EXPERIMENT NO. 13

Write Python programs to implement different Linear Algebra functions using Scipy.

Roll No.	01
Name	Aamir Ansari
Class	D10A
Subject	Python Lab
LO Mapped	LO1: Understand the structure, syntax, and semantics of the Python language. LO6: Design and Develop cost-effective robust applications using the latest Python trends and technologies.

Aim: Write Python programs to implement different Linear Algebra functions using Scipy.

Introduction:

SciPy:
Scipy is a scientific library for Python is an open source, BSD-licensed library for mathematics, science and engineering. The SciPy library depends on NumPy, which provides convenient and fast N-dimensional array manipulation. The main reason for building the SciPy library is that it should work with NumPy arrays. It provides many user-friendly and efficient numerical practices such as routines for numerical integration and optimization.

Standard Python distribution does not come bundled with any SciPy module. A lightweight alternative is to install SciPy using the popular Python package installer,

```
pip install pandas
```

SciPy sub-packages: SciPy is organized into sub-packages covering different scientific computing domains.

Sub-Packages	Description
scipy.cluster	Vector quantization / Kmeans
scipy.constants	Physical and mathematical constants
scipy.fftpack	Fourier transform
scipy.integrate	Integration routines
scipy.interpolate	Interpolation
scipy.io	Data input and output
scipy.linalg	Linear algebra routines
scipy.ndimage	n-dimensional image package
scipy.odr	Orthogonal distance regression
scipy.optimize	Optimization
scipy.signal	Signal processing
scipy.sparse	Sparse matrices
scipy.spatial	Spatial data structures and algorithms
scipy.special	Any special mathematical functions

scipy.stats	Statistics

SciPy.linalg vs NumPy.linalg:
A scipy.linalg contains all the functions that are in numpy.linalg. Additionally, scipy.linalg also has some other advanced functions that are not in numpy.linalg. Another advantage of using scipy.linalg over numpy.linalg is that it is always compiled with BLAS/LAPACK support, while for NumPy this is optional. Therefore, the SciPy version might be faster depending on how NumPy was installed.

Square root function: This function takes the matrix and returns the square root of the matrix.

```
a = np.array([[1.0, 3.0], [1.0, 4.0]])

x = sqrtm(a)
print x
```

Logarithmic function: This function takes the matrix and returns the matrix logarithm.

```
a = np.array([[1.0, 3.0], [1.0, 4.0]])

x = logm(a)
print x
```

Linear Equations: The scipy.linalg.solve feature solves the linear equation a * x + b * y = Z, for the unknown x, y values. It is better to use the linalg.solve command, which can be faster and more numerically stable. The solve function takes two inputs ‘a’ and ‘b’ in which ‘a’ represents the coefficients and ‘b’ represents the respective right hand side value and returns the solution array.

```
a = np.array([[3, 2, 0], [1, -1, 0], [0, 5, 1]])
b = np.array([2, 4, -1])

x = linalg.solve(a, b)
print x
```

Inverse of a matrix: The inverse of a matrix A is the matrix B, such that AB = I, where I is the identity matrix consisting of ones down the main diagonal. Usually, B is denoted $B = A^{-1}$. In SciPy, the matrix inverse of the NumPy array, A, is obtained using linalg.inv (A), or using A.I if A is a Matrix.

```
a = np.array([[1,3,5],[2,5,1],[2,3,8]])
x = linalg.inv(a)
print x
```

Finding the determinant: The determinant of a square matrix A is often denoted |A| and is a quantity often used in linear algebra. Suppose a_{ij} are the elements of the matrix A and let M_{ij} =

$|A_{ij}|$ be the determinant of the matrix left by removing the ith row and jth column from A. Then, for any row i,

$$|\mathbf{A}| = \sum_j (-1)^{i+j} a_{ij} M_{ij}.$$

This is a recursive way to define the determinant, where the base case is defined by accepting that the determinant of a 1 x 1 matrix is the only matrix element. In SciPy the determinant can be calculated with linalg.det.

```
a = np.array([[1,2],[3,4]])

x = linalg.det(a)
print x
```

