Branch: master

Find file Copy path

ma591spring2020 / material / numpy_scipy.ipynb

arvindks updated lecture 2/28

607b6ab on Feb 28

0 contributors

Raw Blame History

1155 lines (1154 sloc) 73.1 KB

Scientific Computing

NumPy/SciPy Overview ¶

- NumPy is a library for numeric array manipulations.
- SciPy is a library for scientific computing.
- Their functionality has some overlap but they are mostly distinct.

Useful resources

- Numpy tutorial (https://docs.scipy.org/doc/numpy/user/quickstart.html)
- Langtangen's books
- NumPy for Matlab Users (https://docs.scipy.org/doc/numpy/user/numpy-for-matlab-users.html)
- SciPy website (http://scipy.org/)

Installing numpy and scipy

The best way is to use

```
pip install numpy
pip install scipy
```

Anaconda also helps with installation. More details can be found here (https://scipy.org/install.html).

Things NumPy is good for

NumPy has the following features:

- `ndarray` which handles matrices/vectors/multidimensional arrays.
- linear algebra, fourier, and random number capabilities.
- Reading and writing files
- interfacing with C/Fortran code

Taken from the NumPy tutorial (https://docs.scipy.org/doc/numpy/user/quickstart.html)

1. Array Creation
   - `arange, array, copy, empty, empty_like, eye, fromfile, fromfunction, identity, linspace, logspace, mgrid, ogrid, ones, ones_like, r, zeros, zeros_like`
2. Conversions
   - `ndarray.astype, atleast_1d, atleast_2d, atleast_3d, mat`
3. Manipulations
   - `array_split, column_stack, concatenate, diagonal, dsplit, dstack, hsplit, hstack, ndarray.item, newaxis, ravel, repeat, reshape, resize, squeeze, swapaxes, take,`

  `repeat, reshape, resize, squeeze, swapaxes, take, transpose, vsplit, vstack`
4. Questions
   - `all, any, nonzero, where`
5. Ordering
   - `argmax, argmin, argsort, max, min, ptp, searchsorted, sort`
6. Operations
   - `choose, compress, cumprod, cumsum, inner, ndarray.fill, imag, prod, put, putmask, real, sum`
7. Basic Statistics
   - `cov, mean, std, var`

Importing NumPy capabilities

The standard way to import `numpy` functionalities is `import numpy` or use an alias `import numpy as np`.

In [1]:
```
import numpy as np
```

We can also import specific functions from numpy. This may be useful when importing submodules. Example:

In [2]:
```
from numpy.linalg import  qr, svd, norm
```

NumPy Arrays

The `ndarray` class handles multidimensional arrays. Type `help(ndarray)`.

Basics of NumPy arrays

- all elements must be of the same type -- either all real, or all complex -- for efficient numerical storage
- the number of elements must be known at the time the array is created
- Use multidimensional arrays in different ways - 0D/1D arrays are vectors, 2D arrays are matrices, 3 and higher dimensions are tensors

Let me first illustrate the difference between 0D and 1D vectors

In [3]:
```
x = np.array([1.,2.,3.]) # 0D array

y = np.array([[1.,2.,3.]]) # 1 x 3 vector

z = np.array([[1.],[2.],[3.]]) # 3 x 1 vector
```

In [32]:
```
print('x is a 0 D array \n', x)
print('y is a 1 x 3 array \n', y)
print('z is a 3 x 1 array \n', z)
```

```
x is a 0 D array
```

```
 [ 0.          1.42857143  2.85714286  4.28571429  5.7142857
1  7.14285714
  8.57142857 10.        ]
y is a 1 x 3 array
 [ 0.          0.98990308  0.2806294  -0.91034694 -0.5387052
9  0.75762842
  0.75348673 -0.54402111]
z is a 3 x 1 array
 [[1.]
 [2.]
 [3.]]
```

Transposes

Use `A.T` for transpose and `A.H` for conjugate transpose (or `A.conj().T`).

