Mc Graw Hill

Because learning changes everything.®

TITLE: Python Programming: Problem Solving, Packages and Libraries

Edition

Lecture PPT Chapter 18: **NumPy, SciPy**

Chapter 18 NumPy, SciPy

LEARNING OBJECTIVES

After studying this chapter, you will be able to:

- **LO 1** Know that ndarray is the basic data structure in NumPy and understand the concept of axis of an ndarray object.
- **LO 2** Understand broadcasting and indexing in NumPy arrays.
- **LO 3** Use the np.linspace() and np.meshgrid() methods.
- **LO 4** Use the **linalg** package for linear algebra for solving system of linear equations.
- **LO 1** Realize that vectors can be thought of as a column matrix, i.e., a matrix with n rows but only 1 column, and various matrix operations which can transform a vector.
- **LO 5** Explore the concepts of eigenvector and eigenvalue.
- **LO 6** Solve an eigen decomposition problem manually and then use the numpy.linalg package to solve the same problem.
- **LO 7** Get insight into the mathematical concepts behind SVD (Singular Value Decomposition).

N-dimensional array in NumPy

The main data structure defined and used in NumPy module is called ndarray. The term ndarray stands for N-dimensional array. An object of type ndarray represents a multi-dimensional array of data.

The use of NumPy will become clear from the following script:

```
1 import numpy as np
2 A = [ n * n for n in range(20)]
3 B = np.array(A)
4 print(B)
5 C = B * 2 #You cant do this in Python but you can in numpy
6 print(C)
7 import matplotlib.pyplot as plt
8 plt.plot(C)
9 plt.show()
```

The above script is explained in details in the book

Some of the common methods in NumPy --1

1. `numpy.array(alist)`, where alist is a python list: It is a function to create a NumPy array which is of type ndarray. Note that in pure Python there are only lists and there is no data type called ndarray. But NumPy has a large number of its own data types. Note that numpy.array() is a method and the object created by this method is ndarray
2. `numpy.zeros(N, M, dtype = float)`: This constructs an n-dimensional array of the specified shape, i.e., N □ M, filled with zeros of the specified dtype.
3. `numpy.ones(N, M, dtype=float)`: This constructs an 2-dimensional array of the specified shape, i.e., N □ M, filled with ones of the specified dtype.

Some of the common methods in NumPy --2

4. `numpy.eye(N, M = None, dtype=float)`: This returns a two-dimensional (2D) array which has number 1 on the diagonal elements and number 0 everywhere else. By default the data type, i.e., the 0s and the 1s are floats. Note if value of M is not specified, it will be same as N and the resulting matrix will be a square matrix. But if M is different from N then it will not be a square matrix.

5. `numpy.transpose(a, axes=None)`: This method permutes the dimensions of the given array, i.e., "a". Note that the parameter a is an array, whose transpose is to be created. You have the option of not specifying any value for axis. If no value is given, then the default value for axes is None and the dimensions get reversed. The other options for axis are not discussed here. You can always look up the online documentation for details. However, note that the axes are passed as a tuple of integers starting from 0 since the 1st axes will have a dimension number of 0. In its simplest form, if you create the transpose of a 2D matrix say of type 2 ⍰ 5, then its transpose will be a matrix of type 5 ⍰ 2.

Some of the common methods in NumPy --3

6. `numpy.arange(start, stop, increment)`: NumPy method arrange() is like the Python range function but with one important difference. In the arrange method, the values for start, stop and step can be real values (not just integers like in ordinary Python), i.e., can be decimals and fractions also.

7. `numpy.random.random((N, M))`: This will create a NumPy array N × M filled with random numbers in range [0,1). Note 1 is not generated but 0 can be generated.

8. `numpy.random.randint(X, Y, (N, M))`: Here X and Y must be integers and so random integers will be generated between [X, Y), i.e., from X to Y but not including Y. N × M are the dimensions of the array created.

Given below are the screen shots of Jupyter notebook of the common methods of numpy discussed before.

(The screen shots given below are from a Jupyter workbook which can also be downloaded)

Some of the common methods in NumPy

numpy.array(alist), where alist is a python list:

- It is a function to create a NumPy array which is of type ndarray.
- Note that in pure Python there are only lists and there is no data type called ndarray.
- But NumPy has a large number of its own data types.
- Note that numpy.array() is a method and the object created by this method is ndarray

In [1]:
```
import numpy as np
A = np.array([1, 2, 3, 4]) # Create numpy array from python list
print('A ->', A)
```

```
A -> [1 2 3 4]
```

Method numpy.ones()

numpy.ones(N, M, dtype=float)

- This constructs an 2-dimensional array of the specified shape,
- i.e., N x M, filled with ones of the specified dtype.
- numpy.ones(N, M, dtype=float):

```
In [8]: D = np.ones((2, 2), dtype = complex) # dtype is complex
        print('D->', D)
```

```
D-> [[1.+0.j 1.+0.j]
 [1.+0.j 1.+0.j]]
```

Method numpy.eye()

numpy.eye(N, M = None, dtype=float):

- This returns a two-dimensional (2D) array which has number 1 on the diagonal elements and number 0 everywhere else.
- By default the data type, i.e., the 0s and the 1s are floats.
- Note if value of M is not specified, it will be same as N and the resulting matrix will be a square matrix.
- But if M is different from N then it will not be a square matrix.

In [3]:
```
E = np.eye(4, dtype = int)
print('E->',E)
```

```
E-> [[1 0 0 0]
 [0 1 0 0]
 [0 0 1 0]
 [0 0 0 1]]
```

Method numpy.transpose()

numpy.transpose(a, axes=None):

- This method permutes the dimensions of the given array, i.e., "a".
- Note that the parameter a is an array, whose transpose is to be created.
- You have the option of not specifying any value for axis.
- If no value is given, then the default value for axes is None and the dimensions get reversed.
- The other options for axis are not discussed here.
- You can always look up the online documentation for details.
- However, note that the axes are passed as a tuple of integers starting from 0 since the 1st axes will have a dimension number of 0.
- In its simplest form, if you create the transpose of a 2D matrix say of type 2 x 5, then its transpose will be a matrix of type 5 x 2.

In [4]:
```
F = np.array([[1,2], [3,4], [5,6], [7,8]]) # F is 4 x 2 array
print('F->', F)
G = np.transpose(F) # G is 2 x 4 array
print('G->', G)
```

```
F-> [[1 2]
 [3 4]
 [5 6]
 [7 8]]
G-> [[1 3 5 7]
 [2 4 6 8]]
```

Method numpy.arrange()

numpy.arange(start, stop, increment):

- NumPy method arrange() is like the Python range function but with one important difference.
- In the arrange method, the values for start, stop and step can be real values (not just integers like in ordinary Python), i.e., can be decimals and fractions also.

In [5]:
```
H = np.arange(2.4, 3.4, 0.2) # start, stop and step are decimals.
print('H->', H)
```

```
H-> [2.4 2.6 2.8 3.  3.2]
```

numpy.random.random((N, M)):

- This will create a NumPy array N □ M filled with random numbers in range [0,1).
- Note 1 is not generated but 0 can be generated.

Method numpy.random.random()

numpy.random.random((N, M)):

- This will create a NumPy array N □ M filled with random numbers in range [0,1).
- Note 1 is not generated but 0 can be generated.

In [6]:
```
I = np.random.random((2, 2)) # Create 2 x 2 matrix of randoms in range [0,1)
print('I->', I)
```

```
I-> [[0.41995364 0.33401088]
 [0.75556383 0.3552693 ]]
```

Method numpy.random.randint()

numpy.random.randint(X, Y, (N, M)):

- Here X and Y must be integers and so random integers will be generated between [X, Y), i.e., from X to Y but not including Y.
- N x M are the dimensions of the array created.

