo tipo. Numpy proporciona un gran conjunto de tipos de datos numéricos que puedes utilizar para construir arrays. Numpy trata de adivinar un tipo de datos cuando creas un array, pero las funciones que construyen arrays normalmente también incluyen un argumento opcional para especificar explícitamente el tipo de datos. He aquí un ejemplo:

[ ]

x=np.array([1,2])#Let numpy choose the datatype  
y=np.array([1.0,2.0])#Let numpy choose the datatype  
z=np.array([1,2],dtype=np.int64)#Force a particular datatype  
  
print(x.dtype,y.dtype,z.dtype)

out

int64 float64 int64

**Array math**

Las funciones matemáticas básicas operan de forma elemental sobre arrays, y están disponibles tanto como sobrecargas de operadores como funciones en el módulo numpy:

[ ]

x=np.array([[1,2],[3,4]],dtype=np.float64)  
y=np.array([[5,6],[7,8]],dtype=np.float64)  
  
#Element wise sum;both produce the array  
print(x+y)  
print(np.add(x,y))

**out**

[[ 6. 8.]

[10. 12.]]

[[ 6. 8.]

[10. 12.]]

[ ]

#Element wise difference;both produce the array  
print(x-y)  
print(np.subtract(x,y))

**out**

[[-4. -4.]

[-4. -4.]]

[[-4. -4.]

[-4. -4.]]

[ ]

#Element wise product;both produce the array  
print(x\*y)  
print(np.multiply(x,y))

**out**

[[ 5. 12.]

[21. 32.]]

[[ 5. 12.]

[21. 32.]]

[ ]

#Element wise division;both produce the array  
#[[0.2 0.33333333]  
#[0.42857143 0.5]]  
print(x/y)  
print(np.divide(x,y))

**out**

[[0.2 0.33333333]

[0.42857143 0.5 ]]

[[0.2 0.33333333]

[0.42857143 0.5 ]]

[ ]

#Element wise square root;produces the array  
#[[1. 1.41421356]  
#[1.732050812. ]]  
print(np.sqrt(x))

**out**

[[1. 1.41421356]

[1.73205081 2. ]]

Ten en cuenta que, a diferencia de MATLAB, \* es una multiplicación elemental, no una multiplicación matricial. En su lugar, utilizamos la función dot para calcular productos internos de vectores, para multiplicar un vector por una matriz y para multiplicar matrices. dot está disponible como una función en el módulo numpy y como un método de instancia de los objetos array:

[ ]

x=np.array([[1,2],[3,4]])  
y=np.array([[5,6],[7,8]])  
  
v=np.array([9,10])  
w=np.array([11,12])  
  
#Inner product of vectors; both produce 219  
print(v.dot(w))  
print(np.dot(v,w))

**out**

219

219

También se puede utilizar el operador @ que es equivalente al operador punto de numpy.

[ ]

print(v@w)

**out**

219

[ ]

#Matrix/vector product;both produce the rank1 array[29 67]  
print(x.dot(v))  
print(np.dot(x,v))  
print(x@v)

**out**

[29 67]

[29 67]

[29 67]

[ ]

#Matrix/matrix product;both produce the rank2 array  
#[[19 22]  
#[43 50]]  
print(x.dot(y))  
print(np.dot(x,y))  
print(x@y)

**out**

[[19 22]

[43 50]]

[[19 22]

[43 50]]

[[19 22]

[43 50]]

Numpy proporciona muchas funciones útiles para realizar cálculos en matrices; una de las más útiles es sum:

[ ]

x=np.array([[1,2],[3,4]])  
  
print(np.sum(x))#Compute sum of all elements;prints"10"  
print(np.sum(x,axis=0))#Compute sum of each column;prints"[46]"  
print(np.sum(x,axis=1))#Compute sum of each row;prints"[37]"

**out**

10

[4 6]

[3 7]

Aparte de calcular funciones matemáticas utilizando arrays, a menudo necesitamos remodelar o manipular de otro modo los datos de los arrays. El ejemplo más sencillo de este tipo de operaciones es la transposición de una matriz; para transponer una matriz, basta con utilizar el atributo T de un objeto array:

[ ]

print(x)  
print("transpose\n",x.T)

[[1 2]

[3 4]]

Transpose

**out**

[[1 3]

[2 4]]

[ ]

v=np.array([[1,2,3]])  
print(v)  
print("transpose\n",v.T)

out

[[1 2 3]]

Transpose

[[1]

[2]

[3]]

**Broadcasting**

Broadcasting es un poderoso mecanismo que permite a numpy trabajar con arreglos de diferentes formas al realizar operaciones aritméticas. Frecuentemente tenemos un arreglo más pequeño y otro más grande, y queremos usar el arreglo más pequeño varias veces para realizar alguna operación en el arreglo más grande.