In [5]:
```
print(x.T)  # Still 0D
```

```
[1. 2. 3.]
[[1.]
 [2.]
 [3.]]
[[1. 2. 3.]]
```

In [33]:
```
print(y.T)  # now 3 x 1
```

```
[ 0.          0.98990308  0.2806294  -0.91034694 -0.53870529
0.75762842
  0.75348673 -0.54402111]
```

In [34]:
```
print(z.T)  # now 1 x 3
```

```
[[1. 2. 3.]]
```

Use numpy arrays for matrices

Do not use `numpy.matrix`! Use 2D arrays are matrices.

In [6]:
```
A = np.array([[1.,2.,3.,],[4.,5.,6.]])
print('A is \n', A)
```

```
A is
 [[1. 2. 3.]
 [4. 5. 6.]]
```

There are many ways to create arrays with the desired sizes. Here are three examples. A comprehensive list is given [here (https://docs.scipy.org/doc/numpy/reference/routines.array-creation.html)](https://docs.scipy.org/doc/numpy/reference/routines.array-creation.html).

In [7]:
```
A = np.empty([5,3], dtype = 'd')
B = np.ones([5,3], dtype = 'd')
C = np.zeros([5,3], dtype = 'd')

print(A)
```

```
[[ 3.10503618e+231  4.33366029e-311  3.95252517e-323]
 [ 0.00000000e+000  0.00000000e+000  0.00000000e+000]
 [ 0.00000000e+000  0.00000000e+000  0.00000000e+000]
 [ 0.00000000e+000  0.00000000e+000  0.00000000e+000]
 [ 0.00000000e+000  0.00000000e+000 -1.29073877e-231]]
```

Exercise

1. Write a function to create the $n \times n$ Hilbert matrix

$$H_{ij} = \frac{1}{i+j-1} \qquad i,\ j = 1, \ldots, n.$$

2. Write a function to create the $n \times n$ minij matrix

$$M_{ij} = \min\{i,\ j\} \qquad i,\ j = 1, \ldots, n.$$

generating arrays using random numbers

`numpy.linalg.randn` generates Normal$(0, 1)$ (pseudo)random numbers, and `numpy.linalg.rand` generates Uniform$(0, 1)$ (pseudo)random numbers.

In [9]:
```
A = np.random.randn(5,3)
print('A is \n', A)
B = np.random.rand(3,5)
print('B is \n', B)
```

```
A is 
 [[-0.69468439  0.55392483  0.13927966]
 [-1.57284116 -0.51114557  0.76185854]
 [ 1.0443127  -0.53741394  0.37101879]
 [ 0.30211978 -0.06018174 -0.16704228]
 [ 0.42631502  0.82177933  2.21353461]]
B is 
 [[0.85878783 0.55218438 0.94573258 0.76749066 0.90981874]
 [0.26559624 0.62941045 0.81189038 0.73109523 0.12906472]
 [0.52533622 0.35061167 0.33797694 0.55490002 0.5203647 ]]
```

Shape and size

Arrays have attributes such as

- shape (tuple, dimensions of the array) and
- size (int, total number of elements).

They can be used in different ways.

In [10]:
```
A = np.zeros([5,2,3], dtype = 'd')
print('Shape of A is ', A.shape)
print('Size of A is ', A.size)
```

```
Shape of A is  (5, 2, 3)
Size of A is  30
```

In [11]:
```
x = np.array([1.,2.,3.])
print(x.shape)
```

```
(3,)
```

Indexing and slicing

Indexing and slicing works very similar to lists.

In [12]:
```
v = np.random.randn(10)
print('Array \n', v)
```

```
Array
 [ 1.2371066   0.45921946 -0.86388601 -0.38604107  2.1345066
7 -1.88693045
  1.1071617   1.0935083   0.7989105  -1.03451575]
```

In [13]:
```
print('First five elements \n', v[:5])
```

```
First five elements
 [ 1.2371066   0.45921946 -0.86388601 -0.38604107  2.1345066
7]
```

In [14]:
```
print('Reverse the array \n', v[::-1])
```

```
Reverse the array
 [-1.03451575  0.7989105   1.0935083   1.1071617  -1.8869304
5  2.13450667
 -0.38604107 -0.86388601  0.45921946  1.2371066 ]
```

In [15]:
```
print('Every third element \n', v[::3])
```

```
Every third element
 [ 1.2371066  -0.38604107  1.1071617  -1.03451575]
```

We can slice rows/columns/submatrices. **Warning**: Slicing a row or column from a matrix gives a 0D vector and does not preserve dimension.