In [7]:
```
J = np.random.randint(2, 4, (2, 3)) # Will generate random number 2,3 but not 4
print('J->', J)
```

```
J-> [[3 2 3]
 [2 2 2]]
```

Common numpy methods discussed above are given at one place in the script below

```
1   import numpy as np
2   A = np.array([1, 2, 3, 4]) # Create numpy array from python list
3   print('A ->', A)
4   B = np.zeros((2, 2)) # Data type defaults to float
5   print('B->', B)
6   C = np.zeros((2,2), dtype = int) # Data type is int, so no decimal after 0
7   print('C->', C)
8   D = np.ones((2, 2), dtype = complex) # dtype is complex
9   print('D->', D)
10  E = np.eye(4, dtype = int)
11  print('E->',E)
12  F = np.array([[1,2], [3,4], [5,6], [7,8]]) # F is 4 x 2 array
13  print('F->', F)
14  G = np.transpose(F) # G is 2 x 4 array
15  print('G->', G)
16  H = np.arange(2.4, 3.4, 0.2) # start, stop and step are decimals.
17  print('H->', H)
18  I = np.random.random((2, 2)) # Create 2 x 2 matrix of randoms in range [0,1)
19  print('I->', I)
20  J = np.random.randint(2, 4, (2, 3)) # Will generate random number 2,3 but not 4
21  print('J->', J)
```

The output from this script is shown on next slide

Output of script from previous page

```
# Output is
A -> [1234]
B-> [[ 0.0.]
 [ 0.0.]]
C-> [[00]
 [00]]
D-> [[ 1.+0.j  1.+0.j]
 [ 1.+0.j  1.+0.j]]
E-> [[1000]
 [0100]
 [0010]
 [0001]]
F-> [[12]
 [34]
 [56]
 [78]]
G-> [[1357]
 [2468]]
H-> [ 2.42.62.83.2]
I->[[ 0.30554412  0.98019027]
     [ 0.11167242  0.96397391]]
J-> [[232]
 [232]]
```

Concept of dimensions in a Numpy array

The concept of dimensions of an array in NumPy will become clear from the following example. As an example, take a 2 × 3 × 4 array. It will have a total of 24 items in it. Fill it up with integers from 0 to 23. The code is as follows:

```
1   import numpy as np
2   A = np.arange(24).reshape(2,3,4)
3   print(A)
4   # Output
5   [[[ 0  1  2  3]
6     [ 4  5  6  7]
7     [ 8  9 10 11]]
8
9    [[12 13 14 15]
10    [16 17 18 19]
11    [20 21 22 23]]]
```

You can think of (1) Dimension 0 as sheet; (2) Dimension 1 as row; and (3) Dimension 2 as column.

The next slide shows a geometric representation of this 3-D array

Concept of dimensions in a Numpy array (Geometric representation of a 3-D ndarray)

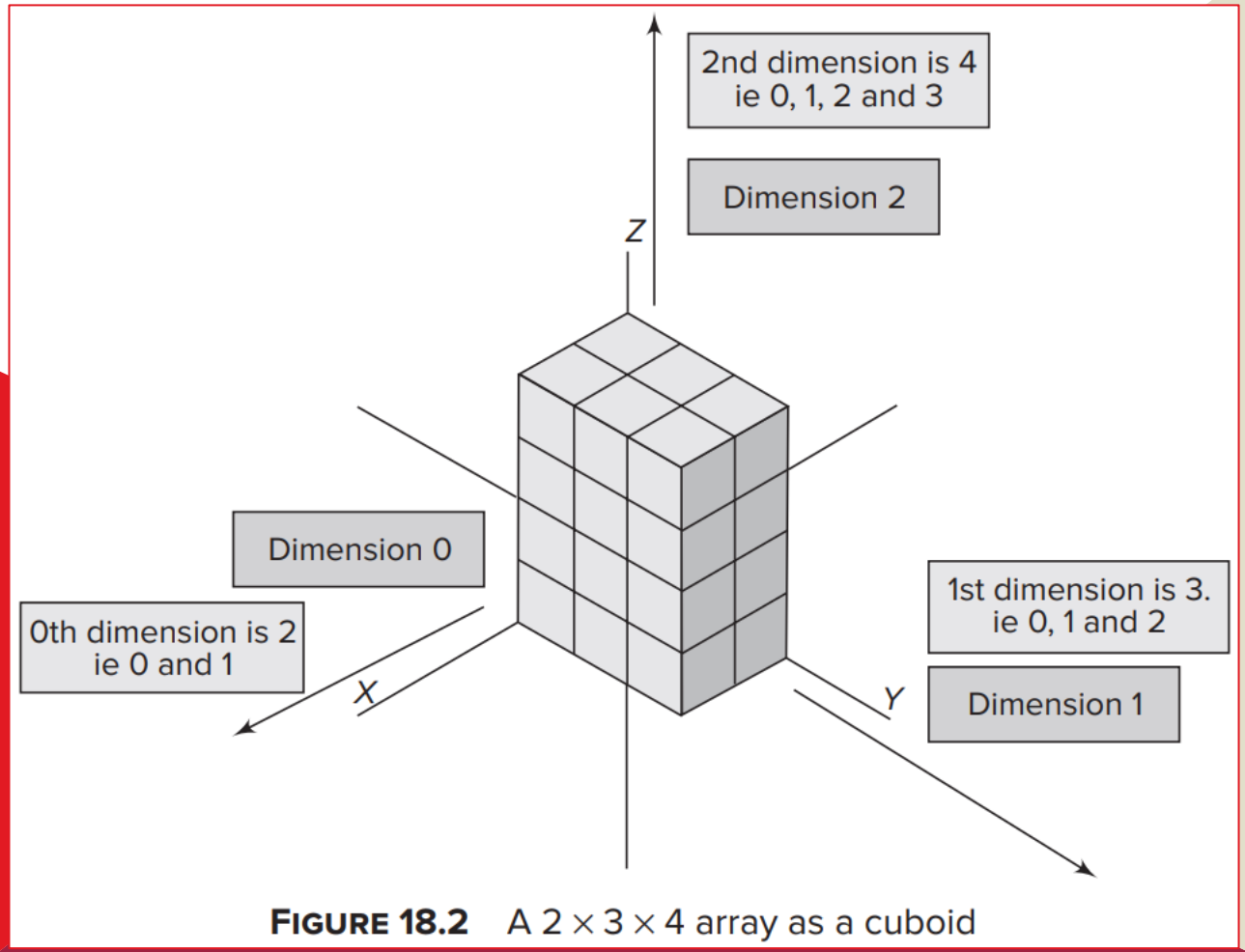


FIGURE 18.2 A $2 \times 3 \times 4$ array as a cuboid

Element-wise operations on arrays

In the NumPy package:-

- **a number of methods are provided for doing operations on the individual elements of the array.**
- **For example, numpy.sum(a) will add all the elements of an array and give the result.**

```
1  import numpy as np
2  A = np.arange(12).reshape(3, 4) # Create 3 x 4 array
3  print("A=",A)
4  B = A.sum(axis = 0) # Add along axis 0
5  print("B=",B)
6  C = A.sum(axis = 1) # Add along axis 1
7  print("C=", C)
8  # Output
9  A= [[ 0123]
10  [ 4567]
11  [ 891011]]
12 B= [12151821]
13 C= [ 62238]
```

Figure showing:- Element-wise operations on a numpy array

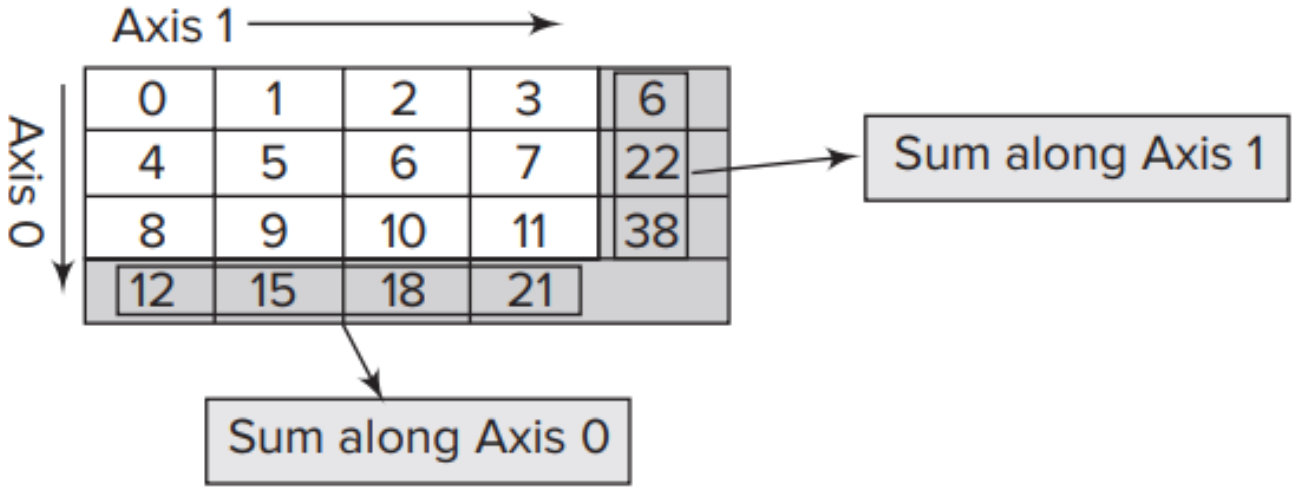


FIGURE 18.3 Adding all the elements of an array along axis 0 and axis 1

Broadcasting in NumPy array operations

- In NumPy array operations, the term broadcasting comes into play when you do arithmetic operations on arrays with different shapes.
- In normal Python arrays like lists, mathematical operations are done on arrays of same shape.
- However, in NumPy, under certain circumstances, it is possible to do some mathematical operations on arrays of different lengths. The way this is done is to broadcast the smaller array into an array of larger dimension.

