1.(a)

$$\frac{4 \times 16}{C_2^{52}} = \frac{32}{663}$$

1.(b)

$$\frac{4 \times 4 + 3}{C_1^{50}} = \frac{19}{50}$$

1.(c)

The probability we bust:

- When dealer face-up = A,2,…,8:

  P1 = $\frac{4+4+4+4+3}{49} = \frac{19}{49}$

- When dealer face-up = 9,10,…,K:

  P2 = $\frac{4+4+4+4+3-1}{49} = \frac{18}{49}$

The probability dealer bust:

- When dealer face-up A:

  P(face-up + new card $\geq$ 21)

  = $\frac{0}{C_2^{49}} = 0$

- When dealer face-up 2:

  P(face-up + new card $\geq$ 20) = P(face-up +new card =21) + P(face-up +new card =20)

  = $\mathrm{P}(\text{face} - \text{up} + \text{new card} \geq 21) + \frac{C_2^{15}}{C_2^{49}} = \frac{105}{1176}$

- When dealer face-up 3:

  P(face-up + new card $\geq$ 19) = P(face-up +new card =21) + P(face-up +new card =20) + P(face-up +new card =19)

  = $\mathrm{P}(\text{face} - \text{up} + \text{new card} \geq 20) + \frac{C_1^4 C_1^{15}}{C_2^{49}} = \frac{165}{1176}$

- When dealer face-up 4:

  P(face-up + new card $\geq$ 18) = P(face-up +new card =21) + …+ P(face-up +new card =18)

  = $\mathrm{P}(\text{face} - \text{up} + \text{new card} \geq 19) + \frac{C_1^4 C_1^{15}}{C_2^{49}} + \frac{C_2^4}{C_2^{49}} = \frac{231}{1176}$

- When dealer face-up 5:

  P(face-up + new card $\geq$ 17) = P(face-up +new card =21) + …+ P(face-up +new card =17)

$= \mathrm{P}(\text{face} - \text{up} + \text{new card} \geq 18) + \frac{C_1^4 C_1^{15}}{C_2^{49}} + \frac{C_1^4 C_1^4}{C_2^{49}} = \frac{307}{1176}$

- When dealer face-up 6:

  P(face-up + new card $\geq$ 16) = P(face-up +new card =21) + …+ P(face-up +new card =16)

  $= \mathrm{P}(\text{face} - \text{up} + \text{new card} \geq 17) + \frac{C_1^3 C_1^{15}}{C_2^{49}} + \frac{C_1^4 C_1^4}{C_2^{49}} + \frac{C_2^4}{C_2^{49}} = \frac{374}{1176}$

- When dealer face-up 7:

  P(face-up + new card $\geq$ 15) = P(face-up +new card =21) + …+ P(face-up +new card =15)

  $= \mathrm{P}(\text{face} - \text{up} + \text{new card} \geq 16) + \frac{C_1^4 C_1^{15}}{C_2^{49}} + \frac{C_1^4 C_1^4}{C_2^{49}} + \frac{C_1^3 C_1^4}{C_2^{49}} = \frac{462}{1176}$

  is bigger than P1(our bust probability) = $\frac{19}{49} = \frac{456}{1176}$, and apparently, the bust probability of dealer will increase when face-up become bigger, while our bust probability will decrease when face-up > 8 (P2 is smaller then P1)

Ans: When dealer face-up = 7,8,9,10,J,Q,K, we will take a card

2.(a)

Expected value: $(-1)\left(\frac{26}{38}\right) + (2)\left(\frac{12}{38}\right) = -0.053$

Variance: $\left(2 - \left(\frac{-2}{38}\right)\right)^2 \left(\frac{12}{38}\right) + \left(-1 - \left(\frac{-2}{38}\right)\right)^2 \left(\frac{26}{38}\right) = 1.9446$

2.(b)

	Expected value	Variance
Red or Black	-0.053	0.997
Odd or Even	-0.053	0.997
1 to 18 or 19 to 36	-0.053	0.997
Dozen (1 to 12,…)	-0.053	1.9446
Column (on the right)	-0.053	1.9446
Single Number	-0.053	33.21

(1) Maximize expected value:

Since all the expected values are same. My strategy is bet 100 dollars on Single Number.

(2) Minimize variance:

I use greedy approach.

First, I bet 96 dollars on 1 to 18, the lowest variance. Then, I bet 4 dollars on Dozen, the second low variance.

2(c)

2(d)

There two limits in this strategy.

First is you don’t have that much money to keep losing and double. Assuming you bet 1 dollar at first, after 30 rounds, you need 1073741824 dollars for next round, which is an extremely big number.

Second, most of the casino set a bet limit. You cannot bet more dollars than this limit.

The questions 3,4 codes have same package and global variables:

```
import numpy as np
import pandas as pd
from math import sqrt
import matplotlib.pyplot as plt

data = np.array([[82,87,94,101,107,114,115,112,112,103,94,83],
                 [77,81,87,94,104,109,110,108,107,97,89,81],
                 [76,73,72,82,88,92,95,97,93,88,78,75],
                 [78,83,91,100,107,110,113,112,111,102,90,77],
                 [93,92,98,105,104,109,109,105,111,111,101,92],
                 [95,95,98,106,102,112,108,105,113,108,100,92],
                 [80,83,88,96,106,117,118,115,116,106,89,78],
                 [76,78,88,95,105,115,114,112,109,104,87,74],
                 [88,90,93,98,97,101,99,98,111,107,100,88],
                 [79,81,87,94,97,103,99,98,106,102,86,76],
                 [86,87,94,96,101,109,109,105,104,103,97,89],
                 [89,89,95,103,105,110,104,104,106,108,96,90]], dtype=np.float64)
n, m = data.shape
```

3(a)

Code:

```
# LU factorization
LU_data = data.copy()
L = np.eye(n, dtype=np.float64)
for column in range(n):
    for row in range(column + 1, n):
        L[row][column] = LU_data[row][column]/LU_data[column][column]
        LU_data[row] =  LU_data[row] - LU_data[column]*L[row][column]
print('L matrix:')
a = pd.DataFrame(L)
print(a.round(2))
print('U matrix:')
a = pd.DataFrame(LU_data)
print(a.round(2))
```

Result:

L matrix:

