Matrix Methods: Using NumPy
MPHYCC-05 unit IV (Sem.-II)

Goal of this document is to teach you the basics of matrix. Specially to create a matrix; understanding their mathematical operation - addition, subtraction, multiplication, inversion; statistical properties – mean, max, variance, standard deviation, and their properties – determinant, trace, eigenvalue, eigenvector; and many more.

Document in a Glance:

1.1: Creating a Matrix
1.2: Creating a Sparse Matrix
1.3: Selecting Elements of Matrix
1.4: Shape, Size, and dimension of Matrix
1.5: Operation with Matrix
1.6: Statistics of a Matrix
1.7: Reshaping the matrix
1.8: Transposing a Matrix and a vector
1.9: Flattening a Matrix
1.10: Rank and Determinant of a Matrix
1.11: Diagonal of a Matrix
1.12: Trace of a Matrix
1.13: Getting Eigenvalues and Eigenvectors
1.14: Adding and Subtracting Matrices
1.15: Product of the Matrices
1.16: Inverting a Matrix
1.17: Generating Random values

SP Gupta

1.1 Creating a Matrix: We can create a matrix using NumPy. First we create a row and column vector using array method of NumPy.

Row and column vector: one dimensional array

```
>>>import numpy as np
# row vector
>>>vector_row = np.array([4, 5, 6])
# column vector
>>>vector_column = np.array([[4], [5], [6]])

>>> print(vector_row)
[4 5 6]

# selecting second element of row vector
>>> vector_row[1]
5
>>> print(vector_column)
[[4]
 [5]
 [6]]

# selecting second element of column vector
>>> vector_column[1]
array([5])
```

Two dimensional arrays:

```
>>>matrix = np.array([[1, 2], [3, 4], [5, 6]])
>>> print(matrix)
[[1 2]
 [3 4]
 [5 6]]

# selecting second row and second column element
>>> matrix[1][1]
4
```

S P Gupta

1.2 Creating a Sparse Matrix:

Sometimes in the scientific data most of the elements in the data are zeros. That is the given data with very few nonzero values, and we want to efficiently represent it. We can use the *sparse,* a method of SciPy which only store nonzero elements and assume all other values will be zero, leading to significant computational savings. If we view the sparse matrix we can see that only the nonzero values are stored:

```
>>>import numpy as np
>>>from scipy import sparse

>>>matrix = np.array([[0, 0],
                [0, 4],
                [5, 0]])

# Create compressed sparse row (CSR) matrix
>>>matrix_sparse = sparse.csr_matrix(matrix)
>>>print(matrix_sparse)
  (1, 1)4
  (2, 0)5
```

There are a number of types of sparse matrices. However, in *compressed sparse row (CSR)* matrices, (1, 1) and (2, 0) represent indices of the zero-indexed (index of the first element is zero, not one) non-zero values 4 and 5, respectively. For example, the element 4 is in the second row and second column. We can see the advantage of sparse matrices if we create a much larger matrix with many more zero elements and then compare this larger matrix with our original sparse matrix:

```
# Create larger matrix
>>>matrix_large = np.array([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                     [0, 4, 0, 0, 0, 0, 0, 0, 0, 0],
                     [5, 0, 0, 0, 0, 0, 0, 0, 0, 0]])

# Create compressed sparse row (CSR) matrix
>>>matrix_large_sparse = sparse.csr_matrix(matrix_large)
# View original sparse matrix
>>>print(matrix_sparse)
  (1, 1)    1
  (2, 0)    3
# View larger sparse matrix
```

S P Gupta

```
>>>print(matrix_large_sparse)
  (1, 1)    1
  (2, 0)    3
```

As we can see, despite the fact that we added many more zero elements in the larger matrix, its sparse representation is exactly the same as our original sparse matrix. That is, the addition of zero elements did not change the size of the sparse matrix.

1.3 Selecting Elements of Matrix:

NumPy offers a wide variety of methods for selecting row, columns, and elements of matrix.

```
>>>import numpy as np
# Create row vector
>>>vector = np.array([1, 2, 3, 4, 5, 6])
# Create matrix
>>>matrix = np.array([[5, 1, 3],
                      [4, 5, 6],
                      [7, 8, 9]])

# Select third element of vector
>>>vector[3]
4
# Select second row, second column
>>>matrix[2,1]
8
# Select all elements of a vector
>>>vector[:]
array([1, 2, 3, 4, 5, 6])
# Select everything up to and including the fourth element
>>>vector[:3]
array([1, 2, 3, 4])
# Select everything after the second element
>>>vector[2:]
array([3, 4, 5, 6])
# Select the last element
>>>vector[-1]
6
# Select the first two rows and all columns of a matrix
```