Python arrays (Lists) and Numpy arrays behave differently

However considering broadcasting, you need to understand that Python arrays and NumPy arrays behave differently.

The following example shows the difference in behavior of Python array (i.e. list) and NumPy array:

```
1  import numpy as np
2  # In Python
3  myL = [1, 2, 3, 4]
4  expand = 2
5  expandedL = myL * 2
6  print('in python->', expandedL)
7  # In numpy
8  a = np.array(myL)
9  expanded_a = a * expand
10 print('in numpy->', expanded_a)
11 # OUTPUT
12 in python-> [1, 2, 3, 4, 1, 2, 3, 4]
13 in numpy-> [2 4 6 8]
```

Certain other aspects of broadcasting (Not covered in details in the book)

The following aspects of broadcasting are relevant:

- **NumPy operations are usually done on 2 ndarrays on an element-by-element basis.**
- **This means that suppose you have 2 ndarray say A and B of dimensions 2 x 3 each, then A + B would lead to the corresponding elements being added.**
- **However NumPy also permits mathematical operations on arrays of "different dimensions", with certain restrictions.**
- **This means that it is possible to do mathematical operations on ndarray of different dimensions provided they obey certain restrictions.**
- **So subject to certain constraints, the smaller array is "broadcast" across the larger array so that they have compatible shapes.**

Broadcasting of a “scalar” -- 1

- **The simplest broadcasting is when an array and a “scalar” are combined in an operation.**
- **A “scalar” here means simply a number.**
- **The scalar is “converted” into an array of same dimension as the array on which it is being operated and each element in this “converted array” is a copy of the scalar.**

Broadcasting of a “scalar” – 2 (Example)

```
1  import numpy as np
2  A = np.arange(12).reshape(3, 4) # Create 3 x 4 array
3  print(A)
4  scalar = 5
5  print(A * scalar)
6  # Output
7  [[ 0  1  2  3]
8   [ 4  5  6  7]
9   [ 8  9 10 11]]
10 [[ 0  5 10 15]
11  [20 25 30 35]
12  [40 45 50 55]]
```

The rules for broadcasting -- 1

1. **The two arrays have exactly the same shape. (Here of course no broadcasting is required)**
2. **The two arrays have the same number of dimensions but the sizes of some or all of the dimensions are different.**
   A. For example you have 2 arrays of dimensions 2 x 3 x 4 and 2 x 3 x 1. These 2 arrays both have 3 dimensions out of which the first 2 dimensions are of same size but the third dimension is of different size.
   B. The rule for broadcasting here is that the "uncommon" dimensions should be 1. So for example if you have two 3-D arrays say A → 2 x 3 x 4 and B → 2 x 3 x 1, then array B can be broadcast into 2 x 3 x 4 array because its uncommon dimension has a size of 1.

[Continued on next slide . . . .]

The rules for broadcasting -- 2

[… Continued from previous slide]

3. The 2 arrays have different dimensions. It is possible to broadcast an array of smaller dimension into an array of bigger dimension by "Lining up the sizes of the trailing axes of these arrays according to the broadcast rules". This appears a bit confusing but can be easily understood with examples.

A. Suppose you have array A → 2 x 3 x 4 and array B → 3 x 4. Here the "trailing dimensions" of A and B are matched and both are 3 x 4. So array B is broadcast into an array of 2 x 3 x 4, so as to match the 2 arrays. In this example the "trailing dimensions" matched. However they need not match.

B. Trailing dimensions can be broadcast even if one of the dimensions is 1. So if you had A → 2 x 3 x 4 and B → 1 x 4, then also B can be broadcast into 2 x 3 x 4. This can be done because the "non-matching" dimension of B is 1.

C. Note that the non-matching dimension can be in either of the 2 arrays. So if you have A → 2 x 1 x 4 and B → 3 x 4, then A will be broadcast to A → 2 x 3 x 4 and B will be broadcast to B → 2 x 3 x 4. Here The size of one of the dimensions of A has been increased from 1 to 3, while the number of dimensions of B have been increased from two to three.

Broadcasting Example

The 2 arrays are of different dimensions but the “size” of trailing dimensions are same

```
1   import numpy as np
2   # Trailing dimensions of A and B are same i.e 3 x 4
3   A = np.arange(24).reshape(2, 3, 4) # Create 2 x 3 x 4 array
4   B = np.arange(12).reshape(3, 4)  # Create a 3 x 4 array
5
6   print('A->', A)
7   print('B->', B)
8   print('A * B->', A * B)
```

```
9   # Output
10  A-> [[[ 0  1  2  3]
11    [ 4  5  6  7]
12    [ 8  9 10 11]]
13
14   [[12 13 14 15]
15    [16 17 18 19]
16    [20 21 22 23]]]
17  B-> [[ 0  1  2  3]
18   [ 4  5  6  7]
19   [ 8  9 10 11]]
20  A * B-> [[[  0   1   4   9]
21    [ 16  25  36  49]
22    [ 64  81 100 121]]
23
24   [[  0  13  28  45]
25    [ 64  85 108 133]
26    [160 189 220 253]]]
```

Broadcasting another example

The two arrays are of different dimensions and also of different sizes. But the dimension which varies on size has a size of 1.

```
1   import numpy as np
2   A = np.arange(8).reshape(2, 1, 4) # Create 2 x 1 x 4 array
3   B = np.arange(12).reshape(3, 4)  # Create a 3 x 4 array
4   # A will be broadcast from (2 x 1 x 4) to (2 x 3 x 4)
5   # B will be broadcast from (3 x 4) to (2 x 3 x 4)
6   print('A->', A)
7   print('B->', B)
8   print('A * B->', A * B)
```

```
9   # Output
10  A-> [[[0 1 2 3]]
11
12   [[4 5 6 7]]]
13  B-> [[ 0  1  2  3]
14   [ 4  5  6  7]
15   [ 8  9 10 11]]
16  A * B-> [[[ 0  1  4  9]
17    [ 0  5 12 21]
18    [ 0  9 20 33]]
19
20   [[ 0  5 12 21]
21    [16 25 36 49]
22    [32 45 60 77]]]
```

Numpy constants
(Infinity, negative infinity, zero, negative zero and some other constants in NumPy)

TABLE 18.1 Some important constants in NumPy

Sl. No.	Constant	Description	S. No.	Constant	Description
1	NINF	Negative infinity	5	e	Euler's constant, or Napier's constant
2	Inf or PINF	Positive infinity	6	pi	Π
3	NZERO	Negative zero	7	Euler_gamma	Also called the Euler–Mascheroni constant
4	PZERO	Positive zero			

[Examples showing use of these Numpy constants are shown on next slide]