Computing norms: Matrix and vector norms can also be computed with SciPy. A wide range of norm definitions are available using different parameters to the order argument of linalg.norm. This function takes a rank-1 (vectors) or a rank-2 (matrices) array and an optional order argument (default is 2). Based on these inputs, a vector or matrix norm of the requested order is computed.

```
a = np.array([[1,2],[3,4]])

x = linalg.norm(A)
y = linalg.norm(A,'fro')
print x
print y
```

Eigenvalues and Eigenvectors: The eigenvalue-eigenvector problem is one of the most commonly employed linear algebra operations. We can find the Eigenvalues (λ) and the corresponding Eigenvectors (v) of a square matrix (A) by considering the following relation **Av = λv.**

```
A = np.array([[1,2],[3,4]])

l, v = linalg.eig(A)
print l                    #eigenvalues
print v                    #eigenvectors
```

Scipy.linalg.eig computes the eigenvalues from an ordinary or generalized eigenvalue problem. This function returns the Eigenvalues and the Eigenvectors.

Single value decomposition: A Singular Value Decomposition (SVD) can be thought of as an extension of the eigenvalue problem to matrices that are not square. The **scipy.linalg.svd** factorizes the matrix A into two unitary matrices U and Vh and a 1-D array s of singular values (real, non-negative) such that a == U*S*Vh, where S is a suitably shaped matrix of zeros with the main diagonal s.

```
a = np.random.randn(3, 2) + 1.j*np.random.randn(3, 2)

U, s, Vh = linalg.svd(a)
print U, Vh, s
```

Arbitary function: Any arbitrary function that takes one complex number and returns a complex number can be called as a matrix function using the command **linalg.funm**. This command takes the matrix and an arbitrary Python function. It then implements an algorithm to compute the function applied to the matrix using a Schur decomposition.

```
a = np.random.seed(1234)

x = random.rand(3, 3)
y = linalg.funm(A, lambda x: special.jv(0, x))
print x
print y
```

Other Basic functions:

Functions	Description
solve_banded(l_and_u, ab, b[, overwrite_ab, …])	Solve the equation a x = b for x, assuming a is a banded matrix.
solveh_banded(ab, b[, overwrite_ab, …])	Solve equation a x = b.
solve_circulant(c, b[, singular, tol, …])	Solve C x = b for x, where C is a circulant matrix.
solve_triangular(a, b[, trans, lower, …])	Solve the equation a x = b for x, assuming a is a triangular matrix.
lstsq(a, b[, cond, overwrite_a, …])	Compute least-squares solution to equation Ax = b.
pinv(a[, cond, rcond, return_rank, check_finite])	Compute the (Moore-Penrose) pseudo-inverse of a matrix.
matrix_balance(A[, permute, scale, …])	Compute a diagonal similarity transformation for row/column balancing.
subspace_angles(A, B)	Compute the subspace angles between two matrices.

Other Eigen functions:

Function	Description
eigvalsh(a[, b, lower, overwrite_a, …])	Solves a standard or generalized eigenvalue problem for a complex Hermitian or real symmetric matrix.
eig_banded(a_band[, lower, eigvals_only, …])	Solve real symmetric or complex Hermitian band matrix eigenvalue problem.
eigvals_banded(a_band[, lower, …])	Solve real symmetric or complex Hermitian band matrix eigenvalue problem.
eigh_tridiagonal(d, e[, eigvals_only, …])	Solve eigenvalue problem for a real symmetric tridiagonal matrix.
eigvalsh_tridiagonal(d, e[, select, …])	Solve eigenvalue problem for a real symmetric tridiagonal matrix.

Other Matrix functions:

Function	Description
expm(A)	Compute the matrix exponential using Pade approximation.
cosm(A)	Compute the matrix cosine.
sinm(A)	Compute the matrix sine.
tanm(A)	Compute the matrix tangent.
coshm(A)	Compute the hyperbolic matrix cosine.
sinhm(A)	Compute the hyperbolic matrix sine.
tanhm(A)	Compute the hyperbolic matrix tangent.
signm(A[, disp])	Matrix sign function.

Other Matrix Equation solver function:

Function	Description
solve_sylvester(a, b, q)	Computes a solution (X) to the Sylvester equation.
solve_continuous_are(a, b, q, r[, e, s, …])	Solves the continuous-time algebraic Riccati equation (CARE).

solve_discrete_are(a, b, q, r[, e, s, balanced])	Solves the discrete-time algebraic Riccati equation (DARE).
solve_continuous_lyapunov(a, q)	Solves the continuous Lyapunov equation.
solve_discrete_lyapunov(a, q[, method])	Solves the discrete Lyapunov equation.

