3D Computer Vision HW0

Songmin Lee

March 2023

1 SVD

1.1 Brief Description of SVD

SVD: a method of decomposing any given matrix into a product of unitary matrices and the eigenvalue matrix. Input = an m x n matrix M.

Output $= USV$, where S is the eigenvalue matrix with singular values along the diagonal, and U, V are unitary matrices with eigenvectors of MM^t and M^tM as columns, respectively.

Basic properties = can decompose any matrix, even non-square matrices and complex matrices, into a combination of special matrices. Applications in calculating pseudo-inverse, solving linear systems of equations, and data compression.

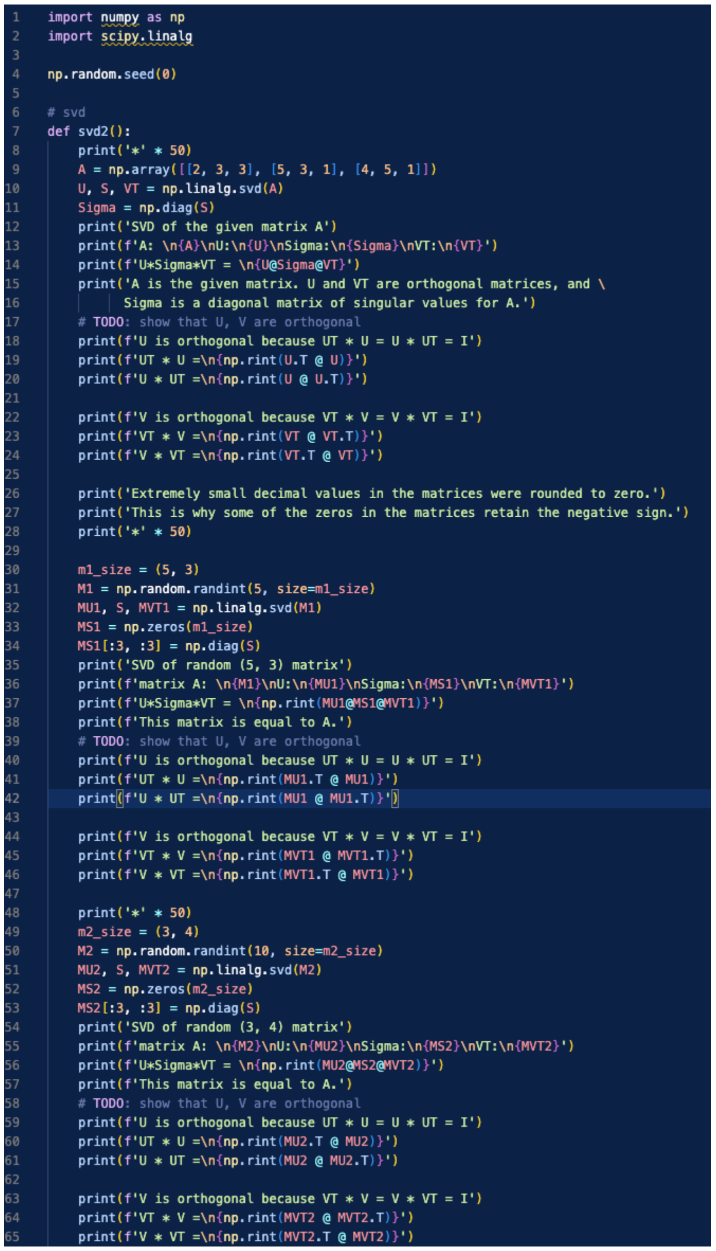
1.2 SVD and Eigenvalue Decomposition

If an m x m matrix M has an inverse, then applying SVD to it produces the result USV, where $V = U^{-1}$, S is the diagonal matrix of eigenvalues, and U is the eigenvector matrix (the result of eigenvalue decomposition). The idea is that SVD generalizes the concept of eigenvalue decomposition of a square matrix to all kinds of matrices, in which U, S, V serve similar roles.