Numpy constants – Example script

```
1  import numpy as np
2  print('Negative infinity->', np.NINF)
3  print('Positive infinity->', np.PINF)
4  print('Negative Zero->', np.NZERO)
5  print('Positive Zero->', np.PZERO)
6  print('Eulers constant->', np.e)
7  print('Pi->', np.pi)
8  print('euler_gamma->', np.euler_gamma)
9  print('log 0 is infinity->', np.log(0) )
10 print('log -1 is nan->', np.log(-1) )
```

```
11 # OUTPUT
12 Negative infinity-> -inf
13 Positive infinity-> inf
14 Negative Zero-> -0.0
15 Positive Zero-> 0.0
16 Eulers constant-> 2.718281828459045
17 Pi-> 3.141592653589793
18 euler_gamma-> 0.5772156649015329
19 log 0 is infinity-> -inf
20 log -1 is nan-> nan
```

np.linspace

- NumPy has a method linspace. This method is very useful in getting evenly spaced numbers over an interval. Meaning of evenly spaced numbers:
- Suppose you have an interval say [2, 4].
- Suppose you want to divide this interval into 6 intervals then your list would be: [2. 2.4 2.8 3.2 3.6 4.]. So this is the case where you divided an interval with start = 2, stop = 4 and interval = 6. Note here you also included the endpoint to the right, i.e., 4; so you did endpoint = True.
- Now normally in many Python functions such as range(), the end point to the right is excluded. (For example, range(10) produces numbers from 0 to 9.) You could do this with the linspace method also by having endpoint = False.
- The signature of linspace method is:

```
1  numpy.linspace(start, stop, interval, endpoint=True)
```

np.linspace -- Example

```
1  import numpy as np
2  x1 = np.linspace(2, 4, 6, endpoint = True)
3  print(x1)
4  x2 = np.linspace(2, 4, 5, endpoint = False) # Have to reduce
5  endpoint by 1 to get similar intervals
6  print(x2)
7  # OUTPUT
8  [2. 2.4 2.8 3.2 3.6 4. ]
9  [2. 2.4 2.8 3.2 3.6]
```

Line 4: Here the endpoint is False. Also note that the value of stop parameter has been reduced from 6 to 5.

Use of linspace() method: linspace() method is very useful in plotting of functions. Suppose you want to plot a function say sin(x) between x = 0° to x = 360°. You could divide the interval [0°, 360°] into say 10 intervals and then get values for sin(x) for each of these values of x and plot them.

Understanding np.meshgrid()

- To be able to do 3D plotting in Matplotlib, you first need to understand the concept of np.meshgrid() method.
- To plot a function in 3D say x, y and z, you could have z = f(x,y) where f(x,y) is some function of x and y.
- Now both x and y will have some domain and z will have a corresponding range.
- Suppose the domain of x was integers from 1 to 4. So x ϵ (1, 2, 3, 4). Further, let y ϵ (7, 8, 9). So some of the possible inputs to f(x,y) could be f(1, 7), f(1, 8) and so on.
- Suppose you wanted all possible combinations of x and y in form of a grid. Suppose you represent x, y and (x,y) as shown in Figure 18.4.

$$x = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix}$$

$$y = \begin{bmatrix} 7 & 7 & 7 & 7 \\ 8 & 8 & 8 & 8 \\ 9 & 9 & 9 & 9 \end{bmatrix}$$

$$\text{Grid for input to } f(x, y) = \begin{bmatrix} (1,7) & (2,7) & (3,7) & (4,7) \\ (1,8) & (2,8) & (3,8) & (4,8) \\ (1,9) & (2,9) & (3,9) & (4,9) \end{bmatrix}$$

FIGURE 18.4 meshgrid formed by x ∈ (1, 2, 3, 4) and y ∈ (7, 8, 9)

np.meshgrid() Example

```
1   import numpy as np
2   X = np.linspace(1, 4, 4) #X-> [1. 2. 3. 4.]
3   Y = np.linspace(7, 9, 3) #Y-> [7. 8. 9.]
4   Xgrid, Ygrid = np.meshgrid(X, Y)
5   print('X->', X,'Y->', Y)
6   print('Xgrid->', Xgrid)
7   print('Ygrid->', Ygrid)
8   #Output. . .
9   X-> [1. 2. 3. 4.] Y-> [7. 8. 9.]
10  Xgrid-> [[1. 2. 3. 4.]
11   [1. 2. 3. 4.]
12   [1. 2. 3. 4.]]
13  Ygrid-> [[7. 7. 7. 7.]
14   [8. 8. 8. 8.]
15   [9. 9. 9. 9.]]
```

Using NumPy, SciPy for getting some basic information about a matrix

- The SciPy library has a package called linalg for linear algebra.
- NumPy also has a linalg package, but it is preferable to use SciPy.
- NumPy and SciPy have a number of built-in functions to get some basic characteristics of a matrix.
- For example, you can get characteristics like trace, rank, conjugate, norm, transpose and conjugate transpose.
- Table 18.2 (On next slide) shows these operations.

Table 18.2: Some important methods of the linalg package of NumPy

	Functions/ Methods	Purpose
1	np.trace(A)	Gives trace of A., i.e., sum of all its diagonal elements
2	linalg.inv(A)	Gives inverse. The inverse of a matrix has the property that if it is multiplied by the original matrix, you get an identity matrix
3	A.T	Transpose of matrix
4	A.H	Conjugate transpose
5	linalg.det(A)	Determinant of a matrix
6	linalg.matrix_norm(A)	Gives Frobenius norm (Default)
7	linalg.norm(A, 1)	Gives L_1 norm
8	np.linalg.matrix_rank(A)	Rank of matrix. Note function matrix_rank() is available in linalg module of NumPy and not in the linalg module of SciPy

Examples of some important methods of the linalg package of NumPy

```
from scipy import linalg
import numpy as np
np.set_printoptions(precision=2, suppress = True)
A = array([[0, -3, -2],
           [1, -4, -2],
           [-3, 4, 1]])
B = A.T
print('Transpose of A->', B)
C = linalg.inv(A)
print('Inverse of A->', C)
D = linalg.det(A)
print('Determinant of A->', D)
E = linalg.norm(A)
print('Frobenius norm of A->', E)
F = linalg.norm(A)
print('L1 norm of A->', F)
# Note that linalg is available both in numpy and scipy
# Method matrix_rank() is available only in numpy
G = np.linalg.matrix_rank(A)
print('Rank of A->', G)
```

```
# Output
Transpose of A-> [[ 0  1 -3]
 [-3 -4  4]
 [-2 -2  1]]
Inverse of A-> [[ 4. -5. -2.]
 [ 5. -6. -2.]
 [-8.  9.  3.]]
Determinant of A-> 0.9999999999999991
Frobenius norm of A-> 7.745966692414834
L1 norm of A-> 7.745966692414834
Rank of A-> 3
```

Solving linear system of equations

- A general system of linear equations can be represented in the form of three matrices.
- These matrices can be called (1) coefficient matrix (represented here as A) (2) variable matrix (represented here as vector x) and (3) constant matrix (represented here as b).
- This is shown in Figure below.

$a_{11}\ x_1 + a_{12}\ x_2 + a_{13}\ x_3 = b_1$ $a_{21}\ x_1 + a_{22}\ x_2 + a_{23}\ x_3 = b_2$ $a_{31}\ x_1 + a_{32}\ x_2 + a_{33}\ x_3 = b_3$	$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}$	A.x = b

Example:- Solving linear system of equations

You can solve the following system of linear equations using the linalg module of NumPy. (1) 2x + 3y + z = 13, (2) x – y + 2z = 7 and (3) 3x + 4y + z = 22

```
1  from scipy import linalg
2  import numpy as np
3  np.set_printoptions(precision=2, suppress = True)
4  #2x + 3y + z = 13, x -  y + 2z = 7, 3x + 4y + z = 22
5  A = np.array([[2, 3, 1],
6                [1, -1, 2],
7                [3, 4, 1]])
8  b = np.array([13, 7, 22])
9  # Use np.linalg.solve(A, b)
10 # Return is an array of same shape as b
11 # Here b is a 1-D array so return will be a 1-D array
12 x = np.linalg.solve(A, b)
13 print('x->', x)
14 # Check solution ie A.x
15 B = np.dot(A, x)
16 print('b->', B)
17 print('Is A.x == b?', np.allclose(B, b))
18 # Output
19 x-> [11. -2. -3.]
20 b-> [13.  7. 22.]
21 Is A.x == b? True
```

Multiplying a matrix by a vector –1

- **The following discussion uses the term matrix as in algebra and not as in NumPy.**
- **Note that a 1D array can be thought of as a vector. So**
- $v1 = \begin{pmatrix} x_1 \\ x_2 \\ .. \\ .. \\ x_n \end{pmatrix}$ **is a 1D array and can also be represented (not in python) but in maths as:** $v1 = (x_1 + x_2 + \cdots + x_n)^T$**. Here the post-script represents a transpose and this is an alternate convenient way of representing a vector.**
- **Multiplication of an n × n matrix with a vector (which is also a matrix of dimension n × 1) leads to transformation of the original vector and a new vector is created. This is shown as follows**

[- - - - -On next slide]:

-

Multiplying a matrix by a vector –2

[Continued from previous slide - - - - -]

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ .. \\ .. \\ x_n \end{pmatrix} = \begin{pmatrix} a_{11}x_1 + a_{21}x_2 + \cdots + a_{n1}x_n \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \\ .. \\ .. \\ a_n x_1 + a_n x_2 + \cdots + a_n x_n \end{pmatrix} = \begin{pmatrix} x'_1 \\ x'_2 \\ .. \\ .. \\ x'_n \end{pmatrix}$$

where $x'_1 = a_1x_1 + a_2x_2 + \cdots + a_nx_n$ **and so on. So you get a new vector which you may call v2 and is shown as follows:**

$v2 = \begin{pmatrix} x'_1 \\ x'_2 \\ .. \\ .. \\ x'_n \end{pmatrix}$, **Here also you can represent this vector as**

$v2 = (x'_1 + x'_2 + \cdots + x'_n)^T$

The point to be noted is that multiplication of a vector by a matrix leads to transformation of a vector.