Other Special Matrices function:

Function	Description
block_diag(*arrs)	Create a block diagonal matrix from provided arrays.
circulant(c)	Construct a circulant matrix.
companion(a)	Create a companion matrix.
convolution_matrix(a, n[, mode])	Construct a convolution matrix.
dft(n[, scale])	Discrete Fourier transform matrix.
fiedler(a)	Returns a symmetric Fiedler matrix.
hadamard(n[, dtype])	Construct a Hadamard matrix.
hankel(c[, r])	Construct a Hankel matrix.
helmert(n[, full])	Create an Helmert matrix of order n.
hilbert(n)	Create a Hilbert matrix of order n.
leslie(f, s)	Create a Leslie matrix.
pascal(n[, kind, exact])	Returns the n x n Pascal matrix.

Results:

```
>>> from scipy import linalg
>>> import numpy as np
>>> a = np.array([[3,2,0],[1,-1,0],[0,5,1]])
>>> b = np.array([2,4,-1])
>>> x = linalg.solve(a,b)
>>> print(x)
[ 2. -2.  9.]
>>> A = np.array([[1,2], [3,4]])
>>> x = linalg.det(A)
>>> print(x)
-2.0

>>> a = np.array([[1. ,2.],[3., 4.]])
>>> linalg.inv(a)
array([[-2. ,  1. ],
       [ 1.5, -0.5]])

>>> A = np.array([[1,2],[3,4]])
>>> l, v = linalg.eig(A)
>>> print(l)
[-0.37228132+0.j  5.37228132+0.j]
>>> print(v)
[[-0.82456484 -0.41597356]
 [ 0.56576746 -0.90937671]]

>>> a = np.random.randn(3,2) + 1.j*np.random.randn(3,2)
>>> U, s, Vh = linalg.svd(a)
>>> print(U, s, Vh)
[[ 0.6262469 +0.28764011j  0.13811225-0.57632227j  0.28973337-0.29985031j]
 [ 0.61298099+0.34531837j -0.23086847+0.64387535j -0.12649219+0.14537382j]
 [-0.0672159 -0.15990479j -0.38856094+0.17298072j  0.88676154+0.05161895j]] [2.9
0297499 1.0846112 ] [[ 0.38416561+0.j         -0.78987852-0.47802584j]
 [ 0.9232642 +0.j          0.3286645 +0.19890416j]]

>>> m = np.array([[1.0, 0.0],
                  [2.0, 3.0],
                  [1.0, 1.0],
                  [0.0, 2.0],
                  [1.0, 0.0]])
>>> linalg.svdvals(m)
array([4.28091555, 1.63516424])
```

```
>>> A = np.array([[4, 3, 2], [-2, 2, 3], [3, -5, 2]])
>>> B = np.array([25, -10, -4])
>>> X = np.linalg.inv(A).dot(B)
>>> X
array([ 5.,  3., -2.])

>>> two_d_matrix = np.array([ [7, 9], [33, 8] ])
>>> print(linalg.det(two_d_matrix ))
-241.0

>>> a = np.array([[1,2,3], [4,5,6], [7,8,9]])
>>> linalg.det(a)
0.0

>>> a = np.array([[0,2,3], [4,5,6], [7,8,9]])
>>> linalg.det(a)
3.0

>>> A = np.array([[1,2],[3,4]])
>>> linalg.inv(A)
array([[-2. ,  1. ],
       [ 1.5, -0.5]])
>>> b = np.array([[5,6]])
>>> b.T
array([[5],
       [6]])
>>> A.dot(b.T)
array([[17],
       [39]])

>>> A = np.array([[1,0],[0,-2]])
>>> results = linalg.eig(A)
>>> print(results[0])
[ 1.+0.j -2.+0.j]
>>> print(results[1])
[[1. 0.]
 [0. 1.]]
```

```
>>> ab = np.array([[0,  0, -1, -1, -1],
...                [0,  2,  2,  2,  2],
...                [5,  4,  3,  2,  1],
...                [1,  1,  1,  1,  0]])
>>> b = np.array([0, 1, 2, 2, 3])