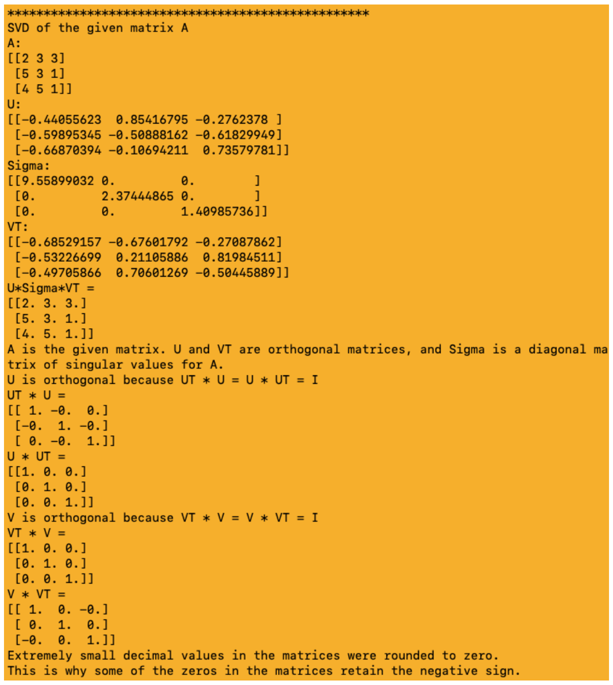
1.3 PCA and SVD

PCA and SVD are both methods that reduce the dimensions of data, by finding linear combinations of features. PCA finds linearly uncorrelated orthogonal axes called principal components (PCs) and projects the data points onto those PCs. Both PCA and SVD involve finding the eigen-decomposition of a given matrix. PCA decomposes an invertible covariance matrix C as the expression WSW^-1, where W is the matrix of eigenvector columns. PCA decomposition can also be turned into SVD decomposition, where the matrix S from PCA equals the expression $D^2/(n-1)$, where D is the eigenvalue matrix derived from SVD.