Multiplication of a diagonal matrix to a vector (Scaling) -- 1

- Take a 2D vector say x = $\begin{pmatrix}1\\2\end{pmatrix}$ and multiply it by say a diagonal matrix M = $\begin{pmatrix}3 & 0\\0 & 3\end{pmatrix}$ so that you get a new vector y, such that y = Mx. So you get y = $\begin{pmatrix}3 & 0\\0 & 3\end{pmatrix}\begin{pmatrix}1\\2\end{pmatrix}$ = $\begin{pmatrix}3\\6\end{pmatrix}$. Notice that M is a diagonal matrix with each diagonal element = 3. So what this matrix multiplication has done is in fact increased or stretched the vector $\begin{pmatrix}1\\2\end{pmatrix}$ by a factor of 3 to $\begin{pmatrix}3\\6\end{pmatrix}$. But suppose you have the vector M = $\begin{pmatrix}3 & 0\\0 & 5\end{pmatrix}$, then you will get y = $\begin{pmatrix}3 & 0\\0 & 5\end{pmatrix}\begin{pmatrix}1\\2\end{pmatrix}$ = $\begin{pmatrix}3\\10\end{pmatrix}$. Here the first component of vector got stretched by a factor of 3 while the second component of the vector got stretched by a factor of 5. So the scaling is not uniform.
- So the conclusion is: If you multiply a vector by a diagonal matrix, then this multiplication stretches the components of the vector. So you can call such a matrix a scalar matrix also, since it only scales or stretches the components. In general, if you have:

[- - - - Continued on next slide]

Multiplication of a diagonal matrix to a vector (Scaling) -- 2

[Continued from previous slide - - -]

$$M = \begin{pmatrix} \lambda_1 & 0 & 0 & .. & .. \\ 0 & \lambda_2 & 0 & .. & .. \\ 0 & 0 & \lambda_3 & .. & .. \\ .. & .. & .. & .. & .. \\ .. & .. & .. & .. & \lambda_n \end{pmatrix} \text{ and } x = \begin{pmatrix} x_1 \\ x_2 \\ .. \\ .. \\ x_n \end{pmatrix} \text{ then}$$

$$y = \begin{pmatrix} \lambda_1 & 0 & 0 & .. & .. \\ 0 & \lambda_2 & 0 & .. & .. \\ 0 & 0 & \lambda_3 & .. & .. \\ .. & .. & .. & .. & .. \\ .. & .. & .. & .. & \lambda_n \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ .. \\ .. \\ x_n \end{pmatrix} = \begin{pmatrix} \lambda_1 . x_1 \\ \lambda_2 . x_2 \\ .. \\ .. \\ \lambda_n . x_n \end{pmatrix}$$

So when you multiply a vector by a diagonal matrix, you are stretching/ (or squashing) the individual components of the matrix.

Matrix multiplication as a reflection

- Consider a 2D system and a point say (1, 2).
- Then the reflection of this point on the x-axis will be (1, -2) and its reflection on y-axis will be(-1, 2).
- Also its reflection about the origin will be (-1, -2)
- Table 18.3 shows the matrices for various reflections in 2D.

TABLE 18.3 Two-dimensional matrices for reflection

Reflection in the x-axis	Reflection in the y-axis	Reflection in the origin	Reflection in line y = x
$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$	$\begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}$	$\begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix}$	$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$

Matrix multiplication as rotation

- Now consider another matrix M = $\begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$.
- What this matrix multiplication does is to rotate the vector counter clockwise by an angle θ.
- So if you rotate a vector x = $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ by an angle θ, then its new coordinates are: x' = $\begin{pmatrix} \cos\theta \\ \sin\theta \end{pmatrix}$.

Problem: Show that the matrix for clockwise rotation by angle θ is: $\begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix}$.

Eigenvalues and eigenvectors -- 1

1. In linear algebra, you can think of a square matrix as a linear transformation.
2. To understand the concept of eigenvalue, you can take a square matrix A of dimension→ n × n.
3. Further let $\vec{v}$ be a vector also of n × 1 dimension (i.e., v is a column matrix of n rows and 1 column).
4. Suppose you do a matrix multiplication of A and $\vec{v}$ such that $\vec{w}$ = A$\vec{v}$.
5. Now $\vec{w}$ will also be a column vector of dimension n × 1, i.e., n rows and 1 column.
6. So the effect of multiplication A$\vec{v}$ is that a new vector that $\vec{w}$ has been created.

[- - - Continued on next slide]

Eigenvalues and eigenvectors -- 2

[- - - Continued from previous slide]

7. But now suppose that for the given transformation matrix A, the vector w (i.e., the newly created vector) is such that $\vec{w} = \lambda\vec{v}$ (where λ is a scalar).
8. What does this mean? This means that vector $\vec{v}$ and $\vec{w}$ may have different magnitudes but have same direction. So the newly created or transformed vector has same orientation as the original vector.
9. So you can think of an eigenvector for a square matrix transformation as that vector which under the transformation changes in magnitude but does not change in direction.
10. So you have $A\vec{v} = \lambda\vec{v} = \vec{w}$. Then $\vec{v}$ is the eigenvector and λ is the eigenvalue for the square matrix A.

Eigenvalues and eigenvectors -- 3

This will become clear from the following example:

- **Suppose you have a 2 × 2 matrix A =** $\begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$ **and you have a vector v =** $\begin{bmatrix} 1 \\ -1 \end{bmatrix}$**, then you get**
- **Av =** $\begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \end{bmatrix} = \begin{bmatrix} 2-1 \\ 1-2 \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$.
- **So you get A$\vec{v}$ = $\vec{v}$. So you can say that the vector v is an eigenvector for matrix A. Further here λ = 1. So the eigenvalue for matrix A is 1.**
- **Further you can write the equation A$\vec{v}$ = λ$\vec{v}$ as A$\vec{v}$ – λ$\vec{v}$ = 0, or |A – λI|$\vec{v}$ = 0 where I is the identity matrix. So an equation of form |A – λI|$\vec{v}$ = 0 has a non-zero solution $\vec{v}$ if determinant of |A – λI|$\vec{v}$ = 0.**

Eigenvalues and eigenvectors – 4
(Intuitive meaning of eigen values and eigen vectors)

Intuitive meaning of eigenvalus and eigenvectors

(a) Think of a globe. Suppose you rotate it on its axis, by a small angle θ. Then the coordinates, i.e., the position vector of every point on the globe will change except the points lying on the axis. So you can think of the axis of rotation as an eigenvector for the transformation of rotation. So the vector representing the axis in a rotational transformation is an eigenvector. Further in such transformation, the eigenvalue, i.e., $\lambda = 1$.

(b) Think of a second scenario, where the globe rotates by a small angle θ and also gets stretched in direction of the axis so that the sphere becomes an ellipse. Here the direction of every point on the axis remains as before the transformation, but its magnitude changes. Here also the direction of the axis represents the eigenvector. But here the eigenvalue, i.e., $\lambda \neq 1$. (In case the globe stretches outwards along the axis, $\lambda > 1$, but if it deflates then $\lambda < 1$.)