	0	1	2	3	4	5	6	7	8	9	10	11
0	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.0
1	0.94	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.0
2	0.93	10.98	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.0
3	0.95	−0.35	−0.96	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.0
4	1.13	9.60	−2.99	−6.25	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.0
5	1.16	8.33	0.28	−2.44	0.30	1.00	0.00	0.00	0.00	0.00	0.00	0.0
6	0.98	2.70	0.24	0.22	−0.03	−10.32	1.00	0.00	0.00	0.00	0.00	0.0
7	0.93	3.79	−4.76	−4.90	0.98	−21.97	2.14	1.00	0.00	0.00	0.00	0.0
8	1.07	4.84	1.46	−2.10	0.11	15.62	−1.48	0.29	1.00	0.00	0.00	0.0
9	0.96	4.05	−1.32	−2.22	0.39	−2.12	0.30	0.46	0.33	1.00	0.00	0.0
10	1.05	6.11	−2.65	−8.09	1.09	−1.50	0.07	0.86	0.49	−0.64	1.00	0.0
11	1.09	7.81	−2.41	−4.22	0.73	−1.98	0.33	0.71	−0.04	2.88	−1.42	1.0

U matrix:

	0	1	2	3	4	5	6	7	8	9	10	11
0	82.0	87.0	94.00	101.00	107.00	114.00	115.00	112.00	112.00	103.00	94.00	83.00
1	0.0	−0.7	−1.27	−0.84	3.52	1.95	2.01	2.83	1.83	0.28	0.73	3.06
2	0.0	0.0	−1.19	−2.37	−49.88	−35.09	−33.68	−37.88	−30.89	−10.54	−17.16	−35.54
3	0.0	0.0	0.00	1.37	−41.22	−31.29	−27.88	−29.75	−24.43	−5.96	−15.56	−34.85
4	0.0	0.0	0.00	0.00	−457.59	−339.27	−315.48	−348.11	−278.41	−77.19	−161.05	−355.35
5	−0.0	0.0	0.00	0.00	0.00	−0.59	−5.49	−5.38	0.98	−1.98	0.39	2.37
6	−0.0	0.0	0.00	0.00	0.00	0.00	−51.18	−51.75	16.70	−14.03	2.27	19.22
7	−0.0	0.0	−0.00	0.00	0.00	−0.00	−0.00	4.27	−10.87	−9.85	0.22	3.42
8	0.0	0.0	−0.00	−0.00	0.00	0.00	0.00	0.00	17.90	19.09	2.27	−8.91
9	0.0	0.0	−0.00	0.00	0.00	0.00	0.00	0.00	0.00	2.80	−2.86	−2.06
10	−0.0	0.0	0.00	−0.00	0.00	0.00	0.00	0.00	0.00	−0.00	−4.67	−3.41
11	−0.0	0.0	0.00	−0.00	0.00	0.00	0.00	0.00	0.00	−0.00	0.00	−1.56

3(b)

Code:

```
# Gram-Schmidt
GS_data = data
Q = np.zeros((n,n), dtype=np.float64)
R = np.zeros((n,n), dtype=np.float64)
for i in range(0, n):
    Q[:, i] = GS_data[:,i]
    for j in range(i):
        R[j][i] = Q[:, j].dot(GS_data[:, i])
        Q[:, i] -= R[j][i]*Q[:, j]
    R[i][i] = np.linalg.norm(Q[:, i])
    Q[:, i] /= R[i][i]

R_inv = np.zeros((n,n), dtype=np.float64)
for j in range(n):
    for i in range(j):
        for k in range(j):
            R_inv[i][j] = R_inv[i][j] + R_inv[i][k] * R[k][j]
    for k in range(j):
        R_inv[k][j] = -R_inv[k][j] / R[j][j]
    R_inv[j][j] = 1 / R[j][j]
Q_inv = Q.transpose()

print('Q matrix:')
a = pd.DataFrame(Q)
print(a.round(2))
print('R matrix:')
b = pd.DataFrame(R)
print(b.round(2))
print('Inverse matrix:')
a = pd.DataFrame(R_inv @ Q_inv)
print(a.round(2))
```

Result:

```
Q matrix:
      0     1     2     3     4     5     6     7     8     9     10    11
0   0.28  0.42 -0.13 -0.02 -0.08  0.07  0.17  0.00 -0.36  0.19  0.03  0.72
1   0.27  0.31 -0.13  0.04  0.31 -0.23 -0.19 -0.51 -0.07 -0.57  0.20 -0.10
2   0.26 -0.54 -0.35  0.43  0.42 -0.18  0.14  0.23 -0.08 -0.06 -0.09  0.17
3   0.27  0.43  0.03  0.34 -0.22 -0.36  0.34  0.28 -0.09  0.07 -0.18 -0.46
4   0.32 -0.34  0.28 -0.05 -0.28 -0.21  0.46 -0.51  0.28  0.11  0.09  0.12
5   0.33 -0.22 -0.25 -0.05 -0.46  0.56  0.07  0.03 -0.28 -0.36  0.01 -0.21
6   0.28  0.18 -0.18  0.13  0.35  0.50  0.02 -0.26  0.26  0.51 -0.01 -0.26
7   0.26  0.07  0.63  0.22  0.17  0.28 -0.01  0.37  0.20 -0.23  0.36  0.11
8   0.30  0.04 -0.41 -0.42 -0.12 -0.23 -0.13  0.35  0.45  0.06  0.39  0.00
9   0.27  0.06  0.06  0.06 -0.16  0.01 -0.39  0.01  0.41 -0.15 -0.70  0.22
10  0.30 -0.08  0.26 -0.66  0.40 -0.04  0.14  0.13 -0.28  0.01 -0.31 -0.16
11  0.31 -0.21  0.20  0.12 -0.17 -0.20 -0.63 -0.07 -0.38  0.39  0.18 -0.11
R matrix:
        0       1       2       3       4       5       6       7       8       9       10      11
0   289.25  294.78  313.66  337.98  352.18  374.69  371.99  365.47  374.39  357.72  320.19  288.06
1     0.00    8.21   14.50   14.57   22.91   24.07   26.37   24.55   22.81   13.09    9.86    1.65
2     0.00    0.00    7.35    5.77    9.23   11.37    9.66    7.83    2.34    7.75    2.57    0.01
3     0.00    0.00    0.00    7.28   13.80   13.50   15.81   18.81   11.47    6.19   -2.90   -3.18
4     0.00    0.00    0.00    0.00   12.30   14.83   19.62   20.08    9.16    4.89    2.15    3.08
5     0.00    0.00    0.00    0.00    0.00    8.18    6.01    3.75    4.67    1.76   -2.41   -3.66
6     0.00    0.00    0.00    0.00    0.00    0.00    6.74    4.44    0.89   -2.23    0.95   -0.79
7     0.00    0.00    0.00    0.00    0.00    0.00    0.00    2.29    1.74    1.98    0.80   -2.03
8     0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    8.15    7.27    1.30   -3.29
9     0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    2.68   -0.24   -0.75
10    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    2.63    1.64
11    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.18
Inverse matrix:
      0     1     2     3     4     5     6     7     8     9     10    11
0  -3.22  0.63 -0.61  1.87 -0.43  1.01  0.86 -0.37 -0.13 -0.69  0.74  0.24
1  -1.99  0.02 -0.59  1.50 -0.47  0.52  1.25 -0.75  0.03 -0.87  0.54  0.68
2   2.26 -0.30  0.62 -1.43  0.25 -0.62 -1.21  0.47 -0.10  1.16 -0.26 -0.69
3   2.14 -0.36  0.43 -1.28  0.43 -0.55 -0.69  0.41  0.06  0.35 -0.69 -0.20
4  -4.52  0.85 -1.04  2.78 -0.62  1.21  1.61 -0.84 -0.09 -1.23  1.00  0.71
5  -0.46  0.10 -0.16  0.20 -0.16  0.23  0.20  0.02  0.03 -0.25  0.05  0.17
6  -0.80  0.12 -0.26  0.52  0.12  0.17  0.51 -0.27 -0.08 -0.36  0.15  0.13
7   2.78 -0.57  0.77 -1.63  0.18 -0.76 -1.18  0.57  0.08  0.90 -0.51 -0.48
8   1.12 -0.01  0.31 -0.81  0.19 -0.28 -0.59  0.27  0.00  0.55 -0.29 -0.39
9   0.98 -0.33  0.19 -0.56  0.19 -0.41 -0.14  0.07  0.04  0.20 -0.21  0.01
10 -2.52  0.42 -0.63  1.56 -0.37  0.76  0.93 -0.27  0.15 -1.06  0.45  0.47
11  4.06 -0.56  0.95 -2.61  0.65 -1.21 -1.49  0.65  0.01  1.27 -0.91 -0.64
```