Por ejemplo, supongamos que queremos añadir un vector constante a cada fila de una matriz. Podríamos hacerlo así:

[ ]

#We will add the vector v to each row of the matrix x,  
#storing the result in the matrix y  
x=np.array([[1,2,3],[4,5,6],[7,8,9],[10,11,12]])  
v=np.array([1,0,1])  
y=np.empty\_like(x) #Create an empty matrix with the same shape as x  
  
#Add the vector v to each row of the matrix x with an explicit loop  
for i in range(4):  
y[i,:]=x[i,:]+v  
  
print(y)

**out**

[[ 2 2 4]

[ 5 5 7]

[ 8 8 10]

[11 11 13]]

Esto funciona; sin embargo, cuando la matriz x es muy grande, calcular un bucle explícito en Python podría ser lento. Obsérvese que añadir el vector v a cada fila de la matriz x es equivalente a formar una matriz vv apilando múltiples copias de v verticalmente, y luego realizando la suma elemental de x y vv. Podríamos implementar este enfoque así:

[ ]

vv=np.tile(v,(4,1)) #Stack 4 copies of v ontop of each other  
print(vv) #Prints"[[101]

#[101]  
#[101]  
#[101]]"

**out**

[[1 0 1]

[1 0 1]

[1 0 1]

[1 0 1]]

[ ]

y=x+vv #Add x and vv element wise  
print(y)

**out**

[[ 2 2 4]

[ 5 5 7]

[ 8 8 10]

[11 11 13]]

La difusión de Numpy nos permite realizar este cálculo sin crear realmente múltiples copias de v. Considere esta versión, utilizando la difusión:

[ ]

import numpy as np  
  
#We will add the vector v to each row of the matrix x,  
#storing the result in the matrix y  
x=np.array([[1,2,3],[4,5,6],[7,8,9],[10,11,12]])  
v=np.array([1,0,1])  
y=x+v #Add v to each row of x using broadcasting  
print(y)

**out**

[[ 2 2 4]

[ 5 5 7]

[ 8 8 10]

[11 11 13]]

La línea y = x + v funciona aunque x tenga la forma (4, 3) y v tenga la forma (3,) debido a la difusión; esta línea funciona como si v tuviera realmente la forma (4, 3), donde cada fila fuera una copia de v, y la suma se realizara por elementos.

La transmisión de dos matrices juntas sigue las siguientes reglas

1. Si las matrices no tienen el mismo rango, se antepone la forma de la matriz de menor rango con 1s hasta que ambas formas tengan la misma longitud.
2. Se dice que las dos matrices son compatibles en una dimensión si tienen el mismo tamaño en la dimensión, o si una de las matrices tiene tamaño 1 en esa dimensión.
3. Las matrices pueden emitirse juntas si son compatibles en todas las dimensiones.
4. Tras la emisión, cada matriz se comporta como si tuviera una forma igual al máximo elemental de las formas de las dos matrices de entrada.
5. En cualquier dimensión en la que una matriz tenga tamaño 1 y la otra matriz tenga tamaño mayor que 1, la primera matriz se comporta como si se copiara a lo largo de esa dimensión

Las funciones que soportan la transmisión se conocen como funciones universales. Puedes encontrar la lista de todas las funciones universales en la documentación.

Estas son algunas aplicaciones de la difusión:

[ ]

#Compute outer product of vectors  
v=np.array([1,2,3])#v has shape(3,)  
w=np.array([4,5])#w has shape(2,)  
#To compute an outer product,we first reshape v to be a column  
#vector of shape(3,1);we can then broadcast it against w to yield  
#an output of shape(3,2),which is the outer product of v and w:  
  
print(np.reshape(v,(3,1))\*w)

**out**

[[ 4 5]

[ 8 10]

[12 15]]

[ ]

#Add a vector to each row of a matrix  
x=np.array([[1,2,3],[4,5,6]])  
#x has shape(2,3)and v has shape(3,)so they broadcast to(2,3),  
#giving the following matrix:  
  
print(x+v)

**out**

[[2 4 6]

[5 7 9]]

[ ]

#Add a vector to each column of a matrix  
#x has shape(2,3)and was shape(2,).  
#If we transpose x then it has shape(3,2)and can be broadcast  
#against w to yield a result of shape(3,2); transposing this result  
#yields the final result of shape(2,3) which is the matrix x with  
#the vector w added to each column.Gives the following matrix:  
  
print((x.T+w).T)

