Experiment 13

Write python programs to implement Different Linear algebra functions using Scipy.

Roll No.	61
Name	V Krishnasubramaniam
Class	D10-A
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 python library that is useful in solving many mathematical equations and algorithms. It is designed on the top of Numpy library that gives more extension of finding scientific mathematical formulae like Matrix Rank, Inverse, polynomial equations, LU Decomposition, etc. Using its high-level functions will significantly reduce the complexity of the code and helps in better analyzing the data. SciPy is an interactive Python session used as a data-processing library that is made to compete with its rivalries such as MATLAB, Octave, R-Lab,etc. It has many user-friendly, efficient and easy-to-use functions that helps to solve problems like numerical integration, interpolation, optimization, linear algebra and statistics.

The benefit of using SciPy library in Python while making ML models is that it also makes a strong programming language available for use in developing fewer complex programs and applications.

SciPy contains varieties of sub packages which help to solve the most common issue related to Scientific Computation. SciPy package in Python is the most used Scientific library only second to GNU Scientific Library for C/C++ or Matlab's. Easy to use and understand as well as fast computational power. It can operate on an array of NumPy library.

Sub-packages of SciPy:

1. File input/output - scipy.io
2. Special Function - scipy.special
3. Linear Algebra Operation - scipy.linalg
4. Interpolation - scipy.interpolate
5. Optimization and fit - scipy.optimize
6. Statistics and random numbers - scipy.stats
7. Numerical Integration - scipy.integrate
8. Fast Fourier transforms - scipy.fftpack
9. Signal Processing - scipy.signal
10. Image manipulation – scipy.ndimage

Installation of Scipy

You can also install SciPy in Windows via pip:
```
pip install scipy
```

To install Scipy on Linux:
```
sudo apt-get install  python-scipy python-numpy
```

To install SciPy in Mac:

```
sudo port install py35-scipy py35-numpy
```

Linear Algebra with SciPy

Linear Algebra of SciPy is an implementation of BLAS and ATLAS LAPACK libraries. Performance of Linear Algebra is very fast compared to BLAS and LAPACK. All of these linear algebra routines expect an object that can be converted into a two-dimensional array. The output of these routines is also a two-dimensional array.

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.

Solving Linear Equations

The scipy.linalg.solve feature solves the linear equation a * x + b * y = Z, for the unknown x, y values. As an example, assume that it is desired to solve the following simultaneous equations.
$x + 3y + 5z = 10$
$2x + 5y + z = 8$
$2x + 3y + 8z = 3$

To solve the above equation for the x, y, z values, we can find the solution vector using a matrix inverse. However, 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.

Example:

```
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)
```

Output:

```
array([ 2., -2., 9.])
```

Finding Determinant of a matrix

The determinant of a square matrix A is often denoted as |A| and is a quantity often used in linear algebra. In SciPy, this is computed using the det() function. It takes a matrix as input and returns a scalar value.

Example:
```
from scipy import linalg
import numpy as np
A = np.array([[1,2],[3,4]])
x = linalg.det(A)
print(x)
```

Output:
```
-2.0
```

Finding Inverse of a Matrix

In SciPy, this is computed using the inv() function. It takes a matrix as input and returns a matrix.

Example:
```
a = np.array([[1., 2.], [3., 4.]])
linalg.inv(a)
```

Output:
```
array([[-2. ,  1. ],
       [ 1.5, -0.5]])
```

Eigenvalues and Eigenvectors

The eigenvalue-eigenvector problem is one of the most commonly employed linear algebra operations. We can find the Eigen values (λ) and the corresponding Eigen vectors (v) of a square matrix (A) by considering the following relation −
$Av = \lambda v$

scipy.linalg.eig computes the eigenvalues from an ordinary or generalized eigenvalue problem. This function returns the Eigen values and the Eigen vectors.