3(c)

Code:

```
# Power Iteration
def eigenvalue(A, v):
    Av = A @ v.transpose()
    return Av.transpose() @ v

np.random.seed(0)
v = np.random.rand(n)
v /=np.linalg.norm(v)
ev = eigenvalue(data, v)

while True:
    x_new = data.dot(v)
    v_new = x_new / np.linalg.norm(x_new)
    ev_new = eigenvalue(data, v_new)
    if np.abs(ev - ev_new) < 0.0000001:
        break
    v = v_new
    ev = ev_new
print('Eigen value: {}'.format(ev_new))
print('Eigen vector: {}'.format(v_new))
```

Result:

```
Eigen value: 1167.7759013685466
Eigen vector: [0.29756758 0.28287902 0.24957879 0.29013861 0.30377393 0.30481108
 0.29475535 0.28603568 0.28885816 0.27368297 0.29154169 0.29620859]
```

3(d)

Code:

```
# QR factorization to find eigenvalue
def QR(A):
    n, _ = A.shape
    Q = np.zeros((n,n), dtype=np.float64)
    R = np.zeros((n,n), dtype=np.float64)
    for i in range(n):
        Q[:, i] = A[:,i]
        for j in range(i):
            R[j][i] = Q[:, j].dot(A[:, i])
            Q[:, i] -= R[j][i]*Q[:, j]
        R[i][i] = np.linalg.norm(Q[:, i])
        Q[:, i] /= R[i][i]
    return Q, R

Ak = data.copy()
Q_product = np.eye(n, dtype=np.float64)
real_ev = set()
for i in range(100):
    d_old = Ak.diagonal()
    Q, R = QR(Ak)
    Q_product = Q_product @ Q
    Ak = R @ Q
    d_new = Ak.diagonal()
    for j in range(n):
        if np.abs(d_new - d_old)[j] < 0.001:
            real_ev.add(j)

real_ev = sorted(real_ev)
print('Eigen Vectors matrix:')
a = pd.DataFrame([Q_product[:, i] for i in real_ev])
print(a.T.round(2))
print('Eigen Values matrix:')
a = pd.DataFrame([Ak[i][i] for i in real_ev])
print(a.round(2))
```

Result:

```
Eigen Vectors matrix:
       0     1     2     3     4     5
0   0.30  0.25 -0.14 -0.06 -0.09 -0.79
1   0.28  0.14 -0.07  0.45  0.35  0.19
2   0.25 -0.88  0.09 -0.19  0.16 -0.10
3   0.29  0.15  0.41 -0.15 -0.20  0.43
4   0.30 -0.05  0.33  0.62  0.19 -0.12
5   0.30 -0.07 -0.71 -0.01  0.08  0.23
6   0.29 -0.12 -0.25  0.31 -0.47  0.11
7   0.29  0.16 -0.08 -0.24  0.20 -0.08
8   0.29 -0.06  0.13 -0.19 -0.19 -0.07
9   0.27  0.14  0.22 -0.25  0.48 -0.02
10  0.29  0.20 -0.10 -0.33  0.05  0.24
11  0.30  0.04  0.21 -0.02 -0.49 -0.03
Eigen Values matrix:
         0
0  1167.78
1    -8.00
2     5.34
3     1.12
4    -0.75
5    -0.37
```

4(a)

Code:

```
# a
def standardize(data):
    data_n = data
    mean = []
    deviation = []
    for i in range(n):
        sum = 0
        mean.append(np.mean(data[:, i]))
        for j in range(n):
            sum += (data[j][i] - mean[i]) ** 2
        deviation.append(sqrt(sum/(n-1)))
    for i in range(n):
        for j in range(n):
            data_n[j][i] = (data_n[j][i] - mean[i]) / deviation[i]
    return data_n

data_n = standardize(data)

print('Standardized matrix:')
a = pd.DataFrame(data_n)
print(a.round(2))

# covariance matrix
covariance = np.eye(n, dtype=np.float64)
for i in range(1, n):
    for j in range(i):
        sum = 0
        for k in range(n):
            sum += (data_n[k][i]) * (data_n[k][j])
        covariance[i][j] = sum / (n-1)
        covariance[j][i] = covariance[i][j]
print('Covariance matrix:')
a = pd.DataFrame(covariance)
print(a.round(2))
```