Eigen decomposition - - 1

- In order to effectively use eigenvalues and eigenvectors, one should understand the concept of eigen decomposition.
- It has already been said above that if A is a square matrix, v is a vector and λ is a scalar value (i.e., simply a number) then you can have $A\vec{v} = \vec{w} = \lambda\vec{v}$.
- Again note that the matrix A was able to change the magnitude but was unable to change the direction of $\vec{v}$ because $A\vec{v} = \vec{w} = \lambda\vec{v}$.
- Further note that the matrix A changed the magnitude of $\vec{v}$ by a factor of λ. (A special case would be λ = 1, i.e., the matrix A has no effect on the vector $\vec{v}$.
- If λ > 1, then the matrix A *stretches* $\vec{v}$ and if λ < 1, then A *compresses* $\vec{v}$.)
- To do eigen decomposition of a matrix, take a square matrix A of dimensions n × n, such that:
- $A = \begin{bmatrix} a_{11} & a_{21} & \dots & a_{n1} \\ a_{12} & a_{22} & \dots & a_{n2} \\ \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots \\ a_{1n} & a_{2n} & \dots & a_{nn} \end{bmatrix}$

Eigen decomposition - - 2

- Further suppose that the above matrix A has n eigenvalues. (Note that for a n × n matrix, the number of eigenvalues can at most be n and can even be less than n or even 0. This means that if you have a 3 × 3 matrix, then it may have from 0 up to 3 eigenvalues).
- You may call these n eigenvalues of Matrix A as: $\lambda_1, \lambda_2, \ldots \lambda_n$.
- So if the matrix A has n eigenvalues, then there will also be n eigenvectors.
- One can represent the n eigenvectors by $\vec{v}_1, \vec{v}_2, \ldots \vec{v}_n$. Note that each of these eigenvectors are actually column vectors of n rows and 1 column. So a typical eigenvector say $\vec{v}_1$ would be something like $\vec{v}_1 = \begin{bmatrix} x_{11} \\ x_{12} \\ \cdots \\ \ldots \\ x_{1n} \end{bmatrix}$ and $\vec{v}_2 = \begin{bmatrix} x_{21} \\ x_{22} \\ \cdots \\ \ldots \\ x_{2n} \end{bmatrix}$.

Eigen decomposition - - 3

- So if you create a matrix say Q so that Q = $[\vec{v}_1\ \vec{v}_2 \ldots \vec{v}_n]$, then Q would actually be
- $Q = \begin{bmatrix} x_{11} & x_{21} & \ldots & x_{n1} \\ x_{12} & x_{22} & \ldots & x_{n2} \\ \ldots & \ldots & \ldots & \ldots \\ \ldots & \ldots & \ldots & \ldots \\ x_{1n} & x_{2n} & \ldots & x_{nn} \end{bmatrix}$.
- Further create a diagonal matrix consisting of these n eigenvalues ($\lambda_1, \lambda_2, \ldots \lambda_n$). You can call this diagonal matrix L. (Note in common literature of maths, this diagonal matrix is represented by symbol Λ, i.e., capital lambda or by λ, i.e., lambda. But since in Python, you cannot use them as variable names so the letter L, i.e., first letter of lambda is used).
- Then L will be of form: $L = \begin{bmatrix} \lambda_1 & 0 & \ldots & 0 \\ 0 & \lambda_2 & \ldots & 0 \\ \ldots & \ldots & \ldots & \ldots \\ 0 & 0 & \ldots & 0 \\ 0 & 0 & \ldots & \lambda_n \end{bmatrix}$

Eigen decomposition - - 4

- Further note that Q^{-1} represents the inverse of matrix Q.
- Further suppose $Q^{-1} = \begin{bmatrix} y_{11} & y_{21} & \cdots & y_{n1} \\ y_{12} & y_{22} & \cdots & y_{n2} \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ y_{1n} & y_{2n} & \cdots & y_{nn} \end{bmatrix}$
- Now the eigenvalue decomposition says that a n × n square matrix A can be represented as:
- $A = QLQ^{-1}$.
- So you can write:

$$A = \begin{bmatrix} a_{11} & a_{21} & \cdots & a_{n1} \\ a_{12} & a_{22} & \cdots & a_{n2} \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ a_{1n} & a_{2n} & \cdots & a_{nn} \end{bmatrix} = \begin{bmatrix} x_{11} & x_{21} & \cdots & x_{n1} \\ x_{12} & x_{22} & \cdots & x_{n2} \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ x_{1n} & x_{2n} & \cdots & x_{nn} \end{bmatrix}$$

$$\begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & \lambda_n \end{bmatrix} \begin{bmatrix} y_{11} & y_{21} & \cdots & y_{n1} \\ y_{12} & y_{22} & \cdots & y_{n2} \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ y_{1n} & y_{2n} & \cdots & y_{nn} \end{bmatrix}$$

Eigen decomposition example - - 1

To make the concept clear, take a real example.

Suppose A = $\begin{bmatrix} 3 & 1 & -1 \\ 1 & 3 & -1 \\ -1 & -1 & 5 \end{bmatrix}$**. Then it can be shown that the three eigenvalues are $\lambda_1 = 6$, $\lambda_2 = 2$ and $\lambda_3 = 3$.**

Then putting the eigenvalues as diagonals of a diagonal matrix, you get L = $\begin{bmatrix} 6 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{bmatrix}$**.**

Further it can also be shown that the three eigenvectors will be

$\vec{v}_1 = \begin{bmatrix} -1/\sqrt{6} \\ -1/\sqrt{6} \\ 2/\sqrt{6} \end{bmatrix}$, $\vec{v}_2 = \begin{bmatrix} -1/\sqrt{2} \\ 1/\sqrt{2} \\ 0 \end{bmatrix}$ **and** $\vec{v}_3 = \begin{bmatrix} -1/\sqrt{3} \\ 1/\sqrt{3} \\ 1/\sqrt{3} \end{bmatrix}$**.**

Eigen decomposition example - - 2

Then the matrix Q consisting of the above three eigenvectors will be Q = $[\vec{v}_1\ \vec{v}_2\ \vec{v}_3]$

Or Q = $\begin{bmatrix} -1/\sqrt{6} & -1/\sqrt{2} & -1/\sqrt{3} \\ -1/\sqrt{6} & 1/\sqrt{2} & 1/\sqrt{3} \\ 2/\sqrt{6} & 0 & 1/\sqrt{3} \end{bmatrix}$**. Also you have Q^{-1} =** $\begin{bmatrix} -1/\sqrt{6} & -1/\sqrt{6} & 2/\sqrt{6} \\ -1/\sqrt{2} & 1/\sqrt{2} & 0 \\ -1/\sqrt{3} & 1/\sqrt{3} & 1/\sqrt{3} \end{bmatrix}$**.**

So the eigen decomposition of $A = Q \cdot L \cdot Q^{-1}$ is:

A = $\begin{bmatrix} 3 & 1 & -1 \\ 1 & 3 & -1 \\ -1 & -1 & 5 \end{bmatrix}$ **=** $\begin{bmatrix} -1/\sqrt{6} & -1/\sqrt{2} & -1/\sqrt{3} \\ -1/\sqrt{6} & 1/\sqrt{2} & 1/\sqrt{3} \\ 2/\sqrt{6} & 0 & 1/\sqrt{3} \end{bmatrix} \cdot \begin{bmatrix} 6 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{bmatrix} \cdot$

$\begin{bmatrix} -1/\sqrt{6} & -1/\sqrt{6} & 2/\sqrt{6} \\ -1/\sqrt{2} & 1/\sqrt{2} & 0 \\ -1/\sqrt{3} & 1/\sqrt{3} & 1/\sqrt{3} \end{bmatrix}$