>>> linalg.solve_banded((1,2),ab,b)
array([-2.37288136,  3.93220339, -4.        ,  4.3559322 , -1.3559322 ])

>>> a = np.arange(9) - 4.0
>>> a
array([-4., -3., -2., -1.,  0.,  1.,  2.,  3.,  4.])
>>> b = a.reshape((3, 3))
>>> b
array([[-4., -3., -2.],
       [-1.,  0.,  1.],
       [ 2.,  3.,  4.]])
>>> linalg.norm(a)
7.745966692414834
>>> linalg.norm(b)
7.745966692414834

>>> a = np.array([[1, -1], [2, 4]])
>>> u, p =  linalg.polar(a)
>>> u
array([[ 0.85749293, -0.51449576],
       [ 0.51449576,  0.85749293]])
>>> p
array([[1.88648444, 1.2004901 ],
       [1.2004901 , 3.94446746]])

>>> a = np.array([[1.0, 3.0], [1.0, 4.0]])
>>> b = linalg.logm(a)
>>> b
array([[-1.02571087,  2.05142174],
       [ 0.68380725,  1.02571087]])
>>> linalg.expm(b)
array([[1., 3.],
       [1., 4.]])
```

```
>>> from scipy.linalg import expm, sinm, cosm
>>> a = np.array([[1.0, 2.0], [-1.0, 3.0]])
>>> expm(1j*a)
array([[ 0.4264593 +1.89217551j, -2.13721484-0.97811252j],
       [ 1.06860742+0.48905626j, -1.71075555+0.91406299j]])
>>> cosm(a) + 1j*sinm(a)
array([[ 0.4264593 +1.89217551j, -2.13721484-0.97811252j],
       [ 1.06860742+0.48905626j, -1.71075555+0.91406299j]])

>>> from scipy.linalg import sqrtm
>>> a = np.array([[1.0, 3.0], [1.0, 4.0]])
>>> r = sqrtm(a)
>>> r
array([[0.75592895, 1.13389342],
       [0.37796447, 1.88982237]])
>>> r.dot(r)
array([[1., 3.],
       [1., 4.]])

>>> from scipy.linalg import fractional_matrix_power
>>> a = np.array([[1.0, 3.0], [1.0, 4.0]])
>>> b = fractional_matrix_power(a, 0.5)
>>> b
array([[0.75592895, 1.13389342],
       [0.37796447, 1.88982237]])
>>> np.dot(b,b)
array([[1., 3.],
       [1., 4.]])

>>> from scipy.linalg import hankel
>>> hankel([1, 17, 99])
array([[ 1, 17, 99],
       [17, 99,  0],
       [99,  0,  0]])
>>> hankel([1,2,3,4], [4,7,7,8,9])
array([[1, 2, 3, 4, 7],
       [2, 3, 4, 7, 7],
       [3, 4, 7, 7, 8],
       [4, 7, 7, 8, 9]])
```

```
>>> import numpy as np
>>> from scipy.linalg import block_diag
>>> A = [[1, 0],
...      [0, 1]]
>>> B = [[3, 4, 5],
...      [6, 7, 8]]
>>> C = [[7]]
>>> P = np.zeros((2, 0), dtype='int32')
>>> block_diag(A, B, C)
array([[1, 0, 0, 0, 0, 0],
       [0, 1, 0, 0, 0, 0],
       [0, 0, 3, 4, 5, 0],
       [0, 0, 6, 7, 8, 0],
       [0, 0, 0, 0, 0, 7]], dtype=int32)

>>> block_diag(A, P, B, C)
array([[1, 0, 0, 0, 0, 0],
       [0, 1, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0],
       [0, 0, 3, 4, 5, 0],
       [0, 0, 6, 7, 8, 0],
       [0, 0, 0, 0, 0, 7]], dtype=int32)

>>> block_diag(1.0, [2, 3], [[4, 5], [6, 7]])
array([[1., 0., 0., 0., 0.],
       [0., 2., 3., 0., 0.],
       [0., 0., 0., 4., 5.],
       [0., 0., 0., 6., 7.]])

>>> from scipy.linalg import circulant
>>> circulant([1, 2, 3])
array([[1, 3, 2],
       [2, 1, 3],
       [3, 2, 1]])

>>> from scipy.linalg import companion