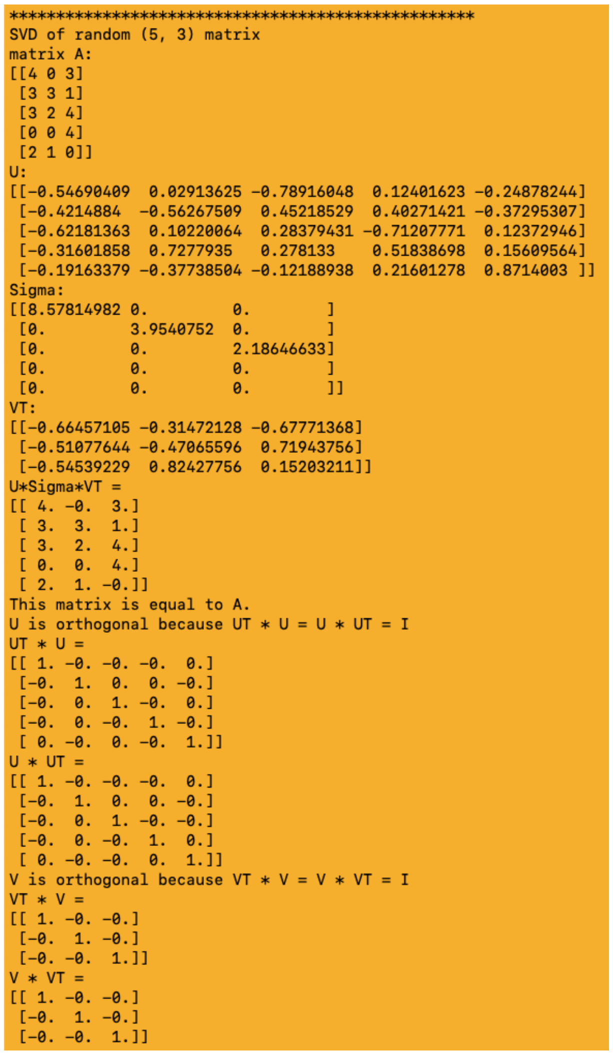
1.4 SVD Code and Results



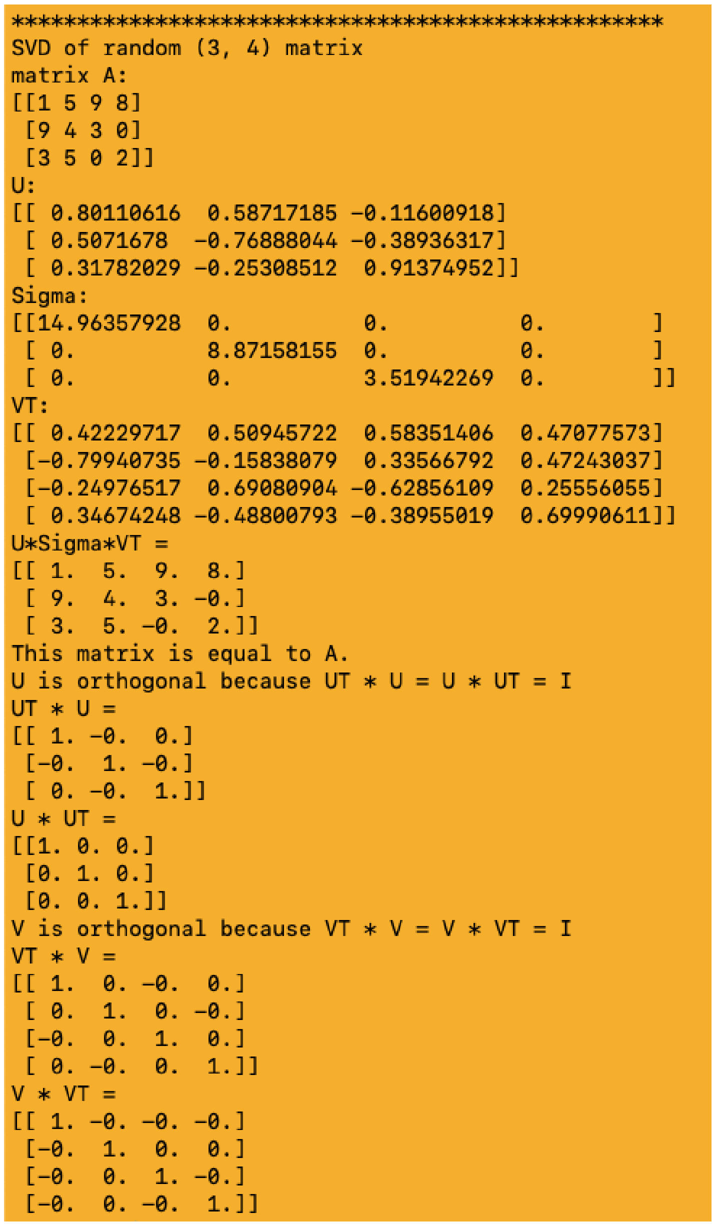
```python
import numpy as np
import scipy.linalg

np.random.seed(0)

# svd
def svd2():
    print('*' * 50)
    A = np.array([[2, 3, 3], [5, 3, 1], [4, 5, 1]])
    U, S, VT = np.linalg.svd(A)
    Sigma = np.diag(S)
    print('SVD of the given matrix A')
    print(f'A: \n{A}\nU:\n{U}\nSigma:\n{Sigma}\nVT:\n{VT}')
    print(f'U*Sigma*VT = \n{U@Sigma@VT}')
    print('A is the given matrix. U and VT are orthogonal matrices, and \
            Sigma is a diagonal matrix of singular values for A.')
    # TODO: show that U, V are orthogonal
    print(f'U is orthogonal because UT * U = U * UT = I')
    print(f'UT * U =\n{np.rint(U.T @ U)}')
    print(f'U * UT =\n{np.rint(U @ U.T)}')

    print(f'V is orthogonal because VT * V = V * VT = I')
    print(f'VT * V =\n{np.rint(VT @ VT.T)}')
    print(f'V * VT =\n{np.rint(VT.T @ VT)}')

    print('Extremely small decimal values in the matrices were rounded to zero.')
    print('This is why some of the zeros in the matrices retain the negative sign.')
    print('*' * 50)

    m1_size = (5, 3)
    M1 = np.random.randint(5, size=m1_size)
    MU1, S, MVT1 = np.linalg.svd(M1)
    MS1 = np.zeros(m1_size)
    MS1[:3, :3] = np.diag(S)
    print('SVD of random (5, 3) matrix')
    print(f'matrix A: \n{M1}\nU:\n{MU1}\nSigma:\n{MS1}\nVT:\n{MVT1}')
    print(f'U*Sigma*VT = \n{np.rint(MU1@MS1@MVT1)}')
    print(f'This matrix is equal to A.')
    # TODO: show that U, V are orthogonal
    print(f'U is orthogonal because UT * U = U * UT = I')
    print(f'UT * U =\n{np.rint(MU1.T @ MU1)}')
    print(f'U * UT =\n{np.rint(MU1 @ MU1.T)}')

    print(f'V is orthogonal because VT * V = V * VT = I')
    print(f'VT * V =\n{np.rint(MVT1 @ MVT1.T)}')
    print(f'V * VT =\n{np.rint(MVT1.T @ MVT1)}')

    print('*' * 50)
    m2_size = (3, 4)
    M2 = np.random.randint(10, size=m2_size)
    MU2, S, MVT2 = np.linalg.svd(M2)
    MS2 = np.zeros(m2_size)
    MS2[:3, :3] = np.diag(S)
    print('SVD of random (3, 4) matrix')
    print(f'matrix A: \n{M2}\nU:\n{MU2}\nSigma:\n{MS2}\nVT:\n{MVT2}')
    print(f'U*Sigma*VT = \n{np.rint(MU2@MS2@MVT2)}')
    print(f'This matrix is equal to A.')
    # TODO: show that U, V are orthogonal
    print(f'U is orthogonal because UT * U = U * UT = I')
    print(f'UT * U =\n{np.rint(MU2.T @ MU2)}')
    print(f'U * UT =\n{np.rint(MU2 @ MU2.T)}')

    print(f'V is orthogonal because VT * V = V * VT = I')
    print(f'VT * V =\n{np.rint(MVT2 @ MVT2.T)}')
    print(f'V * VT =\n{np.rint(MVT2.T @ MVT2)}')
```



```
**************************************************
SVD of the given matrix A
A:
[[2 3 3]
 [5 3 1]
 [4 5 1]]
U:
[[-0.44055623  0.85416795 -0.2762378 ]
 [-0.59895345 -0.50888162 -0.61829949]
 [-0.66870394 -0.10694211  0.73579781]]
Sigma:
[[9.55899032 0.         0.        ]
 [0.         2.37444865 0.        ]
 [0.         0.         1.40985736]]
VT:
[[-0.68529157 -0.67601792 -0.27087862]
 [-0.53226699  0.21105886  0.81984511]
 [-0.49705866  0.70601269 -0.50445889]]
U*Sigma*VT =
[[2. 3. 3.]
 [5. 3. 1.]
 [4. 5. 1.]]
A is the given matrix. U and VT are orthogonal matrices, and Sigma is a diagonal ma
trix of singular values for A.
U is orthogonal because UT * U = U * UT = I
UT * U =
[[ 1. -0.  0.]
 [-0.  1. -0.]
 [ 0. -0.  1.]]
U * UT =
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
V is orthogonal because VT * V = V * VT = I
VT * V =
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
V * VT =
[[ 1.  0. -0.]
 [ 0.  1.  0.]
 [-0.  0.  1.]]
Extremely small decimal values in the matrices were rounded to zero.
This is why some of the zeros in the matrices retain the negative sign.
```



```
**************************************************
SVD of random (5, 3) matrix
matrix A:
[[4 0 3]
 [3 3 1]
 [3 2 4]
 [0 0 4]
 [2 1 0]]
U:
[[-0.54690409  0.02913625 -0.78916048  0.12401623 -0.24878244]
 [-0.4214884  -0.56267509  0.45218529  0.40271421 -0.37295307]
 [-0.62181363  0.10220064  0.28379431 -0.71207771  0.12372946]
 [-0.31601858  0.7277935   0.278133    0.51838698  0.15609564]
 [-0.19163379 -0.37738504 -0.12188938  0.21601278  0.8714003 ]]
Sigma:
[[8.57814982 0.         0.        ]
 [0.         3.9540752  0.        ]
 [0.         0.         2.18646633]
 [0.         0.         0.        ]
 [0.         0.         0.        ]]
VT:
[[-0.66457105 -0.31472128 -0.67771368]
 [-0.51077644 -0.47065596  0.71943756]
 [-0.54539229  0.82427756  0.15203211]]
U*Sigma*VT =
[[ 4. -0.  3.]
 [ 3.  3.  1.]
 [ 3.  2.  4.]
 [ 0.  0.  4.]
 [ 2.  1. -0.]]
This matrix is equal to A.
U is orthogonal because UT * U = U * UT = I
UT * U =
[[ 1. -0. -0. -0.  0.]
 [-0.  1.  0.  0. -0.]
 [-0.  0.  1. -0.  0.]
 [-0.  0. -0.  1. -0.]
 [ 0. -0.  0. -0.  1.]]
U * UT =
[[ 1. -0. -0. -0.  0.]
 [-0.  1.  0.  0. -0.]
 [-0.  0.  1. -0. -0.]
 [-0.  0. -0.  1.  0.]
 [ 0. -0. -0.  0.  1.]]
V is orthogonal because VT * V = V * VT = I
VT * V =
[[ 1. -0. -0.]
 [-0.  1. -0.]
 [-0. -0.  1.]]
V * VT =
[[ 1. -0. -0.]
 [-0.  1. -0.]
 [-0. -0.  1.]]
```