Result:

```
Standardized matrix:
      0     1     2     3     4     5     6     7     8     9     10    11
0  -0.19  0.33  0.51  0.55  0.92  0.81  1.01  1.00  0.64 -0.04  0.25  0.01
1  -0.93 -0.63 -0.49 -0.55  0.38  0.08  0.31  0.34 -0.21 -1.03 -0.47 -0.28
2  -1.08 -1.90 -2.62 -2.42 -2.51 -2.39 -1.78 -1.46 -2.59 -2.52 -2.05 -1.14
3  -0.78 -0.31  0.08  0.39  0.92  0.23  0.73  1.00  0.47 -0.21 -0.32 -0.85
4   1.45  1.13  1.08  1.17  0.38  0.08  0.17 -0.15  0.47  1.28  1.26  1.31
5   1.75  1.61  1.08  1.32  0.02  0.52  0.03 -0.15  0.81  0.79  1.11  1.31
6  -0.48 -0.31 -0.34 -0.23  0.74  1.25  1.43  1.49  1.31  0.45 -0.47 -0.71
7  -1.08 -1.11 -0.34 -0.39  0.56  0.96  0.87  1.00  0.13  0.12 -0.75 -1.28
8   0.71  0.81  0.37  0.08 -0.89 -1.08 -1.22 -1.30  0.47  0.62  1.11  0.73
9  -0.63 -0.63 -0.49 -0.55 -0.89 -0.79 -1.22 -1.30 -0.38 -0.21 -0.90 -1.00
10  0.41  0.33  0.51 -0.23 -0.17  0.08  0.17 -0.15 -0.72 -0.04  0.68  0.88
11  0.85  0.65  0.65  0.86  0.56  0.23 -0.52 -0.31 -0.38  0.79  0.54  1.02
Covariance matrix:
      0     1     2     3     4     5     6     7     8     9     10    11
0   1.00  0.93  0.77  0.75  0.13  0.14 -0.08 -0.22  0.34  0.72  0.89  0.95
1   0.93  1.00  0.92  0.90  0.39  0.36  0.15  0.01  0.59  0.82  0.97  0.91
2   0.77  0.92  1.00  0.96  0.66  0.61  0.40  0.27  0.72  0.90  0.93  0.77
3   0.75  0.90  0.96  1.00  0.70  0.63  0.42  0.32  0.75  0.89  0.86  0.71
4   0.13  0.39  0.66  0.70  1.00  0.93  0.88  0.85  0.77  0.61  0.42  0.17
5   0.14  0.36  0.61  0.63  0.93  1.00  0.92  0.87  0.79  0.61  0.37  0.14
6  -0.08  0.15  0.40  0.42  0.88  0.92  1.00  0.98  0.69  0.37  0.18 -0.05
7  -0.22  0.01  0.27  0.32  0.85  0.87  0.98  1.00  0.62  0.25  0.04 -0.19
8   0.34  0.59  0.72  0.75  0.77  0.79  0.69  0.62  1.00  0.78  0.58  0.26
9   0.72  0.82  0.90  0.89  0.61  0.61  0.37  0.25  0.78  1.00  0.83  0.63
10  0.89  0.97  0.93  0.86  0.42  0.37  0.18  0.04  0.58  0.83  1.00  0.91
11  0.95  0.91  0.77  0.71  0.17  0.14 -0.05 -0.19  0.26  0.63  0.91  1.00
```

4(b)

Code:

```
# b
def eigenvalue(A, v):
    Av = A @ v
    return Av.dot(v)

def powerIteration(data, n):
    np.random.seed(0)
    v = np.random.rand(n)
    v /=np.linalg.norm(v)
    ev = eigenvalue(data, v)
    while True:
        x_new = data.dot(v)
        v_new = x_new / np.linalg.norm(x_new)
        ev_new = eigenvalue(data, v_new)
        if np.abs(ev - ev_new) < 0.00001:
            break
        v = v_new
        ev = ev_new
    return ev_new, v_new

ev_vec = []
v_vec = []
_covariance = covariance.copy()
while len(ev_vec) < n:
    ev, v = powerIteration(_covariance, n)
    ev_vec.append(ev)
    v_vec.append(v)
    for i in range(n):
        _covariance[i] -= _covariance[i].dot(v) * v
print('Top 3 principle components: ')
a = pd.DataFrame(v_vec[:3])
print(a.T.round(2))
print('Cumulative percentage: ')
print('{}%'.format(np.sum(ev_vec[:3])*100 / np.sum(ev_vec)))
```

Result:

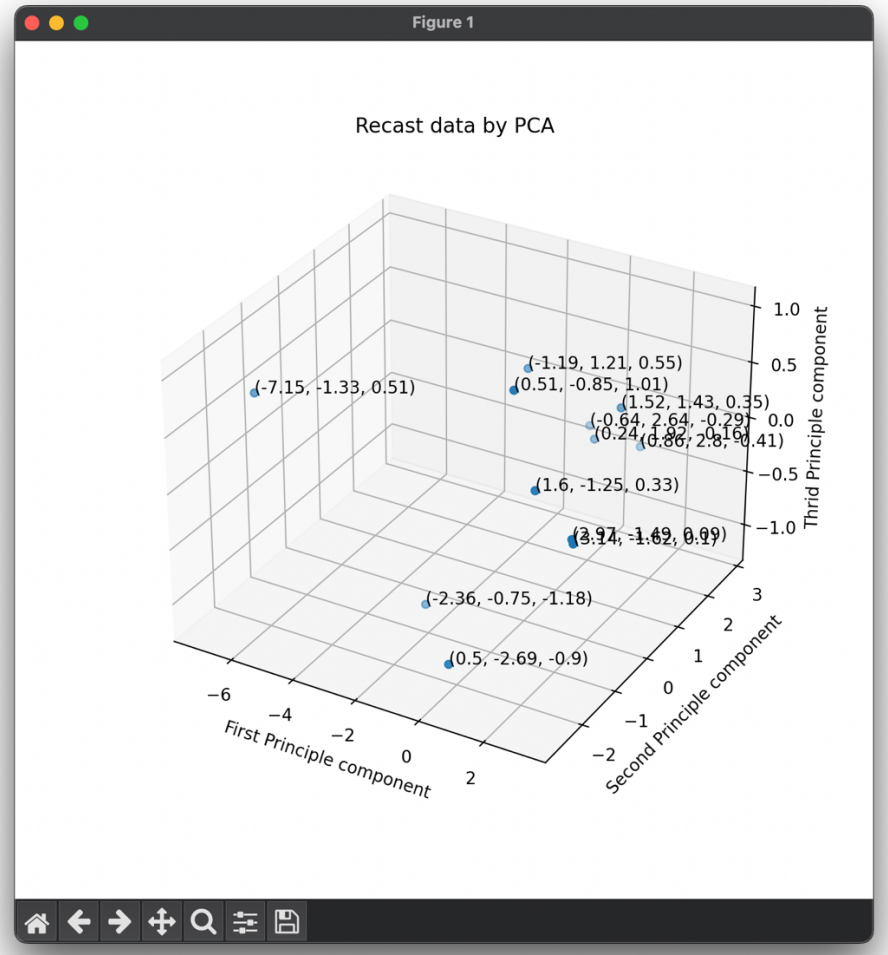
```
Top 3 principle components:
        0     1     2
0    0.26 -0.35  0.14
1    0.32 -0.24  0.03
2    0.35 -0.09 -0.00
3    0.35 -0.06 -0.08
4    0.28  0.32  0.12
5    0.27  0.34  0.09
6    0.20  0.42  0.29
7    0.16  0.46  0.27
8    0.30  0.18 -0.59
9    0.33 -0.06 -0.46
10   0.32 -0.22  0.09
11   0.26 -0.34  0.48
Cumulative percentage:
95.68586815858009%
```