Example:
```
from scipy import linalg
import numpy as np
A = np.array([[1,2],[3,4]])
l, v = linalg.eig(A)
print(l)
print(v)
```

Output:
```
array([-0.37228132+0.j, 5.37228132+0.j])
array([[-0.82456484, -0.41597356],
       [ 0.56576746, -0.90937671]])
```

Singular 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’.

Example:
```
from scipy import linalg
import numpy as np
a = np.random.randn(3, 2) + 1.j*np.random.randn(3, 2)
U, s, Vh = linalg.svd(a)
print(U, Vh, s)
```

Output:
```
(
  array([
    [ 0.54828424-0.23329795j, -0.38465728+0.01566714j,
    -0.18764355+0.67936712j],
    [-0.27123194-0.5327436j , -0.57080163-0.00266155j,
    -0.39868941-0.39729416j],
    [ 0.34443818+0.4110186j , -0.47972716+0.54390586j,
    0.25028608-0.35186815j]
  ]),
  array([ 3.25745379, 1.16150607]),
  array([
    [-0.35312444+0.j , 0.32400401+0.87768134j],
    [-0.93557636+0.j , -0.12229224-0.33127251j]
  ])
```

Singular Values of a Matrix

Singular values of a matrix of a matrix can be found out by using the functions svdvals(). svdvals(a) only differs from svd(a, compute_uv=False) by its handling of the edge case of empty a, where it returns an empty sequence:

Example:
```
from scipy.linalg import svdvals
m = np.array([[1.0, 0.0],
         [2.0, 3.0],
         [1.0, 1.0],
         [0.0, 2.0],
         [1.0, 0.0]])
svdvals(m)
```

Output:
```
array([ 4.28091555,  1.63516424])
```

Other useful functions of scipy.linalg:

Basics:

inv(a[, overwrite_a, check_finite])	Compute the inverse of a matrix.
solve(a, b[, sym_pos, lower, overwrite_a, …])	Solves the linear equation set a * x = b for the unknown x for square a matrix.
solve_banded(l_and_u, ab, b[, overwrite_ab, …])	Solve the equation a x = b for x, assuming a is 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.
solve_toeplitz(c_or_cr, b[, check_finite])	Solve a Toeplitz system using Levinson Recursion
matmul_toeplitz(c_or_cr, x[, check_finite, …])	Efficient Toeplitz Matrix-Matrix Multiplication using FFT
det(a[, overwrite_a, check_finite])	Compute the determinant of a matrix
norm(a[, ord, axis, keepdims, check_finite])	Matrix or vector norm.
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.
pinv2(a[, cond, rcond, return_rank, …])	Compute the (Moore-Penrose) pseudo-inverse of a matrix.
pinvh(a[, cond, rcond, lower, return_rank, …])	Compute the (Moore-Penrose) pseudo-inverse of a Hermitian matrix.
kron(a, b)	Kronecker product.
khatri_rao(a, b)	Khatri-rao product
tril(m[, k])	Make a copy of a matrix with elements above the kth diagonal zeroed.

triu(m[, k])	Make a copy of a matrix with elements below the kth diagonal zeroed.
orthogonal_procrustes(A, B[, check_finite])	Compute the matrix solution of the orthogonal Procrustes problem.
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.
LinAlgError	Generic Python-exception-derived object raised by linalg functions.
LinAlgWarning	The warning emitted when a linear algebra related operation is close to fail conditions of the algorithm or loss of accuracy is expected.