In [16]:
```
A = np.random.rand(4,4)
print('First column is \n ', A[:,0])
```

```
First column is
  [0.47911539 0.25844869 0.79423498 0.5480143 ]
```

In [17]:
```
print('First two rows are \n', A[:2,:])
```

```
First two rows are
 [[0.47911539 0.47902364 0.85554381 0.92920329]
 [0.25844869 0.67823745 0.89843828 0.9185015 ]]
```

In [18]:
```
print('A 2 x 2 submatrix is \n', A[::2,::2])
```

```
A 2 x 2 submatrix is
 [[0.47911539 0.85554381]
 [0.79423498 0.86300986]]
```

Exercise:

Create an array w with values $[0, 0.1, 0.2, ..., 3]$. Write out `w[:]`, `w[:-2]`, `w[::5]`, `w[2:-2:6]`. Convince yourself in each case that you understand which elements of the array that are printed.

Hint: Check out `numpy.arange`, `numpy.linspace`.

N-Dimensional Grids

The function `numpy.meshgrid` allows constructing n-dimensional arrays from one-dimensional arrays.

Given two arrays x and y of size m and n respectively, `numpy.meshgrid` returns two matrices X and Y with entries

$$X_{ij} = x_i \qquad Y_{ij} = y_j \qquad i = 1, \ldots, m \quad j = 1, \ldots, m.$$

This is very valuable when plotting two dimensional functions or computing pairwise distances between two sets of points.

In [36]:
```
x = np.arange(3); y = np.arange(3)
X,Y = np.meshgrid(x,y)
print('X is \n', X)
print('Y is \n', Y)
```

```
X is 
 [[0 1 2]
 [0 1 2]
 [0 1 2]]
Y is 
 [[0 0 0]
 [1 1 1]
 [2 2 2]]
```

Example:

Consider the Hilbert matrix as before

$$H_{ij} = \frac{1}{i+j-1} \qquad i,\ j = 1, \ldots, n.$$

We can write this efficiently using meshgrid as

In [39]:
```
i = np.arange(i)
I,J = np.meshgrid(i,i)
H = 1/(I+J+1)
print(H)
```

```
[[1.         0.5        0.33333333 0.25       0.2       ]
 [0.5        0.33333333 0.25       0.2        0.16666667]
 [0.33333333 0.25       0.2        0.16666667 0.14285714]
 [0.25       0.2        0.16666667 0.14285714 0.125     ]
 [0.2        0.16666667 0.14285714 0.125      0.11111111]]
```

Norms

Recall the vector norms for $x \in \mathbb{R}^n$

$$\| x \|_1 = \sum_{j=1}^n |x_i| \qquad \| x \|_2 = \left(\sum_{j=1}^n |x_i|^2\right)^{1/2} \qquad \| x \|_\infty = \max_{1 \leq j \leq n} |x_i|.$$

The function `numpy.linalg.norm` implements both vector and matrix norms.

In [19]:
```
v = np.random.randn(100)

print('1-norm is ', np.linalg.norm(v,1))
print('2-norm is ', np.linalg.norm(v,2))
print('$\infty$-norm is ', np.linalg.norm(v,np.inf))
```

```
1-norm is  85.09625901632997
2-norm is  10.638192304822253
$\infty$-norm is  2.5597811488299076
```

For practice, we will compute these norms by writing for loops

In [20]:
```
onenorm = 0.
twonorm = 0.
infnorm = 0.

for j in range(v.size):
    onenorm += np.abs(v[j])
    twonorm += v[j]**2.

    infnorm = np.abs(v[j]) if np.abs(v[j]) > infnorm else in
fnorm

twonorm = np.sqrt(twonorm)

print('1-norm is ', onenorm)
print('2-norm is ', twonorm)
print('$\infty$-norm is ', infnorm)
```

```
1-norm is  85.09625901632994
2-norm is  10.638192304822251
$\infty$-norm is  2.5597811488299076
```

Similarly we can compute several matrix norms using the same function.