S P Gupta

```
>>>matrix[:2,:]
array([[5, 1, 3],
       [4, 5, 6]])
# Select all rows and the second column
>>>matrix[:,1:2]
array([[2],
       [5],
       [8]])
# Select all rows and first second columns
>>>matrix[:,:2]
array([[5, 1],
       [4, 5],
       [7, 8]])
```

1.4 Shape, Size, and dimension of Matrix:

You can also find out the shape, size, and dimensions of the matrix using *shape*, *size* and *ndim* methods.

```
>>>import numpy as np
# Create matrix
>>>matrix = np.array([[1, 2, 3, 4],
                  [5, 6, 7, 8],
                  [9, 10, 11, 12]])

# View number of rows and columns
>>>matrix.shape
(3, 4)
# View number of elements (rows * columns)
>>>matrix.size
12
# View number of dimensions
>>>matrix.ndim
2
```

1.5 Operation with Matrix:

You can apply addition, subtraction, multiplication.

```
>>>import numpy as np
```

S P Gupta

```
# Create matrix
>>>matrix = np.array([[1, 2, 3],
                      [4, 5, 6],
                      [7, 8, 9]])

>>>print(matrix+100)
[[101 102 103]
 [104 105 106]
 [107 108 109]]

>>>print(matrix*100)
[[100 200 300]
 [400 500 600]
 [700 800 900]]

>>>print(matrix/100)
[[0.01 0.02 0.03]
 [0.04 0.05 0.06]
 [0.07 0.08 0.09]]

>>>print(matrix//2)
[[0 1 1]
 [2 2 3]
 [3 4 4]]

>>>print(matrix%2)
[[1 0 1]
 [0 1 0]
 [1 0 1]]
```

S P Gupta

1.6 Statistics of a Matrix:

Often we want to know the maximum and minimum value in an array or subset of an array. This can be accomplished with the max and min methods. Using the axis parameter we can also apply the operation along a certain axis:

```
>>>import numpy as np
```

```
# Create matrix
>>>matrix = np.array([[1, 2, 3],
                 [4, 5, 6],
                 [7, 8, 9]])

# find the maximum of the matrix
>>>np.max(matrix)
9
# find the minimum of the matrix
>>>np.min(matrix)
1
# find maximum in each column
>>>np.max(matrix, axis=0)
array([7, 8, 9])
# Find maximum element in each row
>>>np.max(matrix, axis=1)
array([3, 6, 9])
```

Most of the time it is required to know the statistics of the matrix data; mean, variance and standard deviation and can be easily calculated using numpy.

```
# Mean of the matrix
>>>np.mean(matrix)
5.0

# Variance of the matrix
>>>np.var(matrix)
6.666666666666667

# Standard deviation of the matrix
>>>np.std(matrix)
2.581988897471611 2

# Find the mean value in each column
>>>np.mean(matrix, axis=0)
array([ 4.,  5.,  6.])
```

S P Gupta

1.7 Reshaping the matrix:

Reshaping allow to change the shape i. e. the number of rows and columns of a matrix without changing the element values. *reshape* allows us to restructure a matrix so that the data remain maintained it is organized as in a different number of rows and columns. The only requirement is that the shape of the original and new matrix contains the same number of elements. We can check this by finding the size of a matrix using *size*:

```
>>>import numpy as np
# Create 4x3 matrix
>>>matrix = np.array([[1, 2, 3],
                [4, 5, 6],
                [7, 8, 9],
                [10, 11, 12]])

# Reshape matrix into 2x6 matrices
>>>matrix.reshape(2, 6)
array([[ 1,  2,  3,  4,  5,  6],
       [ 7,  8,  9, 10, 11, 12]])

>>>matrix.size
12
```

One useful argument in reshape is -1, which effectively means "as many as needed," so reshape(-1, 1) means one row and as many columns as needed:

```
>>>matrix.reshape(1, -1)
array([[ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12]])
```

Finally, if we provide one integer, reshape will return a 1D array of that length:

```
>>>matrix.reshape(12)
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
```

S P Gupta

1.8 Transposing a Matrix and a vector:

In transposing operation column and row indices of each element are interchanged. Technically, a vector cannot be transposed because it is just a collection of values. However, it is common practice to transposing a vector is the nothing but converting a row vector to a column vector or vice versa: For transposing in python the method is known as *T* method:

```
>>>import numpy as np
>>>matrix = np.array([[1, 2, 3],
                 [4, 5, 6],
                 [7, 8, 9]])

# Transpose the above matrix
>>>matrix.T
array([[1, 4, 7],
       [2, 5, 8],
       [3, 6, 9]])
# Transpose a vector
>>>np.array([1, 2, 3, 4, 5, 6]).T
array([1, 2, 3, 4, 5, 6])

# Transpose row vector
>>>np.array([[1, 2, 3, 4, 5, 6]]).T
array([[1],
       [2],
       [3],
       [4],
       [5],
       [6]])

# Transpose column vector
>>>np.array([[1], [2], [3]]]).T
 array([[1, 2, 3]])
```