Eigenvalue Problems

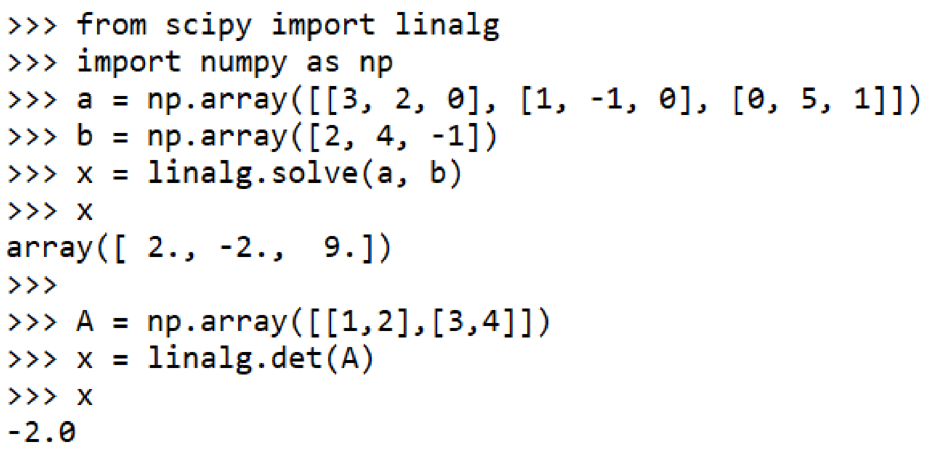
eig(a[, b, left, right, overwrite_a, …])	Solve an ordinary or generalized eigenvalue problem of a square matrix.
eigvals(a[, b, overwrite_a, check_finite, …])	Compute eigenvalues from an ordinary or generalized eigenvalue problem.
eigh(a[, b, lower, eigvals_only, …])	Solve a standard or generalized eigenvalue problem for a complex Hermitian or real symmetric matrix.
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.

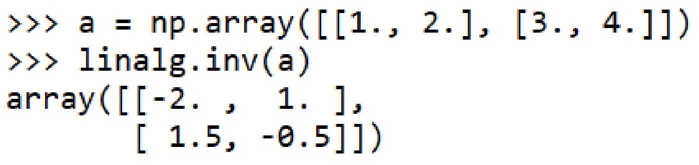
Matrix Functions

expm(A)	Compute the matrix exponential using Pade approximation.
logm(A[, disp])	Compute matrix logarithm.
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.
sqrtm(A[, disp, blocksize])	Matrix square root.
funm(A, func[, disp])	Evaluate a matrix function specified by a callable.
expm_frechet(A, E[, method, compute_expm, …])	Frechet derivative of the matrix exponential of A in the direction E.
expm_cond(A[, check_finite])	Relative condition number of the matrix exponential in the Frobenius norm.
fractional_matrix_power(A, t)	Compute the fractional power of a matrix.

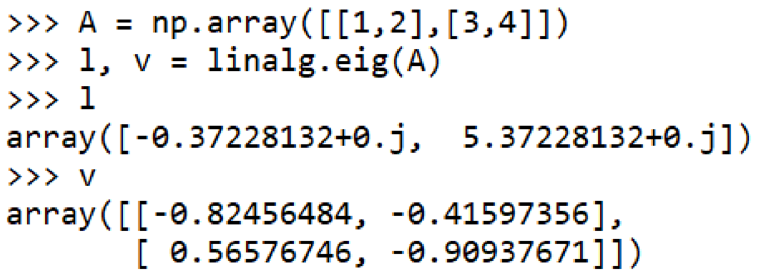
Results:



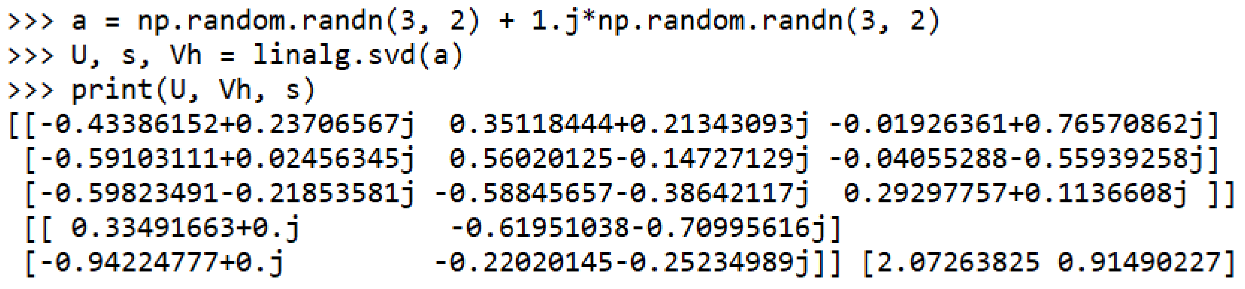
```
>>> from scipy import linalg
>>> import numpy as np
>>> a = np.array([[3, 2, 0], [1, -1, 0], [0, 5, 1]])
>>> b = np.array([2, 4, -1])
>>> x = linalg.solve(a, b)
>>> x
array([ 2., -2.,  9.])
>>>
>>> A = np.array([[1,2],[3,4]])
>>> x = linalg.det(A)
>>> x
-2.0
```