Eigen decomposition example - - 3

The following example from the book shows the use of eigen decomposition.
[The output is not given]

```
import numpy as np
from numpy import dot #  Otherwise you have to use np.dot
from numpy.linalg import eig, inv
np.set_printoptions(precision=2, suppress = True)
#1--------define a matrix
A = np.array([[3, 1, -1],
              [1, 3, -1],
              [-1, -1, 5]])
print('A->')
print(A)# Check A
#2(a)---get eigen values as a 1-D array and eigen vectors as matrix Q
eig_arr, Q = eig(A)
print('Eigen values->', eig_arr)# eigen values are a 1-D array
print('Eigen vectors as a matrix->')
print(Q)# eigen vectors are columns of a square matrix
#2(b)------ Convert 1-D eigenvalue array eig_arr into matrix L
L = np.diag(eig_arr)
print('Eigen values as a diagonal matrix->')
print(L)# Confirm L is a diagonal matrix
#3(a)----------Get inverse of the eigen vector matrix Q
Q_inv = inv(Q)
print('Inverse of eigen vector matrix->')
print(Q_inv)# Confirm that inverse of eigen vector matrix is OK
#3(b)----Intermediate result ie B = L.Q_inv
B = L.dot(Q_inv)
print('Intermediate result B->')
print(B) # B is just an intermediate result
#3(c)------ get C = (Q).((L).(Q_inv)) ie C = Q.B
C = Q.dot(B)
print('C = (Q).((L).(Q_inv))->')
print(C)
#4----------Check if A and C are same/ similar
print('Is A similar to C?', np.allclose(A, C))
```

Singular Value Decomposition

- SVD is a decomposition of a $m \times n$ matrix say A into three matrices U, Σ and V such that: $A = U\Sigma V^T$;
- Where
  - U is an $m \times m$ unitary column orthogonal matrix. (Note for a unitary matrix U, $UU^T = I$). (Also note that column orthogonal means that any two columns of this matrix are orthogonal to each other meaning that the dot products of entries in any two columns will give a 0).
  - Σ is a $m \times n$ rectangular diagonal matrix with nonnegative numbers on its diagonal. (Rectangular diagonal matrix means all entries not on diagonal are 0).
  - V is a $n \times n$ unitary column orthogonal matrix.
- Note:
  - The diagonal entries of Σ are referred to as the singular values of A.
  - The columns of U are referred to as the left-singular vectors of A.
  - The columns of V are referred to as the right-singular vectors of A.

SVD -- 1

You can clearly see that SVD is very similar to eigen decomposition. The important differences are:

1. **Eigen decomposition is done for a square matrix, but SVD can be done for a non-square matrix also.**
2. **In eigen decomposition, when you form the diagonal matrix L from the eigenvalues, you don't insist upon the eigenvalues to be in any particular order. But in SVD, by convention you put the eigenvalues in the diagonal matrix in descending order on the diagonal.**

SVD -- 2

Suppose you have a matrix A = $\begin{bmatrix} 1 & -1 \\ 0 & 1 \\ 1 & 0 \end{bmatrix}$

Then, you can represent this as a composition of three matrices as:

$$A = \begin{bmatrix} 1 & -1 \\ 0 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} -2/\sqrt{6} & 0 & -1/\sqrt{3} \\ 1/\sqrt{6} & -1/\sqrt{2} & -1/\sqrt{3} \\ -1/\sqrt{6} & -1/\sqrt{2} & 1/\sqrt{3} \end{bmatrix} \begin{bmatrix} \sqrt{3} & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} -1/\sqrt{2} & 1/\sqrt{2} \\ -1/\sqrt{2} & -1/\sqrt{2} \end{bmatrix}$$

So you have U = $\begin{bmatrix} -2/\sqrt{6} & 0 & -1/\sqrt{3} \\ 1/\sqrt{6} & -1/\sqrt{2} & -1/\sqrt{3} \\ -1/\sqrt{6} & -1/\sqrt{2} & 1/\sqrt{3} \end{bmatrix}$, $\Sigma = \begin{bmatrix} \sqrt{3} & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix}$ **and**

$\mathbf{V^T} = \begin{bmatrix} -1/\sqrt{2} & 1/\sqrt{2} \\ -1/\sqrt{2} & -1/\sqrt{2} \end{bmatrix}$

SVD -- 3

- Have a look at the matrices U and V above.
- First, each column of U and V represents a unit vector.
- For example, the first column of U is $\begin{bmatrix} -2/\sqrt{6} \\ 1/\sqrt{6} \\ -1/\sqrt{6} \end{bmatrix}$ and is a unit vector. This is true for each column of U and V.
- Second, the dot product of any two column vectors of U or of v is 0. For example, the dot product of 2nd and 3rd column of U is
- $0 \times (-1/\sqrt{3}) + (-1/\sqrt{2}) \times (-1/\sqrt{3}) + (-1/\sqrt{2}) \times (1/\sqrt{3}) = 0$. So any two columns of U and v are orthonormal vectors.
- Now have a look at the Σ matrix. Here it is shown as $\Sigma = \begin{bmatrix} \sqrt{3} & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix}$. Note that the diagonal element with biggest value comes at top and the rest in descending order.

Using SVD for dimension reduction -- 1

- The example below explains how SVD works.
- In Table 18.4, 4 readers are represented in rows, i.e., Reader 1 to Reader4.
- Further the columns of this table are five books.
- Each entry in the table shows how each reader rates each book.

TABLE 18.4 Ratings of 5 books by four readers

	A History of π By Peter Beckmann	Things to Make and Do in the Fourth Dimension by Matt Parker	What is Mathematics? By Courant and Robbins	Don Quixote by Miguel de Cervantes	Moby Dick by Herman Melville	The Wealth of Nations by Adam Smith
Reader 1	10	8	9	1	0	0
Reader 2	9	10	9	0	1	1
Reader 3	1	0	1	10	8	1
Reader 4	0	1	1	9	8	0

Notice that:

- The first 3 books are related to maths, while the 4th and 5th books are related to literature. Book number 6 is of economics so unrelated to maths or literature.
- Also note that the first 2 readers seem to have a choice for maths books while reader 3 and 4 seem to prefer literature books.

Using SVD for dimension reduction -- 2

So here you have A:

$$A \begin{bmatrix} 10 & 8 & 9 & 1 & 0 & 0 \\ 9 & 10 & 9 & 0 & 1 & 1 \\ 1 & 0 & 1 & 10 & 8 & 1 \\ 0 & 1 & 1 & 9 & 8 & 0 \end{bmatrix} = U \begin{bmatrix} -0.67 & 0.17 & 0.64 & 0.33 \\ -0.7 & 0.19 & -0.62 & -0.32 \\ -0.18 & -0.71 & 0.31 & -0.61 \\ -0.17 & -0.66 & -0.33 & 0.65 \end{bmatrix}$$

$$\Sigma \begin{bmatrix} 22.81 & 0 & 0 & 0 & 0 & 0 \\ 0 & 17.26 & 0 & 0 & 0 & 0 \\ 0 & 2 & 2.14 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.96 & 0 & 0 \end{bmatrix}$$

$$V^T \begin{bmatrix} -0.58 & -0.55 & -0.56 & -0.18 & -0.16 & -0.04 \\ 0.16 & 0.15 & 0.11 & -0.74 & -0.62 & -0.03 \\ 0.54 & -0.65 & 0.09 & 0.33 & -0.39 & -0.15 \\ -0.19 & 0.11 & 0.14 & 0.05 & -0.02 & -0.96 \\ 0.04 & -0.49 & 0.45 & -0.49 & 0.56 & -0.03 \\ -0.56 & -0.09 & 0.67 & 0.25 & -0.35 & 0.22 \end{bmatrix}$$

Using SVD for dimension reduction -- 3

If you look at Σ, its diagonals are: [22.81 17.26 2.14 0.96].

Out of these 4 values, only the first and second are significant. So you can think of these two as the two categories in which the books are falling into. The 3rd and 4th entries can be thought of as noise. Keep only the first 2 entries of Σ and convert the other 2 to 0. Call this modified Σ as Σmod. Then Σmod = $\begin{bmatrix} 22.81 & 0 & 0 & 0 & 0 & 0 \\ 0 & 17.26 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$ **. Also keep only the first 2 columns of matrix U. Then Umod =** $\begin{bmatrix} -0.67 & 0.17 \\ -0.7 & 0.19 \\ -0.18 & -0.71 \\ -0.17 & -0.66 \end{bmatrix}$ **. Looking at U, you can see that the first two entries of column 1 are towards one topic, while the last 2 entries of column 2 are towards topic 2. You can think of Umod as a Reader-category matrix, where the matrix shows the preference of a reader for books of a particular category.**

Using SVD for dimension reduction -- 4

- You can think of V^Tmod as a category-book matrix.
- The first row represents the category of Maths books while the second row represents the category of Literature books.
- So the first three books belong more to the category Maths while 4th and 5th books belong more to the Literature category.
- Note that the 6th book does not fall in any particular category.

So you can say that:

- The U matrix represents the Reader-Category relationship
- The V matrix represents the Category-Books relationship.
- The Σ matrix represents the relative abundance/ strength of each category.