1.9 Flattening a Matrix:

Flatten is a simple method to change the matrix into a one-dimensional array. We can also use reshape to create a row vector. We can use *flatten* or *reshape* method. The

S P Gupta

method *reshape* is already discussed in the previous section, now the discussion is about *flatten*.

```
import numpy as np
>>>matrix = np.array([[1, 2, 3],
                 [4, 5, 6],
                 [7, 8, 9]])

# Flatten the matrix
>>>matrix.flatten()
array([1, 2, 3, 4, 5, 6, 7, 8, 9])

>>>matrix.reshape(1, -1)
array([[1, 2, 3, 4, 5, 6, 7, 8, 9]])
>>>matrix.reshape(9)
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

1.10 Rank and Determinant of a Matrix:

The rank of a matrix is the dimensions of the vector space spanned by its columns or rows. Rank of matrix can be easily found in NumPy using *matrix_rank*.

```
import numpy as np
>>>matrix np.array([[1, 2, 3],
                 [4, 5, 6],
                 [7, 8, 9]])
# find the matrix rank
>>>np.linalg.matrix_rank(matrix)
2
```

Determinant of a matrix can also be calculated using NumPy function.

```
# calculate determinant of matrix
>>>np.linalg.det(matrix)
6.66133814775094e-16
#note e is 10
```

SP Gupta

1.11 Diagonal of a Matrix:

It is also possible to get a diagonal element using *diagonal* method and the off diagonal element using offset method along with *diagonal* method.

```
>>>import numpy as np
>>>matrix np.array([[1, 2, 3],
                [4, 5, 6],
                [7, 8, 9]])

# finding the diagonal elements
>>>matrix.diagonal()
array([1, 5, 9])

# finding diagonal one above the main diagonal
>>>matrix.diagonal(offset=1)
array([2, 6])

# finding diagonal one below the main diagonal
>>>matrix.diagonal(offset=-1)
array([4, 8])

# finding diagonal two below and two above the main diagonal
>>> matrix.diagonal(offset=-2)
array([7])
>>> matrix.diagonal(offset=2)
array([3])
```

S P Gupta

1.12 Trace of a Matrix:

The trace of a matrix is the sum of the diagonal elements and can be calculated by using trace method. Trace can also be calculated by adding the diagonal element after finding the diagonal element. For adding one can use *sum* method.

```
>>>import numpy as np
>>>matrix np.array([[1, 2, 3],
                [4, 5, 6],
                [7, 8, 9]])
```

```
# calculate the trace
>>>matrix.trace()
15
# Finding the diagonal and then sum of the elements
>>>sum(matrix.diagonal())
15
```

1.13 Getting Eigenvalues and Eigenvectors:

If a linear transformation represented by a matrix, A, then eigenvectors are vectors that, when that transformation is applied, change only in scale but not in direction. This can be represented mathematically as follows. More formally:

$$A\,v = \lambda\,v$$

Where A is a square matrix, λ contains the eigenvalues and v contains the eigenvectors. There is linear algebra toolset known as *eig* in NumPy is helpful to calculate the eigenvalues, and eigenvectors of any square matrix.

```
>>>import numpy as np
>>>matrix np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

# calculate eigenvalues and eigenvectors
>>>eigenvalues, eigenvectors = np.linalg.eig(matrix)

# View eigenvalues
>>>eigenvalues
array([ 1.61168440e+01, -1.11684397e+00, -1.30367773e-15])

# View eigenvectors
>>>eigenvectors
array([[-0.23197069, -0.78583024,  0.40824829],
       [-0.52532209, -0.08675134, -0.81649658],
       [-0.8186735,  0.61232756,  0.40824829]])
```

1.14 Adding and Subtracting Matrices:

Use add and subtract method, alternatively, we can simply use the + and – operators.