In [21]:
```
A = np.random.rand(5,3)


print('1-norm is ', np.linalg.norm(A,1))
print('2-norm is ', np.linalg.norm(A,2))
print('$\infty$-norm is ', np.linalg.norm(A,np.inf))
```

```
1-norm is  2.710598278604593
2-norm is  1.8432625173675665
$\infty$-norm is  1.8160594210498942
```

Exercise: Frobenius norm

Exercise: Frobenius norm

Implement the Frobenius norm for three different ways:

Check your answer with `numpy.linalg.norm(A, 'fro')`.

Matrix and vector multiplication

There are two types of vector multiplication

- inner product where
- outer product $xy^\top$ where $x \in \mathbb{R}^m,\ y \in \mathbb{R}^n$

Matrix multiplication: where $A \in \mathbb{R}^{m \times n},\ B \in \mathbb{R}^{n \times p}$ is given by

The relevant functions are `numpy.inner`, `numpy.outer`, and `numpy.dot` or `@`.

Note: `*` is reserved for elementwise or Hadamard product.

In [22]:
```
x = np.random.randn(3); y = np.random.randn(3);
print('Inner product is ', np.inner(x,y))
print('Outer product is \n', np.outer(x,y))
```

```
Inner product is  -1.0432064752335939
Outer product is
 [[ 0.19127714 -0.14023235 -0.44260236]
 [ 0.88774183 -0.65083639 -2.05417453]
 [ 0.25223176 -0.18492044 -0.58364722]]
```

In [23]:
```
A = np.random.rand(3,5); B = np.random.randn(5,2)
print('C = AB \n', np.dot(A,B))
print('C = AB \n', A @ B)
```

```
C = AB
 [[ 0.93782368 -1.75304125]
 [ 0.00437667 -1.012766  ]
 [ 0.59826893 -2.39410928]]
C = AB
 [[ 0.93782368 -1.75304125]
 [ 0.00437667 -1.012766  ]
 [ 0.59826893 -2.39410928]]
```

Factorizations

NumPy/SciPy have wrappers to LAPACK libraries and ship with all the standard factorizations

- LU with pivoting
- QR with/without pivoting
- SVD

Let's start with the LU Factorization

This computes a factorization of the form

$$A = PLU$$

$$A = PLU,$$

where

- P is a permutation matrix
- L is a lower triangular matrix with 1's on the diagonals
- U is an upper triangular matrix

In [24]:
```
from scipy.linalg import lu

# LU Factorization
A = np.random.randn(3,3)
p, l, u = lu(A)


print('The lower triangular factor L is \n', l)
print('\n')
print('The upper triangular factor U is \n', u)
```

```
The lower triangular factor L is 
 [[ 1.          0.          0.        ]
 [ 0.07962559  1.          0.        ]
 [-0.30890423  0.93259988  1.        ]]


The upper triangular factor U is 
 [[ 0.7656521  -0.35229772 -1.22673199]
 [ 0.         -1.89484792 -0.49250165]
 [ 0.          0.         -0.06631888]]
```

Check that the LU factorization is correct

In [25]:
```
print('P * L * U is \n', p @ l @ u )
print('The matrix A is \n', A)
print(np.allclose(p @ l @ u, A))
```

```
P * L * U is 
 [[ 0.0609655  -1.92289983 -0.59018091]
 [-0.23651317 -1.65830868 -0.14668315]
 [ 0.7656521  -0.35229772 -1.22673199]]
The matrix A is 
 [[ 0.0609655  -1.92289983 -0.59018091]
 [-0.23651317 -1.65830868 -0.14668315]
 [ 0.7656521  -0.35229772 -1.22673199]]
True
```