4(c)

Code:

```python
# c
rotation_mat = np.zeros((3, 12), dtype=np.float64)
for i in range(3):
    rotation_mat[i] = v_vec[i]
recast_data = rotation_mat @ data.transpose()
fig = plt.figure(figsize=(7,7))
ax = fig.add_subplot(111, projection='3d')
ax.set_title("Recast data by PCA")
ax.set_xlabel("First Principle component")
ax.set_ylabel("Second Principle component")
ax.set_zlabel("Thrid Principle component")
ax.scatter(recast_data[0], recast_data[1], recast_data[2])
for i_x, i_y, i_z in zip(recast_data[0], recast_data[1], recast_data[2]):
    ax.text(i_x, i_y, i_z, '({}, {}, {})'.format(round(i_x, 2), round(i_y, 2), round(i_z, 2)))
plt.show()
```

Result:



Since the plot has some overlap, I provide the exact recast data. Each row is a data, and column 0 is first principal component, column 1 is second principal component…

Recast data

	0	1	2
0	1.52	1.43	0.35
1	-1.19	1.21	0.55
2	-7.15	-1.33	0.51
3	0.24	1.92	-0.16
4	2.97	-1.49	0.09
5	3.14	-1.62	0.10
6	0.86	2.80	-0.41
7	-0.64	2.64	-0.29
8	0.50	-2.69	-0.90
9	-2.36	-0.75	-1.18
10	0.51	-0.85	1.01
11	1.60	-1.25	0.33

4(d)

Code:

```
# d
def standardize(data, n):
    data_n = data
    mean = []
    deviation = []
    for i in range(n):
        sum = 0
        mean.append(np.mean(data[:, i]))
        for j in range(n):
            sum += (data[j][i] - mean[i]) ** 2
        deviation.append(sqrt(sum/(n-1)))
    for i in range(n):
        for j in range(n):
            data_n[j][i] = (data_n[j][i] - mean[i]) / deviation[i]
    return data_n

def QRalgo(data, n):
    def QR(A):
        n, _ = A.shape
        Q = np.zeros((n,n), dtype=np.float64)
        R = np.zeros((n,n), dtype=np.float64)
        for i in range(n):
            Q[:, i] = A[:,i]
            for j in range(i):
                R[j][i] = Q[:, j].dot(A[:, i])
                Q[:, i] -= R[j][i]*Q[:, j]
            R[i][i] = np.linalg.norm(Q[:, i])
            Q[:, i] /= R[i][i]
        return Q, R

    Ak = data
    Q_product = np.eye(n, dtype=np.float64)
    for i in range(100):
        Q, R = QR(Ak)
        Q_product = Q_product @ Q
        Ak = R @ Q
    return Ak.diagonal(), Q_product
```

```
data_n = standardize(data, n)
# do SVD on data/sqrt(n-1)
ATA = (data_n.transpose() @ data_n) / (n - 1)

s_square, v = QRalgo(ATA, n)
s = s_square.copy()
for i in range(n):
    if s_square[i] > 0:
        s[i] = sqrt(s_square[i])
u = np.zeros((n, n), dtype=np.float64)
for i in range(n):
    if s[i] > 0:
        u[:, i] = ((data_n/sqrt(n-1)) @ v[:, i]) / s[i]

print('Principle component matrix:')
a = pd.DataFrame(v)
print(a.round(2))
print('Eigen Values matrix:')
a = pd.DataFrame(s_square)
print(a.round(2))
```

Result:

```
Principle component matrix:
      0     1     2     3     4     5     6     7     8     9     10    11
0   0.26 -0.35  0.14 -0.24  0.37 -0.42  0.13 -0.11  0.06 -0.16 -0.56 -0.44
1   0.32 -0.24  0.03 -0.19 -0.21 -0.20 -0.22 -0.00 -0.11  0.69  0.27 -0.33
2   0.35 -0.08 -0.00  0.31 -0.16  0.25 -0.32 -0.46  0.13  0.18 -0.40 -0.15
3   0.35 -0.05 -0.08  0.36 -0.20 -0.56  0.12 -0.19 -0.05 -0.37  0.42 -0.12
4   0.28  0.32  0.13  0.49 -0.22  0.02  0.19  0.46  0.18  0.08 -0.31  0.33
5   0.27  0.34  0.09  0.05  0.45 -0.01 -0.67  0.16 -0.25 -0.19  0.09  0.35
6   0.20  0.42  0.28 -0.31  0.05  0.11  0.10 -0.39  0.58 -0.05  0.24  0.46
7   0.16  0.46  0.27 -0.20 -0.01 -0.13  0.38 -0.07 -0.54  0.23 -0.13  0.52
8   0.30  0.19 -0.60 -0.49 -0.31 -0.06 -0.09  0.28  0.13 -0.16 -0.16  0.17
9   0.33 -0.06 -0.45  0.17  0.58  0.26  0.39  0.02  0.04  0.22  0.20 -0.12
10  0.32 -0.22  0.09 -0.13 -0.25  0.54  0.13 -0.13 -0.41 -0.38  0.05 -0.31
11  0.26 -0.34  0.48 -0.13  0.03  0.13  0.03  0.50  0.22 -0.05  0.19 -0.43
Eigen Values matrix:
       0
0   7.57
1   3.52
2   0.39
3   0.19
4   0.14
5   0.10
6   0.05
7   0.03
8   0.01
9   0.00
10  0.00
11 -0.00
```

4(e)

U matrix can interpret as the principal components of transpose of dataset. For instance, a dataset is 12 dimensions with 100 data, then the columns of V matrix is principal components of these 100 data in 12 dimensions, and the columns of U matrix is principal components of 100 dimensions of 12 data.