**out**

[[ 5 6 7]

[ 9 10 11]]

[ ]

#Another solution is to reshape w to be a row vector of shape(2,1);  
#we can then broadcast it directly against x to produce the same  
#output.  
print(x+np.reshape(w,(2,1)))

**out**

[[ 5 6 7]

[ 9 10 11]]

[ ]

#Multiply a matrix by a constant:  
#x has shape(2,3).Numpy treats scalars as arrays of shape();  
#these can be broadcast together to shape(2,3),producing the  
#following array:  
print(x\*2)

**out**

[[ 2 4 6]

[ 8 10 12]]

Broadcasting suele hacer que tu código sea más conciso y rápido, por lo que deberías esforzarte en utilizarla siempre que sea posible.

Este breve resumen ha tocado muchas de las cosas importantes que necesitas saber sobre numpy, pero está lejos de ser completo.

**Matplotlib**

Matplotlib es una biblioteca de trazado. En esta sección se da una breve introducción al módulo matplotlib.pyplot, que proporciona un sistema de trazado similar al de MATLAB.

[ ]

import matplotlib.pyplot as plt

Ejecutando este comando especial de iPython, visualizaremos los gráficos en línea:

[ ]

%matplotlib inline

**Plotting**

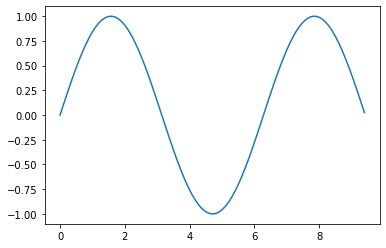
La función más importante de matplotlib es plot, que permite trazar datos en 2D. He aquí un ejemplo sencillo:

[ ]

#Compute the x and y coordinates for points on a sine curve  
x=np.arange(0,3\*np.pi,0.1)  
y=np.sin(x)  
  
#Plot the points using matplotlib  
plt.plot(x,y)

**out**

[<matplotlib.lines.Line2D at 0x1ec2dbd9640>]



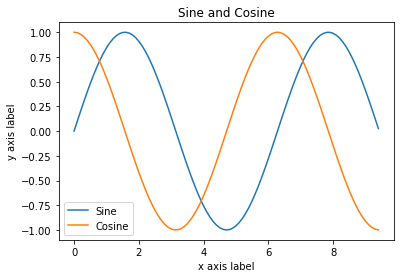
Con un poco de trabajo extra podemos trazar fácilmente varias líneas a la vez, y añadir un título, una leyenda y etiquetas en los ejes:

[ ]

y\_sin=np.sin(x)  
y\_cos=np.cos(x)  
  
#Plot the points using matplotlib  
plt.plot(x,y\_sin)  
plt.plot(x,y\_cos)  
plt.xlabel('xaxislabel')  
plt.ylabel('yaxislabel')  
plt.title('SineandCosine')  
plt.legend(['Sine','Cosine'])

**out**

<matplotlib.legend.Legend at 0x1ec2da8b040>



**Subplots**

La función subplot() añade una subtrama a una figura actual en la posición de la cuadrícula especificada. Es similar a la función subplots() sin embargo, a diferencia de subplots(), añade un subplot a la vez. Por lo tanto, para crear múltiples gráficos se necesitan varias líneas de código con la función subplot(). Otra desventaja de la función subplot() es que borra el gráfico preexistente en su figura.

Es una wrapper de Figure.add\_subplot.

***Sintaxis:***

subplot(nrows, ncols, index, \*\*kwargs)

subplot(pos, \*\*kwargs)

subplot(ax)

***Parámetros*** :

*args*: Un entero de 3 dígitos o tres enteros separados que describen la posición del subplot.

*pos* es un entero de tres dígitos donde el primer, segundo y tercer entero son nrows,ncols, index.

*projection* : [{None, 'aitoﬀ', 'hammer', 'lambert', 'mollweide', 'polar', 'rectilínear', str}, opcional]. El tipo de proyección del subgrupo (Ejes). El valor por defecto None da como resultado una proyección 'rectilínea'.

*label* : [str] Una etiqueta para los ejes devueltos.

\*\**kwargs*: Este método también toma los argumentos de palabra clave para la clase base de los ejes devueltos;

excepto el argumento ﬁgure, por ejemplo facecolor.

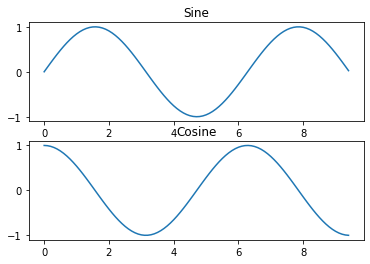
***Devuelve*** : Una subclase axes.SubplotBase de Axes o una subclase de Axes. La clase base de los ejes devuelta depende de la proyección utilizada.