>>> companion([1, -10, 31, -30])
array([[ 10., -31.,  30.],
       [  1.,   0.,   0.],
       [  0.,   1.,   0.]])
```

```
>>> from scipy.linalg import convolution_matrix
>>> A = convolution_matrix([-1, 4, -2], 5, mode='same')
>>> A
array([[ 4, -1,  0,  0,  0],
       [-2,  4, -1,  0,  0],
       [ 0, -2,  4, -1,  0],
       [ 0,  0, -2,  4, -1],
       [ 0,  0,  0, -2,  4]])

>>> from scipy.linalg import helmert
>>> helmert(5, full=True)
array([[ 0.4472136 ,  0.4472136 ,  0.4472136 ,  0.4472136 ,  0.4472136 ],
       [ 0.70710678, -0.70710678,  0.        ,  0.        ,  0.        ],
       [ 0.40824829,  0.40824829, -0.81649658,  0.        ,  0.        ],
       [ 0.28867513,  0.28867513,  0.28867513, -0.8660254 ,  0.        ],
       [ 0.2236068 ,  0.2236068 ,  0.2236068 ,  0.2236068 , -0.89442719]])

>>> from scipy.linalg import hilbert
>>> hilbert(3)
array([[1.        , 0.5       , 0.33333333],
       [0.5       , 0.33333333, 0.25      ],
       [0.33333333, 0.25      , 0.2       ]])
>>> from scipy.linalg import leslie
>>> leslie([0.1, 2.0, 1.0, 0.1], [0.2, 0.8, 0.7])
array([[0.1, 2. , 1. , 0.1],
       [0.2, 0. , 0. , 0. ],
       [0. , 0.8, 0. , 0. ],
       [0. , 0. , 0.7, 0. ]])

>>> from scipy.linalg import toeplitz
>>> toeplitz([1,2,3], [1,4,5,6])
array([[1, 4, 5, 6],
       [2, 1, 4, 5],
       [3, 2, 1, 4]])
>>> toeplitz([1.0, 2+3j, 4-1j])
array([[1.+0.j, 2.-3.j, 4.+1.j],
       [2.+3.j, 1.+0.j, 2.-3.j],
       [4.-1.j, 2.+3.j, 1.+0.j]])
```

```
>>> from scipy.linalg import cholesky
>>> a = np.array([[1,-2j],[2j,5]])
>>> L = cholesky(a, lower=True)
>>> L
array([[1.+0.j, 0.+0.j],
       [0.+2.j, 1.+0.j]])
>>> L @ L.T.conj()
array([[1.+0.j, 0.-2.j],
       [0.+2.j, 5.+0.j]])

>>> from scipy.linalg import solve_circulant, solve, circulant, lstsq
>>> c = np.array([2, 2, 4])
>>> b = np.array([1, 2, 3])
>>> solve_circulant(c, b)
array([ 0.75, -0.25,  0.25])
>>> solve(circulant(c), b)
array([ 0.75, -0.25,  0.25])

>>> c = np.array([[1.5, 2, 3, 0, 0], [1, 1, 4, 3, 2]])
>>> b = np.arange(15).reshape(-1, 5)
>>> x = solve_circulant(c[:, np.newaxis, :], b, baxis=-1, outaxis=-1)
>>> x.shape
(2, 3, 5)
>>> np.set_printoptions(precision=3)
>>> x
array([[[-0.118,  0.22 ,  1.277, -0.142,  0.302],
        [ 0.651,  0.989,  2.046,  0.627,  1.072],
        [ 1.42 ,  1.758,  2.816,  1.396,  1.841]],

       [[ 0.401,  0.304,  0.694, -0.867,  0.377],
        [ 0.856,  0.758,  1.149, -0.412,  0.831],
        [ 1.31 ,  1.213,  1.603,  0.042,  1.286]]])
```

Conclusion: Hence, we have successfully learned and understood the concepts and basics of performing complex scientific linear algebra calculations in Python using Scipy and linalg. We learnt various ways to find determinant and inverse of a matrix, and perform singular value decomposition and find singular values of a matrix.