4(f)

Code:

```
# rank 3 approximate of data
data_rank3 = data.copy()
ATA = data_rank3.transpose() @ data_rank3
s_square, v = QRalgo(ATA, n)
s = s_square.copy()
for i in range(n):
    if s_square[i] > 0:
        s[i] = sqrt(s_square[i])
u = np.zeros((n, n), dtype=np.float64)
for i in range(n):
    if s[i] > 0:
        u[:, i] = (data_rank3 @ v[:, i]) / s[i]

rank3_s = np.zeros((n, m), dtype=np.float64)
for i in range(3):
    rank3_s[i][i] = s[i]
rank3_data = u @ rank3_s @ v.transpose()
print('Rank 3 approximation of data:')
a = pd.DataFrame(rank3_data)
print(a.round(2))
```

Result:

```
Rank 3 approximation of data:
      0      1      2      3      4      5      6      7      8      9      10     11
0   83.53  85.84  92.09  99.77  106.86 114.06 114.40 112.50 112.36 105.67 93.33 83.37
1   79.08  80.91  86.44  93.91  102.19 109.14 110.59 108.88 105.91 99.08  88.08 79.73
2   73.12  72.96  76.07  82.38  88.59  94.21  96.11  94.75  90.35  85.93  79.57 75.13
3   78.94  82.44  89.78  97.49  105.43 112.84 112.88 110.93 111.70 103.92 89.54 77.84
4   92.65  93.06  97.59  104.24 104.01 109.99 107.53 105.49 112.13 109.99 101.03 92.33
5   93.28  93.56  97.97  104.59 104.13 110.07 107.57 105.53 112.26 110.30 101.57 93.03
6   78.56  82.39  90.09  98.25  108.39 116.24 117.25 115.32 113.73 104.68 89.55 77.74
7   75.47  79.62  87.52  95.50  105.55 113.28 114.08 112.17 111.13 101.99 86.50 74.32
8   88.98  89.82  94.65  100.63 97.72  103.15 98.98  96.89  107.83 106.75 97.34 87.47
9   77.91  81.02  87.84  94.36  96.45  102.70 99.86  97.82  105.73 101.01 87.76 75.56
10  87.70  87.44  91.08  97.93  101.53 107.62 107.83 106.10 105.95 102.53 95.19 89.09
11  89.43  89.98  94.53  101.18 102.08 108.09 106.21 104.25 109.44 106.73 97.72 89.29
```

5(a)

Since this dataset picks the highest temperature in a month, it's likely like a bunch of outliers. This kind of dataset is unlikely to be Gaussian.

5(b)

Header code of the 5b, 5c, 5d.

```
import numpy as np
import pandas as pd
from math import sqrt
import matplotlib.pyplot as plt

data = np.array([[87, 114, 103],
                 [81,109,97],
                 [73,92,88],
                 [83,110,102],
                 [92,109,111],
                 [95,112,108],
                 [83,117,106],
                 [78,115,104],
                 [90,101,107],
                 [81,103,102],
                 [87,109,103],
                 [89,110,108]], dtype=np.float64)

# preprocess data
data_p = data.transpose().copy()
```

Step 1:

Code:

```
# step 1
for i in range(3):
    mean = np.mean(data_p[i])
    for j in range(12):
        data_p[i][j] -= mean

print('Preprocessed step 1 data:')
a = pd.DataFrame(data_p)
print(a.round(2))
```

Result:

```
Preprocessed step 1 data:
      0     1      2     3     4      5     6     7     8     9     10    11
0  2.08 -3.92 -11.92 -1.92  7.08  10.08 -1.92 -6.92  5.08 -3.92  2.08  4.08
1  5.58  0.58 -16.42  1.58  0.58   3.58  8.58  6.58 -7.42 -5.42  0.58  1.58
2 -0.25 -6.25 -15.25 -1.25  7.75   4.75  2.75  0.75  3.75 -1.25 -0.25  4.75
```

Step 2:

Code:

```
# step 2
def QRalgo(data, n):
    def QR(A):
        n, _ = A.shape
        Q = np.zeros((n,n), dtype=np.float64)
        R = np.zeros((n,n), dtype=np.float64)
        for i in range(n):
            Q[:, i] = A[:,i]
            for j in range(i):
                R[j][i] = Q[:, j].dot(A[:, i])
                Q[:, i] -= R[j][i]*Q[:, j]
            R[i][i] = np.linalg.norm(Q[:, i])
            Q[:, i] /= R[i][i]
        return Q, R

    Ak = data
    Q_product = np.eye(n, dtype=np.float64)
    for i in range(100):
        Q, R = QR(Ak)
        Q_product = Q_product @ Q
        Ak = R @ Q
    return Ak.diagonal(), Q_product

covariance = data_p @ data_p.transpose()
ev, v = QRalgo(covariance, 3)

data_p = v.transpose() @ data_p
print('Preprocessed step 2 data:')
a = pd.DataFrame(data_p)
print(a.round(2))
```

Result:

```
Preprocessed step 2 data:
     0     1      2     3     4      5     6     7     8     9     10    11
0  4.20 -5.61 -25.22 -0.91  8.94  10.51  5.51  0.39  0.84 -6.03  1.34  6.04
1  3.19  4.02  -2.76  2.62 -5.27  -4.09  7.29  9.08 -9.55 -1.68 -0.72 -2.12
2 -2.79 -2.66  -0.73 -0.23  1.67  -3.14  1.14  3.03  1.73  2.67 -1.56  0.88
```

Step 3:

Code:

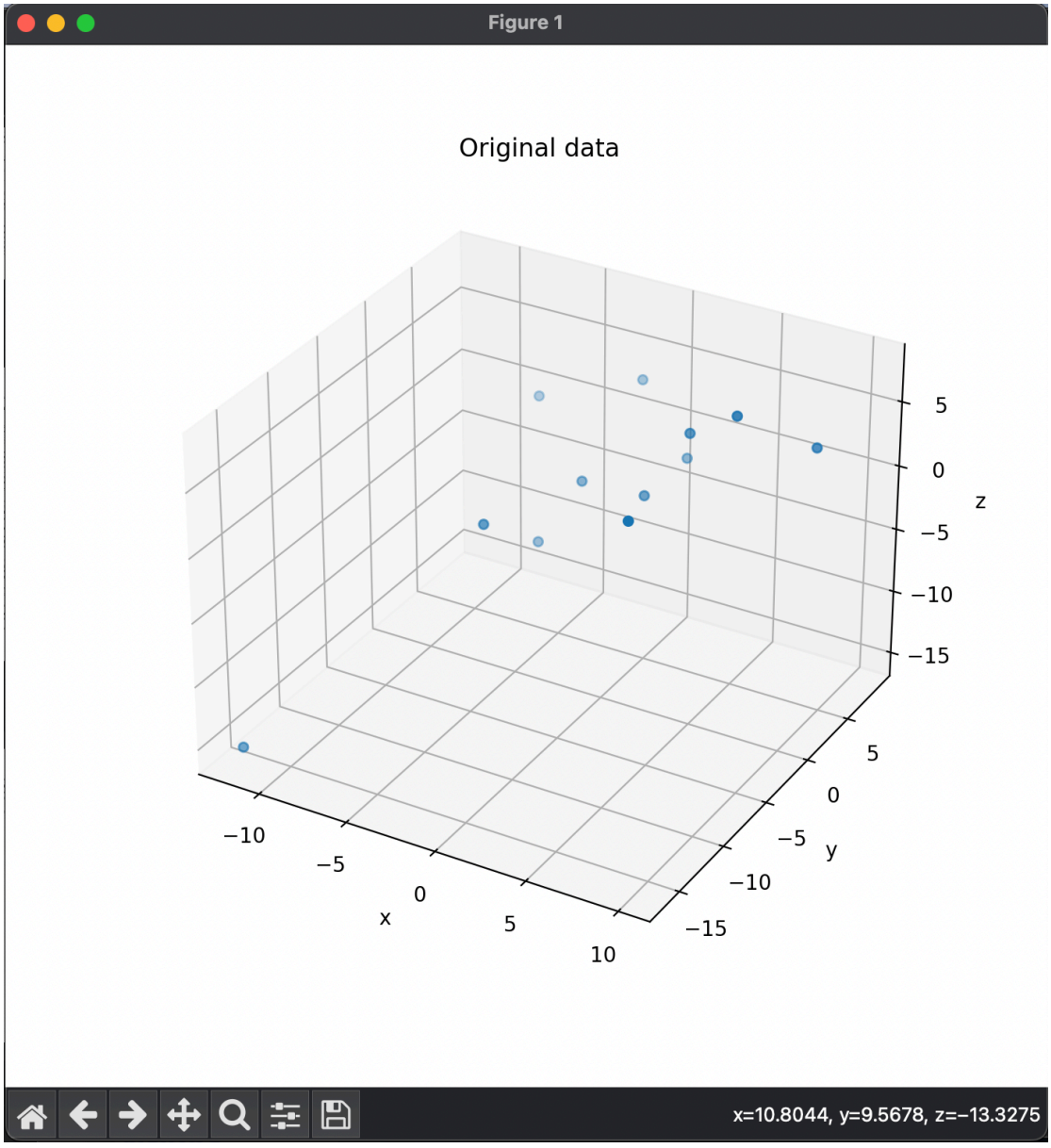
```
# step 3
data_p = np.diag(np.power(ev, -0.5)) @ data_p
print('Preprocessed step 3 data:')
a = pd.DataFrame(data_p)
print(a.round(2))
```

Result:

```
Preprocessed step 3 data:
      0     1     2     3     4     5     6     7     8     9     10    11
0  0.13 -0.18 -0.80 -0.03  0.29  0.34  0.18  0.01  0.03 -0.19  0.04  0.19
1  0.18  0.22 -0.15  0.15 -0.29 -0.23  0.41  0.51 -0.53 -0.09 -0.04 -0.12
2 -0.39 -0.37 -0.10 -0.03  0.23 -0.44  0.16  0.42  0.24  0.37 -0.22  0.12
```

5(c)

After step 1:



After U=VD:

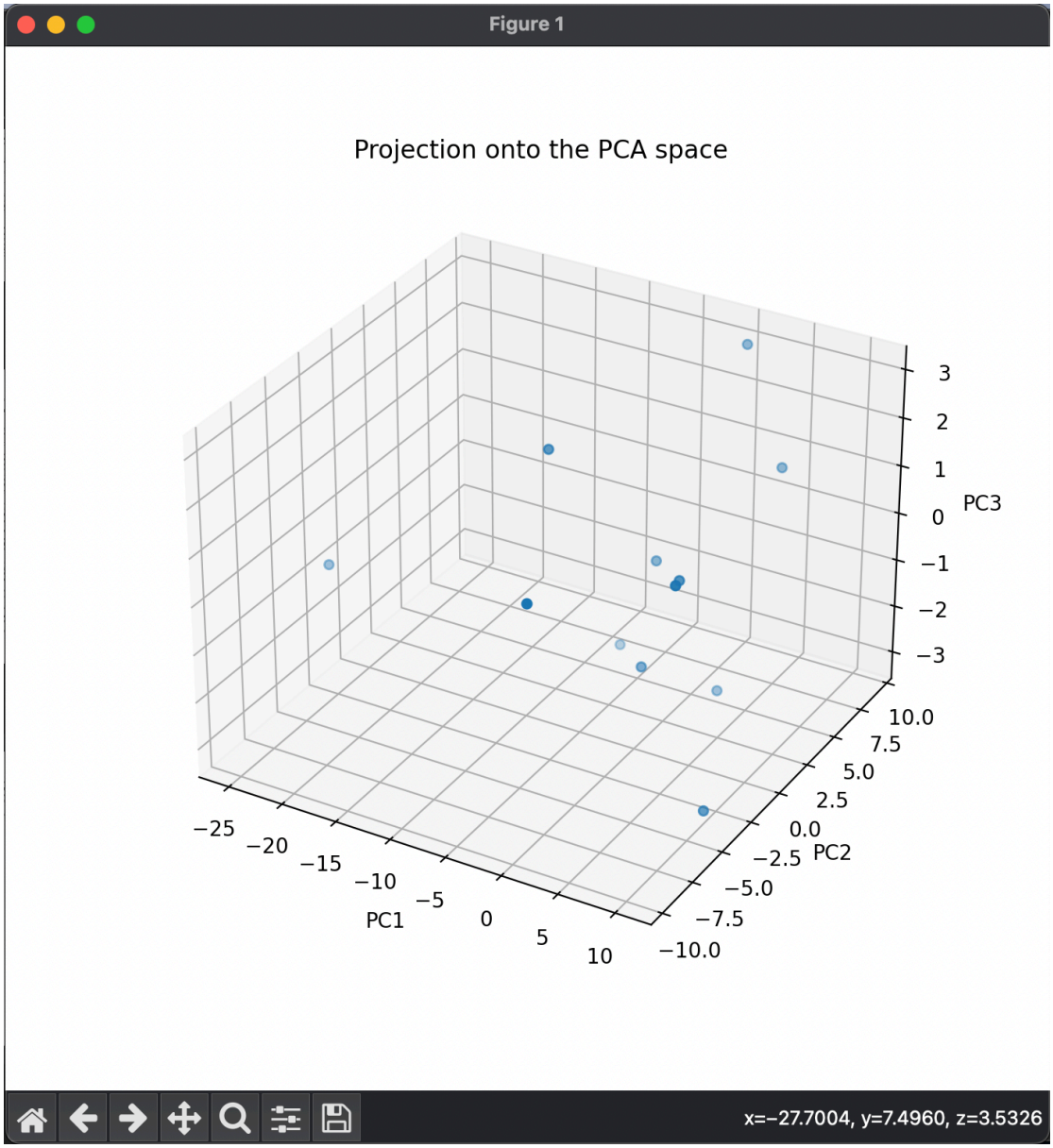
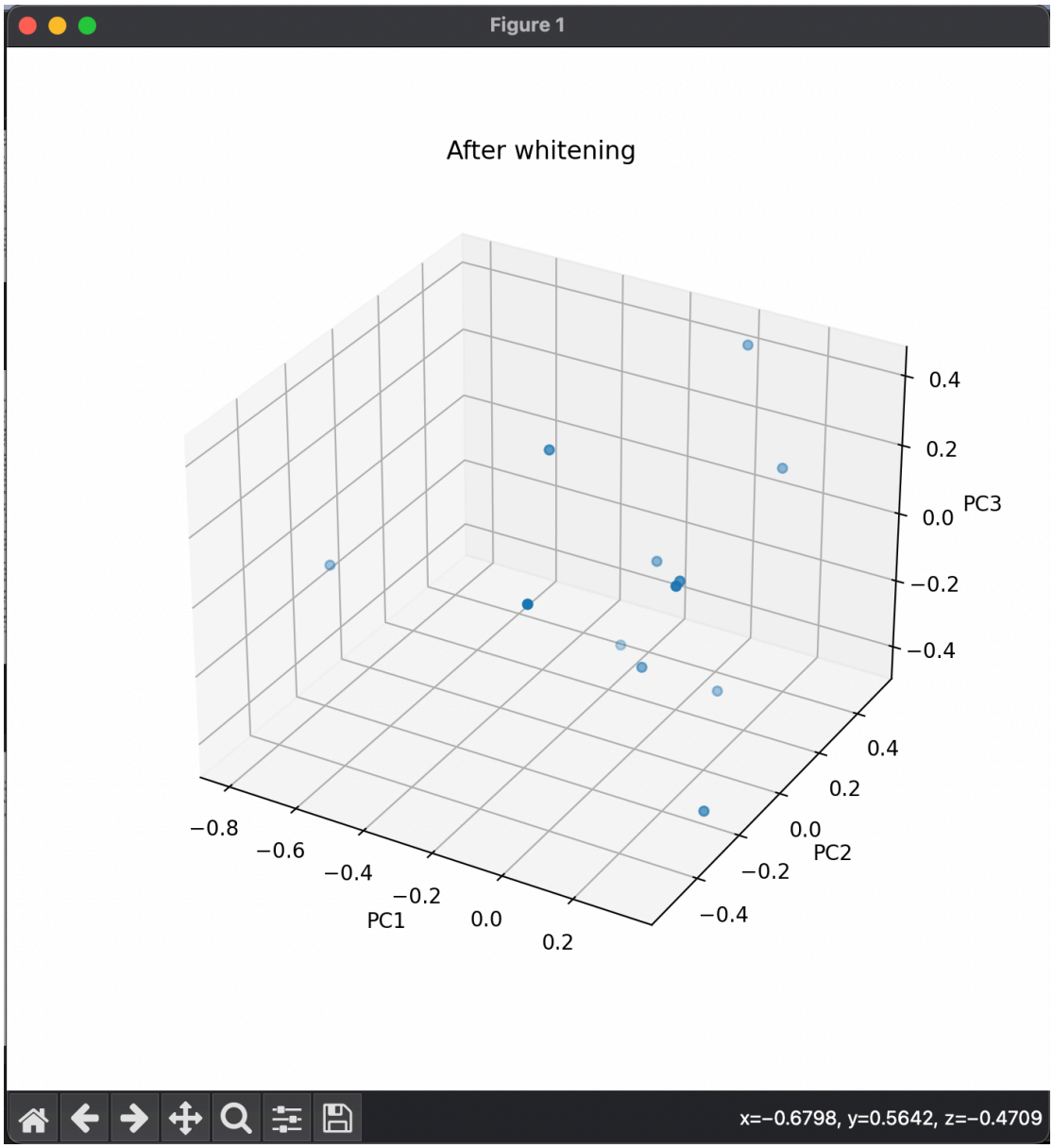