```
>>>import numpy as np
>>>matrix_a = np.array([[1, 1, 1],
                        [1, 1, 1],
                        [1, 1, 2]])
>>>matrix_b = np.array([[1, 3, 1],
                        [1, 3, 1],
                        [1, 3, 8]])
# Add two matrices
>>>np.add(matrix_a, matrix_b)
array([[ 2,  4,  2],
       [ 2,  4,  2],
       [ 2,  4, 10]])
# Subtract two matrices
>>>np.subtract(matrix_a, matrix_b)
array([[ 0, -2,  0],
       [ 0, -2,  0],
       [ 0, -2, -6]])
# Alternative way to Add two matrices
>>>matrix_a + matrix_b
array([[ 2,  4,  2],
       [ 2,  4,  2],
       [ 2,  4, 10]])
```

1.15 Product of the Matrices:

Dot product of two vectors ***a*** and ***b***, is defined as:

$$\boldsymbol{a.b} = \sum_{i=1}^{n} \boldsymbol{a}_i \boldsymbol{b}_i$$

Where a_i and b_i are the ith element of vector *a* and *b*. We can use NumPy's dot (.) class to calculate the dot product. Alternatively, we can use the new @ operator:

```
>>>import numpy as np

# Create two vectors
>>>vector_a = np.array([1,2,3])
>>>vector_b = np.array([4,5,6])

# calculate dot product
```

S P Gupta

```
>>>np.dot(vector_a, vector_b)
32
#Alternative way to calculate dot product
>>>vector_a @ vector_b
32
```

Cross product of two matrices

```
# Calculate cross product
>>>np.cross(vector_a, vector_b)
[-3  6 -3]
```

Here, vector $\boldsymbol{a} = 1\mathrm{i} + 2\mathrm{j} + 3\mathrm{k}$ and vector $\boldsymbol{b} = 4\mathrm{i} + 5\mathrm{j} + 6\mathrm{k}$ and their cross product is again a vector say c, as $\boldsymbol{c} = -3\mathrm{i} + 6\mathrm{j} - 3\mathrm{k}$

Multiply two matrices

```
>>>import numpy as np
# create matrix
>>>matrix_a = np.array([[1, 1],
                        [1, 2]])
# create matrix
>>>matrix_b = np.array([[1, 3],
                        [1, 2]])

# multiplying two matrices
>>>np.dot(matrix_a, matrix_b)
array([[2, 5],
       [3, 7]])

# Alternative way to multiply two matrices
>>>matrix_a @ matrix_b
array([[2, 5],
       [3, 7]])
```

Multiply two matrices element-wise

```
>>>matrix_a * matrix_b
array([[1, 3],
       [1, 4]])
```

S P Gupta

1.16 Inverting a Matrix:

The inverse of a square matrix, A, is also a matrix A^{-1}, such that:

$$A\,A^{-1} = I$$

Where I is the identity matrix. We can use *linalg.inv* to calculate A^{-1} if it exists.

```
>>>import numpy as np
# Create matrix
>>>matrix = np.array([[1, 4],
                [2, 5]])

# Calculate inverse of matrix
>>>np.linalg.inv(matrix)
array([[-1.66666667,  1.33333333],
      [ 0.66666667, -0.33333333]])
# Multiply matrix and its inverse
>>>matrix @ np.linalg.inv(matrix)
array([[ 1.,  0.],
      [ 0.,  1.]])
```

1.17 Generating Random values:

If you want to generate pseudorandom values, that can be generated using *random* method of NumPy.

```
# Generate three random floats between 0.0 and 1.0
>>> import numpy as np
>>> np.random.random(3)
array([0.54488318, 0.4236548 , 0.64589411])

# Generate three random integers between 0 and 10
>>>np.random.randint(0, 11, 3)
array([3, 7, 9])

# Draw three numbers from a normal distribution with mean 0.0
# and standard deviation of 1.0
>>>np.random.normal(0.0, 1.0, 3)
array([-1.42232584,  1.52006949, -0.29139398])
```

```
# Draw three numbers from a logistic distribution with mean 0.0 #and scale of 1.0
>>>np.random.logistic(0.0, 1.0, 3)
array([-0.98118713, -0.08939902,  1.46416405])

# Draw three numbers greater than or equal to 1.0 and less than 2.0
>>>np.random.uniform(1.0, 2.0, 3)
array([ 1.47997717,  1.3927848 ,  1.83607876])
```

Finally, it can sometimes be useful to return the same random numbers for multiple times to get predictable, repeatable results. We can do this by setting the "seed" (an integer) of the pseudorandom generator. Random processes with the same seed will always produce the same output.

```
# Set seed and see the effect of seed

# with seed value 0
>>> np.random.seed(0)
>>> np.random.random(4)
array([0.5488135 , 0.71518937, 0.60276338, 0.54488318])

>>> np.random.seed(0)
>>> np.random.random(4)
array([0.5488135 , 0.71518937, 0.60276338, 0.54488318])

# without seed
>>> np.random.random(4)
array([0.4236548 , 0.64589411, 0.43758721, 0.891773  ])

# with seed value 3
>>> np.random.seed(3)
>>> np.random.random(4)
array([0.5507979 , 0.70814782, 0.29090474, 0.51082761])
>>> np.random.seed(3)
>>> np.random.random(4)
array([0.5507979 , 0.70814782, 0.29090474, 0.51082761])
```

S P Gupta