Solving linear systems

In [26]:
```
from numpy.linalg import solve


A = np.random.randn(100,100)
xt = np.ones((100,))              #Create a true solution
b = A @ xt                        #Create a right hand side b
```

```
x = solve(A,b)

print(np.allclose(x,xt))
print('The relative error is ', np.linalg.norm(x-xt)/np.lina
lg.norm(xt))
```

```
True
The relative error is  7.983275570239809e-15
```

Vectorization

Instead of using for loops, we can speed up many calculations using vectorization. The message is that: Avoid for loops when possible, and use numpy's implementations. If you really want to use for loops but it is slow, you can use Cython.

In [27]:
```
a = np.random.randn(10000)
asum = 0.
%time for j in np.arange(a.size): asum += a

%time asum2 = np.sum(a)
```

```
CPU times: user 232 ms, sys: 72.6 ms, total: 304 ms
Wall time: 60.2 ms
CPU times: user 724 µs, sys: 256 µs, total: 980 µs
Wall time: 267 µs
```

In [28]:
```
%timeit asum2 = np.sum(a)
```

```
7.98 µs ± 204 ns per loop (mean ± std. dev. of 7 runs, 100000
loops each)
```

SciPy

Here is a very high level view of `scipy`

- Special functions (scipy.special)
- Integration (scipy.integrate)
- Optimization (scipy.optimize)
- Interpolation (scipy.interpolate)
- Fourier Transforms (scipy.fftpack)
- Signal Processing (scipy.signal)
- Linear Algebra (scipy.linalg)
- Compressed Sparse Graph Routines (scipy.sparse.csgraph)
- Spatial data structures and algorithms (scipy.spatial)
- Statistics (scipy.stats)
- Multidimensional image processing (scipy.ndimage)
- File IO (scipy.io)

I will illustrate some of these capabilities with a couple of examples.

Interpolation

Interpolation

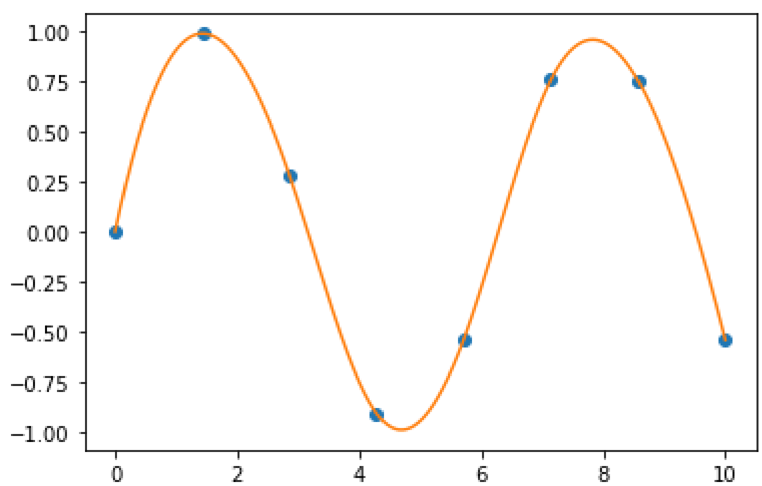
Taken from Jake VDP's book (https://jakevdp.github.io/WhirlwindTourOfPython/15-preview-of-data-science-tools.html).

In [29]:
```
from scipy import interpolate
import matplotlib.pyplot as plt
%matplotlib inline
# choose eight points between 0 and 10
x = np.linspace(0, 10, 8)
y = np.sin(x)

# create a cubic interpolation function
func = interpolate.interp1d(x, y, kind='cubic')

# interpolate on a grid of 1,000 points
x_interp = np.linspace(0, 10, 1000)
y_interp = func(x_interp)

# plot the results
plt.figure()  # new figure
plt.plot(x, y, 'o')
plt.plot(x_interp, y_interp);
```



ODE integration