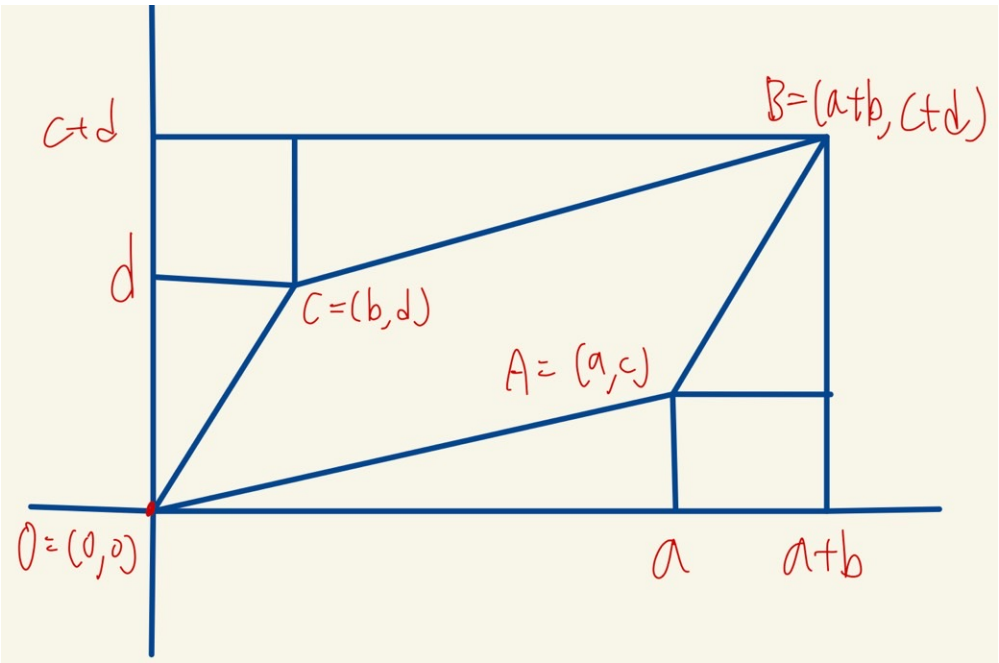
Figure 1
Projection onto the PCA space
PC1
PC2
PC3
x=−27.7004, y=7.4960, z=3.5326

After Z = $\lambda^{-0.5}U$



5(d)

6.



The area of parallelogram is the area of big rectangle minus the area of two small rectangle and the area of two triangles.

Area(OABC) = (a+b)(c+d) – (ac/2) – (bd/2) – ((a+b-b)(c+d-d)/2) – ((a+b-a)(c+d-c)/2) – (a+b-a)c – b(c+d-c)

= ac + ad+ bc +bd – ac/2 -ac/2 -bd/2 -bd/2

= ac-bd

= $\begin{vmatrix} a & b \\ c & d \end{vmatrix}$