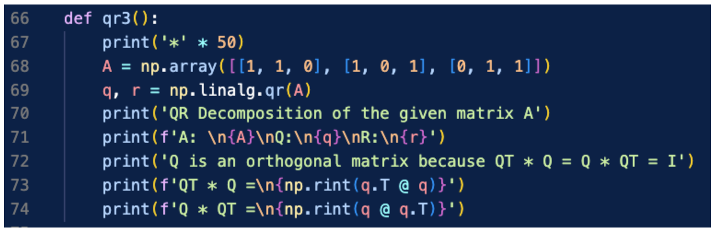


```
**************************************************
SVD of random (3, 4) matrix
matrix A:
[[1 5 9 8]
 [9 4 3 0]
 [3 5 0 2]]
U:
[[ 0.80110616  0.58717185 -0.11600918]
 [ 0.5071678  -0.76888044 -0.38936317]
 [ 0.31782029 -0.25308512  0.91374952]]
Sigma:
[[14.96357928  0.          0.          0.        ]
 [ 0.          8.87158155  0.          0.        ]
 [ 0.          0.          3.51942269  0.        ]]
VT:
[[ 0.42229717  0.50945722  0.58351406  0.47077573]
 [-0.79940735 -0.15838079  0.33566792  0.47243037]
 [-0.24976517  0.69080904 -0.62856109  0.25556055]
 [ 0.34674248 -0.48800793 -0.38955019  0.69990611]]
U*Sigma*VT =
[[ 1.  5.  9.  8.]
 [ 9.  4.  3. -0.]
 [ 3.  5. -0.  2.]]
This matrix is equal to A.
U is orthogonal because UT * U = U * UT = I
UT * U =
[[ 1. -0.  0.]
 [-0.  1. -0.]
 [ 0. -0.  1.]]
U * UT =
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
V is orthogonal because VT * V = V * VT = I
VT * V =
[[ 1.  0. -0.  0.]
 [ 0.  1.  0. -0.]
 [-0.  0.  1.  0.]
 [ 0. -0.  0.  1.]]
V * VT =
[[ 1. -0. -0. -0.]
 [-0.  1.  0.  0.]
 [-0.  0.  1. -0.]
 [-0.  0. -0.  1.]]
```