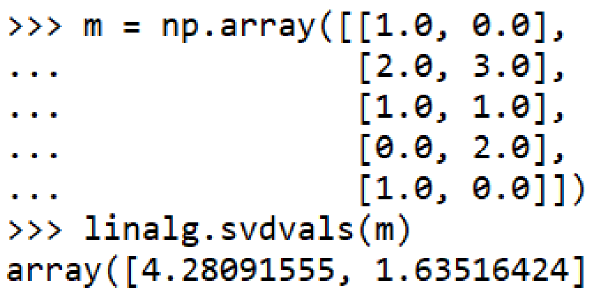
```
>>> a = np.array([[1., 2.], [3., 4.]])
>>> linalg.inv(a)
array([[-2. ,  1. ],
       [ 1.5, -0.5]])
```



```
>>> A = np.array([[1,2],[3,4]])
>>> l, v = linalg.eig(A)
>>> l
array([-0.37228132+0.j,  5.37228132+0.j])
>>> v
array([[-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, Vh, s)
[[-0.43386152+0.23706567j  0.35118444+0.21343093j -0.01926361+0.76570862j]
 [-0.59103111+0.02456345j  0.56020125-0.14727129j -0.04055288-0.55939258j]
 [-0.59823491-0.21853581j -0.58845657-0.38642117j  0.29297757+0.1136608j ]]
 [[ 0.33491663+0.j          -0.61951038-0.70995616j]
 [-0.94224777+0.j          -0.22020145-0.25234989j]] [2.07263825 0.91490227]
```



```
>>> 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.]]
```

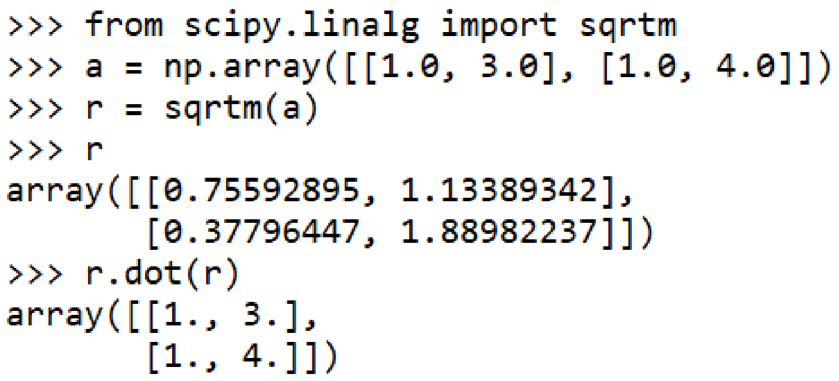
```
>>> ab = np.array([[0,  0, -1, -1, -1],
...                [0,  2,  2,  2,  2],
...                [5,  4,  3,  2,  1],
...                [1,  1,  1,  1,  0]])
>>> b = np.array([0, 1, 2, 2, 3])
>>> linalg.solve_banded((1,2),ab,b)
array([-2.37288136,  3.93220339, -4.        ,  4.3559322 , -1.3559322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