Puede trazar diferentes cosas en la misma figura utilizando la función de subtrazado. He aquí un ejemplo:

[ ]

#Compute the x and y coordinates for points on sine and cosine curves  
x=np.arange(0,3\*np.pi,0.1)  
y\_sin=np.sin(x)  
y\_cos=np.cos(x)  
  
#Configure una rejilla de subtrama que tenga una altura de 2 y una anchura de 1,  
#y establecer la primera de estas subtramas como activa.

plt.subplot(2,1,1)  
  
#Makethefirstplot  
plt.plot(x,y\_sin)  
plt.title('Sine')  
  
#Set the second subplot as active,and make the second plot.  
plt.subplot(2,1,2)  
plt.plot(x,y\_cos)  
plt.title('Cosine')  
  
#Show the figure.  
plt.show()



1. **Scipy**

SciPy en Python es una biblioteca de código abierto utilizada para resolver problemas matemáticos, científicos, de ingeniería y técnicos. Permite a los usuarios manipular los datos y visualizarlos utilizando una amplia gama de comandos de alto nivel de Python. SciPy se basa en la extensión NumPy de Python.

**Subpaquetes de SciPy:**

|  |  |
| --- | --- |
| **Package Name** | **Description** |
| scipy.io | Entrada/salida de archivos |
| scipy.special | Función especial |
| scipy.linalg | Operación de álgebra lineal |
| scipy.interpolate | Interpolación |
| scipy.optimize | Optimización y ajuste |
| scipy.stats | Estadísticas y números aleatorios |
| scipy.integrate | Integración numérica |
| scipy.fftpack | Transformadas rápidas de Fourier |
| scipy.signal | Procesamiento de señales |
| scipy.ndimage | Manipulación de imágenes |

**Por qué usar SciPy**

* SciPy contiene variedades de subpaquetes que ayudan a resolver los problemas más comunes relacionados con la computación científica.
* El paquete SciPy en Python es la biblioteca científica más utilizada, sólo superada por la biblioteca científica GNU para C/C++ o Matlab.
* Es fácil de usar y de entender, así como una potencia de cálculo rápida.
* Puede operar en una matriz de la biblioteca NumPy.

Antes de empezar a aprender SciPy Python, es necesario conocer la funcionalidad básica así como los diferentes tipos de un array de NumPy

La forma estándar de importar módulos de SciPy y Numpy:

[]

from scipy import special #same for other modules

import numpy as np

**Paquete de entrada/salida de archivos:**

Scipy, paquete de E/S, tiene una amplia gama de funciones para trabajar con diferentes formatos de archivos que son Matlab, Arff, Wave, Matrix Market, IDL, NetCDF, TXT, CSV y formato binario.

Tomemos un ejemplo de formato de archivo Python SciPy como el que se utiliza regularmente en MatLab:

[]

import numpy as np

from scipy import io as sio

array = np.ones((4, 4))

sio.savemat('example.mat', {'ar': array})

data = sio.loadmat(‘example.mat', struct\_as\_record=True)

data['ar']

out:

array([[ 1., 1., 1., 1.],

[ 1., 1., 1., 1.],

[ 1., 1., 1., 1.],

[ 1., 1., 1., 1.]])

**Paquete de funciones especiales**

El paquete scipy.special contiene numerosas funciones de física matemática.

La función especial de SciPy incluye Raíz Cúbica, Exponencial, Suma Log Exponencial, Lambert, Permutación y Combinaciones, Gamma, Bessel, hipergeométrica, Kelvin, beta, cilindro parabólico, Error Relativo Exponencial, etc.

Algunos ejemplos de operaciones:

1. ***Raiz cubica***

[]

from scipy.special import cbrt

#Find cubic root of 27 & 64 using cbrt() function

cb = cbrt([27, 64])

#print value of cb

print(cb)

**out**

[3. 4.]

1. **Permutaciones**

[]

from scipy.special import comb

#find combinations of 5, 2 values using comb(N, k)

com = comb(5, 2, exact = False, repetition=True)

print(com)

**out**

15.0

1. **Función exponencial**

La función exponencial calcula el 10\*\*x de forma elemental.

[]

from scipy.special import exp10

#define exp10 function and pass value in its

exp = exp10([1,10])

print(exp)

**out**

[1.e+01 1.e+10]

**Álgebra lineal con SciPy**

El Álgebra Lineal de SciPy es una implementación de las librerías BLAS y ATLAS LAPACK.