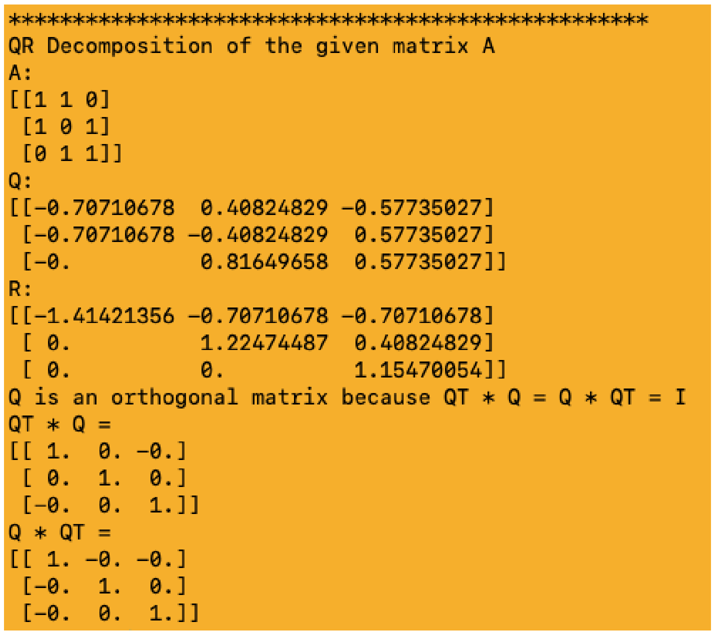
2 QR Decomposition

QR decomposition: the matrix version of the Gram-Schmidt orthonormalization process. Also a method that decomposes any given matrix into two special matrices. inputs: any matrix A (can be non-square)
outputs: Q and R where Q has orthonormal columns and R is upper triangular with positive diagonal entries
basic properties: QR for a matrix is unique. Overall complexity of the algorithm is $O(n^3)$.
applications: used for solving linear systems, eigenvalue problems, and least squares approximations. Makes these processes more efficient as the method involves special matrices.
RQ decomposition: similar but transforms a matrix into the product of an upper triangular matrix R and an orthogonal matrix Q. Same as QR decomposition, but carries out the Gram-Schmidt orthogonalization of rows of the matrix A.
Modify QR to RQ by inputting the transpose of A.

2.1 QR Decomposition Code and Results



```python
def qr3():
    print('*' * 50)
    A = np.array([[1, 1, 0], [1, 0, 1], [0, 1, 1]])
    q, r = np.linalg.qr(A)
    print('QR Decomposition of the given matrix A')
    print(f'A: \n{A}\nQ:\n{q}\nR:\n{r}')
    print('Q is an orthogonal matrix because QT * Q = Q * QT = I')
    print(f'QT * Q =\n{np.rint(q.T @ q)}')
    print(f'Q * QT =\n{np.rint(q @ q.T)}')
```

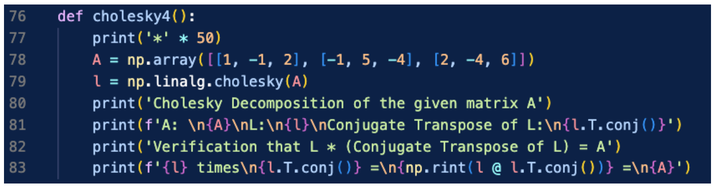


```
**************************************************
QR Decomposition of the given matrix A
A:
[[1 1 0]
 [1 0 1]
 [0 1 1]]
Q:
[[-0.70710678  0.40824829 -0.57735027]
 [-0.70710678 -0.40824829  0.57735027]
 [-0.          0.81649658  0.57735027]]
R:
[[-1.41421356 -0.70710678 -0.70710678]
 [ 0.          1.22474487  0.40824829]
 [ 0.          0.          1.15470054]]
Q is an orthogonal matrix because QT * Q = Q * QT = I
QT * Q =
[[ 1.  0. -0.]
 [ 0.  1.  0.]
 [-0.  0.  1.]]
Q * QT =
[[ 1. -0. -0.]
 [-0.  1.  0.]
 [-0.  0.  1.]]
```