```
>>> 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
```

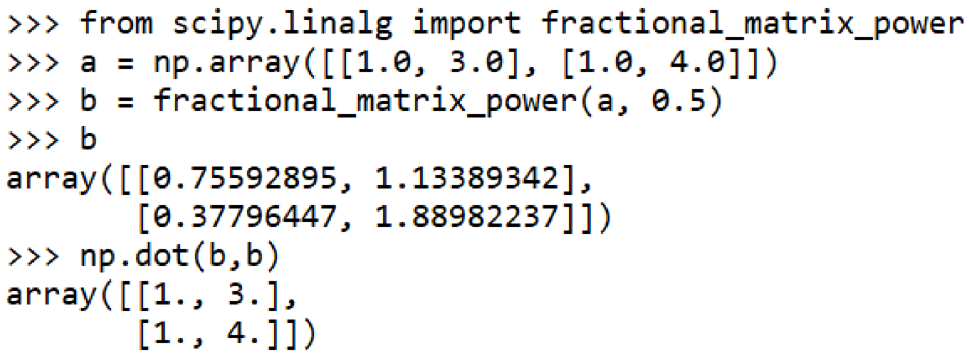
```
>>> a = np.array([[1, -1], [2, 4]])
>>> u, p =  linalg.polar(a)
>>> u
array([[ 0.85749293, -0.51449576],
       [ 0.51449576,  0.85749293]])
>>> p
array([[1.88648444, 1.2004901 ],
       [1.2004901 , 3.94446746]])
```

```
>>> 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.]])
```

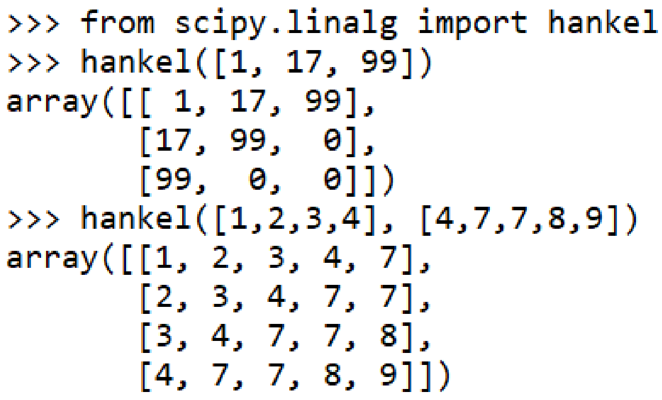
```
>>> from scipy.linalg import expm, sinm, cosm
>>> a = np.array([[1.0, 2.0], [-1.0, 3.0]])
>>> expm(1j*a)
array([[ 0.4264593 +1.89217551j, -2.13721484-0.97811252j],
       [ 1.06860742+0.48905626j, -1.71075555+0.91406299j]])
>>> cosm(a) + 1j*sinm(a)
array([[ 0.4264593 +1.89217551j, -2.13721484-0.97811252j],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

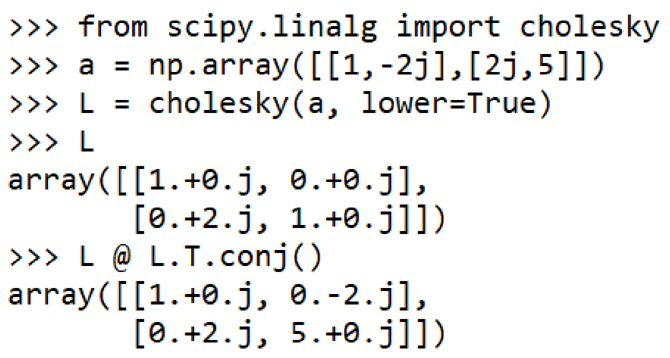
```
>>> import numpy as np
>>> from scipy.linalg import block_diag
>>> A = [[1, 0],
...      [0, 1]]
>>> B = [[3, 4, 5],
...      [6, 7, 8]]
>>> C = [[7]]
>>> P = np.zeros((2, 0), dtype='int32')
>>> block_diag(A, B, C)
array([[1, 0, 0, 0, 0, 0],
       [0, 1, 0, 0, 0, 0],
       [0, 0, 3, 4, 5, 0],
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

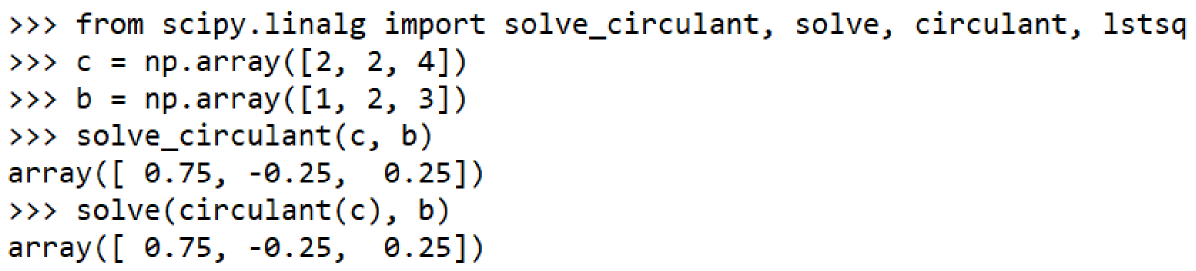
```
>>> from scipy.linalg import helmert
>>> helmert(5, full=True)
array([[ 0.4472136 ,  0.4472136 ,  0.4472136 ,  0.4472136 ,  0.4472136 ],
       [ 0.70710678, -0.70710678,  0.        ,  0.        ,  0.        ],
       [ 0.40824829,  0.40824829, -0.81649658,  0.        ,  0.        ],
       [ 0.28867513,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