Using SVD for dimension reduction -- 5

- You may reconstruct the original matrix from the modified matrices. Reconstruction means finding a matrix
- Y = Umod·Σmod ·V^Tmod.

Then:

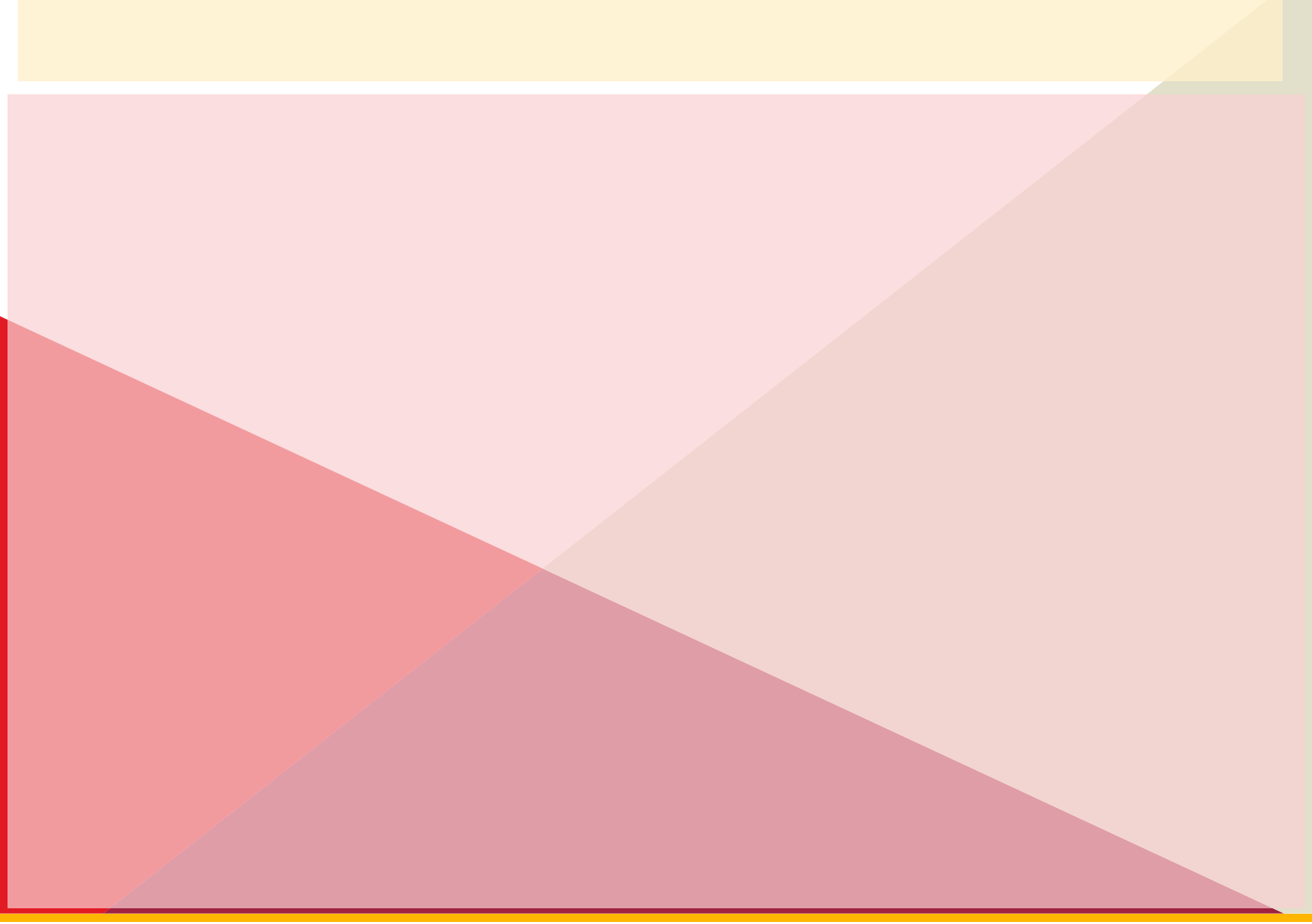
- Y = *-0.670.17-0.70.19-0.18-0.71-0.17-0.66 . 22.810017.26. -0.58-0.55-0.56-0.18-0.16-0.040.160.150.11-0.74-0.62-0.03*
- If you do the above calculation, you will get:

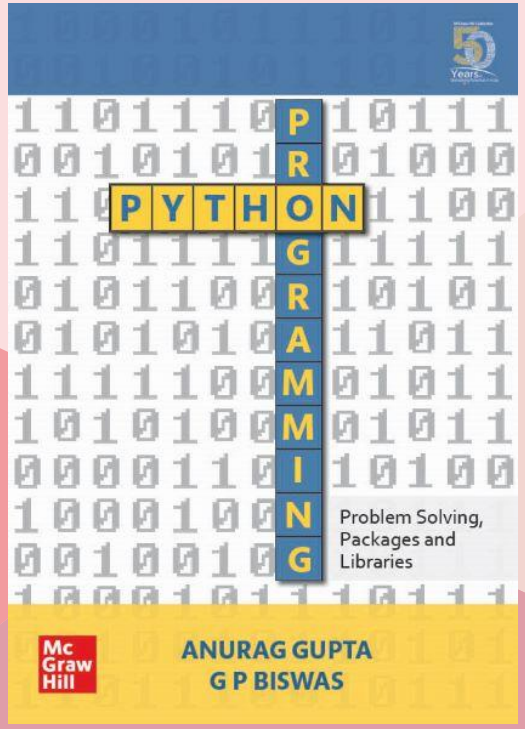
  Y-> [[9.31 8.85 8.84 0.53 0.54 0.5]
  [9.66 9.18 9.16 0.45 0.48 0.52]
  [0.53 0.49 1.03 9.82 8.25 0.53]
  [0.51 0.47 0.97 9.2 7.73 0.5]]
- You can see that matrix Y is very similar to original Matrix A.
- You can compare these two matrices by taking their Frobenius norm.
- The Frobenius norm of a matrix of dimensions m × n is given by:

$$\|A_F\| = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|a_{ij}|^2}$$

Script to implement SVD

```
1   from scipy import linalg
2   from numpy import array, dot, shape, diag, zeros
3   A = array([[10, 8, 9, 1, 0, 0],
4              [9, 10, 9, 0, 1, 1],
5              [1, 0, 1, 10, 8, 1],
6              [0, 1, 1, 9, 8, 0],])
7   # 1---Create SVD of matrix A and convert S from 1_d array to matrix---
8   #1(a)---SVD of A ie A = U.S.Vt
9   U, S, Vt = linalg.svd(A)
10  print('U->', U)
11  print('S->', S)
12  print('Vt->', Vt)
13  print(type(S))
14  # 1(b) ---Create Sm ie matrix from linear 1-D array S----
15  M,N = A.shape
16  Sm = linalg.diagsvd(S,M,N)
17  print('Sm->', Sm)
18  #2 Modify Vt to have 2 rows. Modify S to be 2x2. Modify Y to have 2 columns
19  S2 = Sm[:2, :2]
20  Vt2 = Vt[:2, :]
21  U2 = U[:, :2]
22  print('Vt2->', Vt2) # Confirm Vt2 has 2 rows
23  print('S2->', S2) # Confirm S2 is 2 x 2
24  print('U2->', U2)# Confirm U2 has 2 columns
25  #3------- Reconstruct Y = U2.S2.Vt2 ---
26  #3(a)----X is an intermediate dot product S2.Vt2
27  X = S2.dot(Vt2)
28  #3(b)-----Y = ((U2.(S2.Vt2)) ie Y = U2.X -----
29  Y = U2.dot(X)
30  print('Y->', Y)
31  #4---Compare original matrix A to matrix Y using Frobenius norm
32  Frob_Y = linalg.norm(Y)
33  print('Frobenius norm of Y->', Frob_Y)
34  Frob_A = linalg.norm(A)
35  print('Frobenius norm of A->', Frob_A)
```




PYTHON
PROGRAMMING
Problem Solving, Packages and Libraries
ANURAG GUPTA
G P BISWAS
Mc Graw Hill

Because learning changes everything.®

Thank You!

For any queries or feedback contact us at:

support.india@mheducation.com

1800-103-5875

www.mheducation.co.in