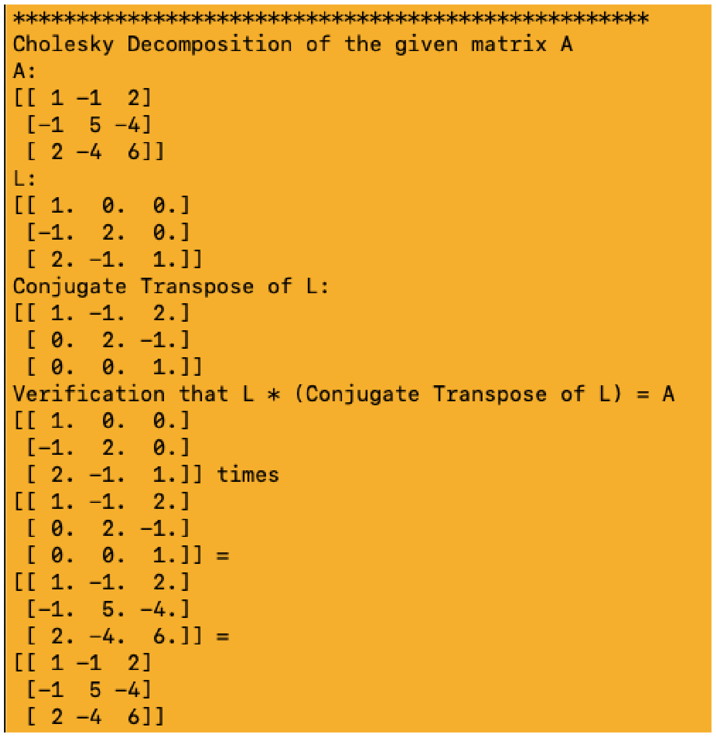
3 Cholesky Decomposition

Cholesky decomposition: works for Hermitian, positive-definite matrices. Decomposes such matrices into a product of lower triangular matrix and conjugate transpose.
inputs: Hermitian positive-definite matrix.
outputs: L, L^*, where L = lower triangular matrix with real and positive diagonal entries. L^* is the conjugate transpose of L.
basic properties: the converse is true. If A can be decomposed into the mentioned form, then it is Hermitian positive-definite. Method is very efficient.

3.1 Cholesky Code and Results



```python
def cholesky4():
    print('*' * 50)
    A = np.array([[1, -1, 2], [-1, 5, -4], [2, -4, 6]])
    l = np.linalg.cholesky(A)
    print('Cholesky Decomposition of the given matrix A')
    print(f'A: \n{A}\nL:\n{l}\nConjugate Transpose of L:\n{l.T.conj()}')
    print('Verification that L * (Conjugate Transpose of L) = A')
    print(f'{l} times\n{l.T.conj()} =\n{np.rint(l @ l.T.conj())} =\n{A}')
```

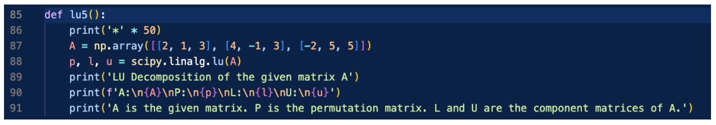


```
**************************************************
Cholesky Decomposition of the given matrix A
A:
[[ 1 -1  2]
 [-1  5 -4]
 [ 2 -4  6]]
L:
[[ 1.  0.  0.]
 [-1.  2.  0.]
 [ 2. -1.  1.]]
Conjugate Transpose of L:
[[ 1. -1.  2.]
 [ 0.  2. -1.]
 [ 0.  0.  1.]]
Verification that L * (Conjugate Transpose of L) = A
[[ 1.  0.  0.]
 [-1.  2.  0.]
 [ 2. -1.  1.]] times
[[ 1. -1.  2.]
 [ 0.  2. -1.]
 [ 0.  0.  1.]] =
[[ 1. -1.  2.]
 [-1.  5. -4.]
 [ 2. -4.  6.]] =
[[ 1 -1  2]
 [-1  5 -4]
 [ 2 -4  6]]
```

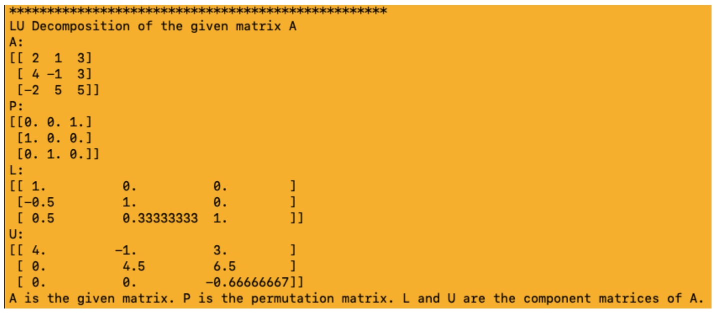
4 LU Decomposition

LU decomposition: a method that decomposes a matrix into a lower triangular matrix L and an upper triangular matrix U.
inputs: any matrix
outputs: lower triangular matrix L and an upper triangular matrix U
basic properties: the matrix UL^T is also an upper triangular matrix.