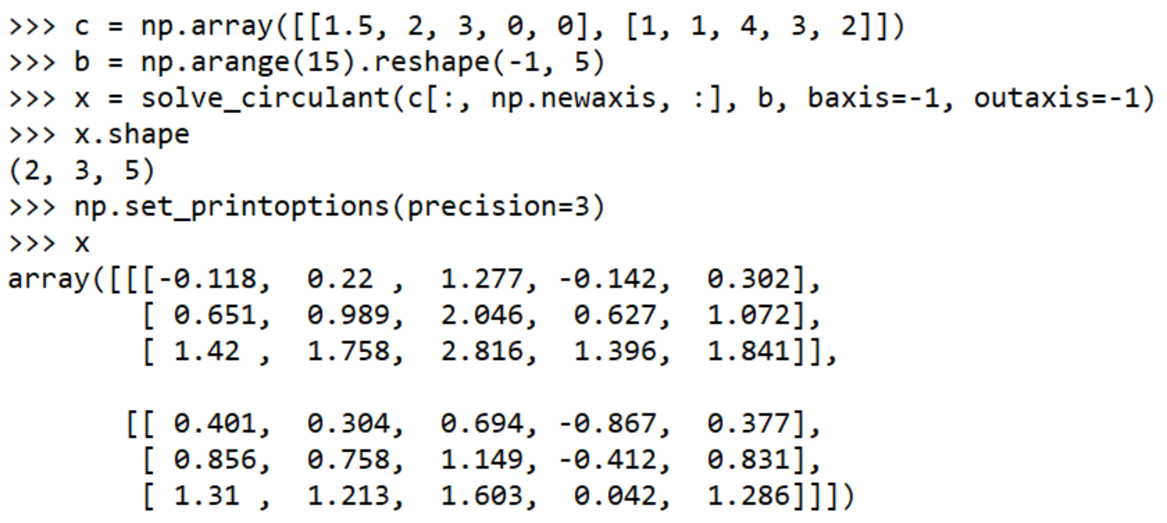
```
>>> from scipy.linalg import toeplitz
>>> toeplitz([1,2,3], [1,4,5,6])
array([[1, 4, 5, 6],
       [2, 1, 4, 5],
       [3, 2, 1, 4]])
>>> toeplitz([1.0, 2+3j, 4-1j])
array([[1.+0.j, 2.-3.j, 4.+1.j],
       [2.+3.j, 1.+0.j, 2.-3.j],
       [4.-1.j, 2.+3.j, 1.+0.j]])
```



```
>>> 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:

Thus, we have understood the 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, by performing hands-on practical programs for each.