El rendimiento del Álgebra Lineal es muy rápido en comparación con BLAS y LAPACK.

La rutina de álgebra lineal acepta un objeto de matriz bidimensional y la salida es también una matriz bidimensional.

Ahora vamos a hacer algunas pruebas con scipy.linalg,

1. **Cálculo del determinante de una matriz bidimensional,**

[]

from scipy import linalg

import numpy as np

#define square matrix

two\_d\_array = np.array([ [4,5], [3,2] ])

#pass values to det() function

linalg.det( two\_d\_array )

**out**

-7.0

**Procesamiento de imágenes con SciPy - scipy.ndimage**

scipy.ndimage es un submódulo de SciPy que se utiliza principalmente para realizar una operación relacionada con la imagen

ndimage significa la imagen de "n" dimensiones.

El procesamiento de imágenes de SciPy proporciona transformaciones geométricas (rotar, recortar, voltear), filtrado de imágenes (nitidez y eliminación), visualización de imágenes, segmentación de imágenes, clasificación y extracción de características.

El paquete MISC de SciPy contiene imágenes preconstruidas que pueden utilizarse para realizar tareas de manipulación de imágenes

Ejemplo: Tomemos un ejemplo de transformación geométrica de imágenes

[]

from scipy import datasets

from matplotlib import pyplot as plt

import numpy as np

#get face image of panda from misc package

panda = datasets.face()

#plot or show image of face

plt.imshow( panda )

plt.show()

**out**



Ahora volteamos la imagen actual:

[]

#Flip Down using scipy misc.face image

flip\_down = np.flipud(panda)

plt.imshow(flip\_down)

plt.show()

**out**



Ejemplo: Rotación de una imagen usando Scipy,

[]

from scipy import ndimage, misc

from matplotlib import pyplot as plt

panda = misc.face()

#rotatation function of scipy for image – image rotated 135 degree

panda\_rotate = ndimage.rotate(panda, 135)

plt.imshow(panda\_rotate)

plt.show()

**out**



**Integración con Scipy - Integración numérica**

Cuando integramos alguna función donde no es posible la integración analítica, necesitamos recurrir a la integración numérica

SciPy proporciona la funcionalidad para integrar la función con la integración numérica.

La librería scipy.integrate tiene integración simple, doble, triple, múltiple, cuadrática gaussiana, Romberg, trapezoidal y reglas de Simpson.

Ejemplo: Ahora tomemos un ejemplo de integración simple



Aquí ***a*** es el límite superior y ***b*** es el límite inferior

[]

from scipy import integrate

# take f(x) function as f

f = lambda x : x\*\*2

#single integration with a = 0 & b = 1

integration = integrate.quad(f, 0 , 1)

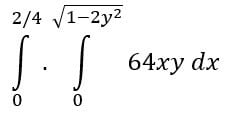
print(integration)

**out**

(0.33333333333333337, 3.700743415417189e-15)

Aquí la función devuelve dos valores, en los que el primer valor es la integración y el segundo valor es el error estimado en la integral.

Ejemplo: Ahora tomemos un ejemplo de integración doble en SciPy. Encontramos la doble integración de la siguiente ecuación,



[]

from scipy import integrate

import numpy as np

#import square root function from math lib

from math import sqrt

# set fuction f(x)

f = lambda x, y : 64 \*x\*y

# lower limit of second integral

p = lambda x : 0

# upper limit of first integral

q = lambda y : sqrt(1 - 2\*y\*\*2)

# perform double integration

integration = integrate.dblquad(f , 0 , 2/4, p, q)

print(integration)

**out**

(3.0, 9.657432734515774e-14)

[]

import numpy as np

check = np.zeros((8, 8))

check[::2, 1::2] = 1

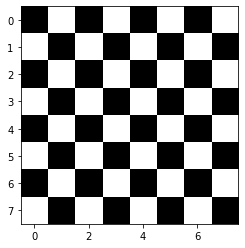
check[1::2, ::2] = 1

import matplotlib.pyplot as plt

plt.imshow(check, cmap='gray', interpolation='nearest')

**out**

<matplotlib.image.AxesImage at 0x1ec3088ae20>



**Acceso a Archivos**

Scipy permite un acceso rápido para operaciones I/O como en el siguiente ejemplo:

[]

import numpy as np

from scipy import io as sio

array = np.ones((4, 4))

sio.savemat('example.mat', {'ar': array})

data = sio.loadmat('example.mat', struct\_as\_record=True)

data['ar']

**out**

array([[1., 1., 1., 1.],

[1., 1., 1., 1.],

[1., 1., 1., 1.],

[1., 1., 1., 1.]])