4.1 LU Decomposition Code and Results



```python
def lu5():
    print('*' * 50)
    A = np.array([[2, 1, 3], [4, -1, 3], [-2, 5, 5]])
    p, l, u = scipy.linalg.lu(A)
    print('LU Decomposition of the given matrix A')
    print(f'A:\n{A}\nP:\n{p}\nL:\n{l}\nU:\n{u}')
    print('A is the given matrix. P is the permutation matrix. L and U are the component matrices of A.')
```



```
**************************************************
LU Decomposition of the given matrix A
A:
[[ 2  1  3]
 [ 4 -1  3]
 [-2  5  5]]
P:
[[0. 0. 1.]
 [1. 0. 0.]
 [0. 1. 0.]]
L:
[[ 1.          0.          0.        ]
 [-0.5         1.          0.        ]
 [ 0.5         0.33333333  1.        ]]
U:
[[ 4.         -1.          3.        ]
 [ 0.          4.5         6.5       ]
 [ 0.          0.         -0.66666667]]
A is the given matrix. P is the permutation matrix. L and U are the component matrices of A.
```

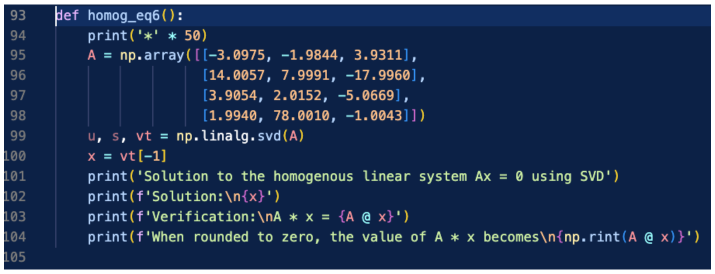
5 Ax = 0

Meaning of null space of a matrix: the set of all solutions to the homogenous equation Ax = 0

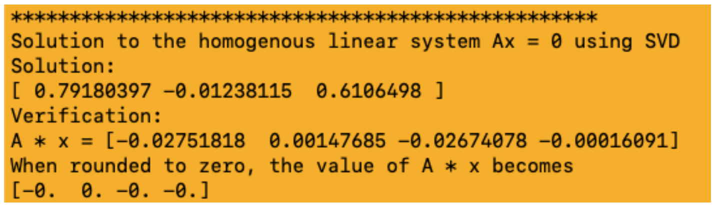
Full-rank definition: the columns of A are all linearly-independent and A has a null space consisting of only the trivial solution 0.
Matrix A has a non-trivial null space when it is not full rank.

SVD can be used to solve the linear system Ax = 0. With SVD, $A = USV^T$. SVD is not unique, but it is always possible to find a set of U, S, V such that S is a diagonal matrix with singular values on the diagonal in descending order, and V is a matrix with eigenvectors as columns. In particular, in V, the i_{th} column is the eigenvector that corresponds to the i_{th} singular value in S. This eigenvector minimizes the difference between A and 0. Hence, the final column of V is the solution to Ax = 0. U, S, $V^T = svd(A)$
$x = k(V^T)$
, where k is a function that finds the final row of V^T, or the final column of V.

5.1 Code and Results for Ax = 0



```
def homog_eq6():
    print('*' * 50)
    A = np.array([[-3.0975, -1.9844, 3.9311],
                  [14.0057, 7.9991, -17.9960],
                  [3.9054, 2.0152, -5.0669],
                  [1.9940, 78.0010, -1.0043]])
    u, s, vt = np.linalg.svd(A)
    x = vt[-1]
    print('Solution to the homogenous linear system Ax = 0 using SVD')
    print(f'Solution:\n{x}')
    print(f'Verification:\nA * x = {A @ x}')
    print(f'When rounded to zero, the value of A * x becomes\n{np.rint(A @ x)}')
```

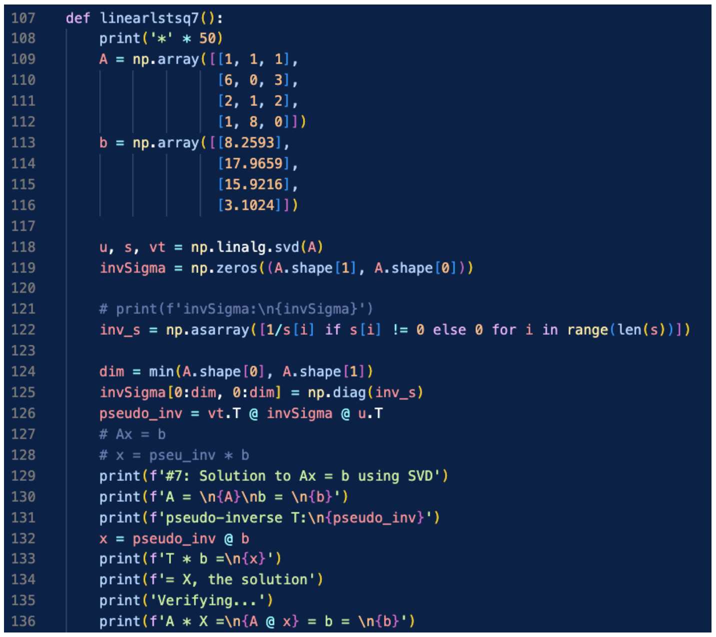


```
**************************************************
Solution to the homogenous linear system Ax = 0 using SVD
Solution:
[ 0.79180397 -0.01238115  0.6106498 ]
Verification:
A * x = [-0.02751818  0.00147685 -0.02674078 -0.00016091]
When rounded to zero, the value of A * x becomes
[-0.  0. -0. -0.]
```

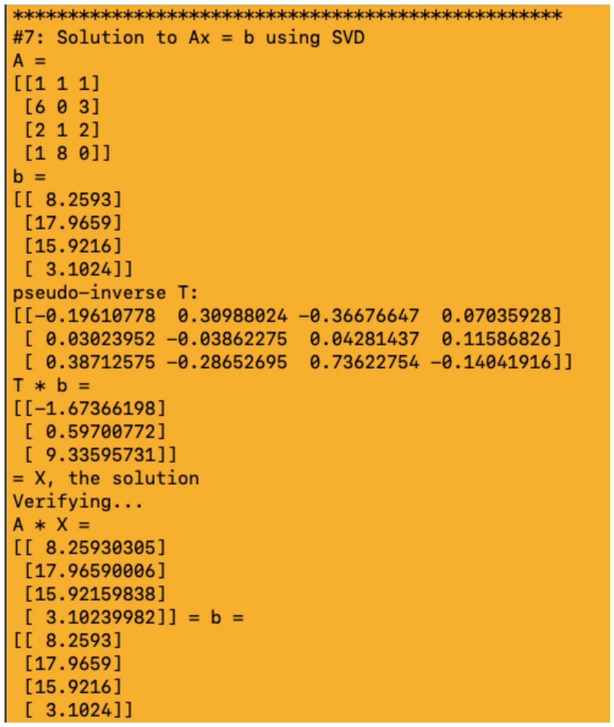
6 Ax = b

Computing the pseudo-inverse of a matrix using SVD: given a matrix A, the SVD is USV^T. The pseudo-inverse A^* of the matrix is the matrix that minimizes the difference between A^*A and I. This is equal to VS^*U^T, where S^* is the diagonal matrix of reciprocals of non-zero entries on S, or zero entries at the positions for zero diagonal entries on S.

6.1 Code and Results for Ax = b



```
def linearlstsq7():
    print('*' * 50)
    A = np.array([[1, 1, 1],
                  [6, 0, 3],
                  [2, 1, 2],
                  [1, 8, 0]])
    b = np.array([[8.2593],
                  [17.9659],
                  [15.9216],
                  [3.1024]])

    u, s, vt = np.linalg.svd(A)
    invSigma = np.zeros((A.shape[1], A.shape[0]))

    # print(f'invSigma:\n{invSigma}')
    inv_s = np.asarray([1/s[i] if s[i] != 0 else 0 for i in range(len(s))])

    dim = min(A.shape[0], A.shape[1])
    invSigma[0:dim, 0:dim] = np.diag(inv_s)
    pseudo_inv = vt.T @ invSigma @ u.T
    # Ax = b
    # x = pseu_inv * b
    print(f'#7: Solution to Ax = b using SVD')
    print(f'A = \n{A}\nb = \n{b}')
    print(f'pseudo-inverse T:\n{pseudo_inv}')
    x = pseudo_inv @ b
    print(f'T * b =\n{x}')
    print(f'= X, the solution')
    print('Verifying...')
    print(f'A * X =\n{A @ x} = b = \n{b}')
```



```
**************************************************
#7: Solution to Ax = b using SVD
A =
[[1 1 1]
 [6 0 3]
 [2 1 2]
 [1 8 0]]
b =
[[ 8.2593]
 [17.9659]
 [15.9216]
 [ 3.1024]]
pseudo-inverse T:
[[-0.19610778  0.30988024 -0.36676647  0.07035928]
 [ 0.03023952 -0.03862275  0.04281437  0.11586826]
 [ 0.38712575 -0.28652695  0.73622754 -0.14041916]]
T * b =
[[-1.67366198]
 [ 0.59700772]
 [ 9.33595731]]
= X, the solution
Verifying...
A * X =
[[ 8.25930305]
 [17.96590006]
 [15.92159838]
 [ 3.10239982]] = b =
[[ 8.2593]
 [17.9659]
 [15.9216]
 [ 3.1024]]
```

7 Levenberg-Marquardt Algorithm

LM algorithm: damped least-squares method used to solve non-linear least squares problems. Applications in fitting curves that best model some data.
Comparison between LM and gradient descent/Gauss-Newton algorithm: LM is a mix of Gauss-Newton algorithm (GNA) and gradient descent, but it is more generalizable to data than GNA, despite being slower. It also tends to be more accurate than gradient descent.