COMPUTATIONAL METHODS FOR DATA SCIENCE
HOMEWORK 1

1. Basic Probability in Blackjack *(15 points)*

(a) Given a single deck, calculate the probability that you get a blackjack.

$$P\left(\text{一開局就 blackjack}\right) = \frac{64}{C_2^{52}} = \frac{64}{(52 \cdot 51)/2} = 0.048265$$

(b) What is the probability that you will bust if you take another card?

$$P\left(\text{下一張牌超過 8 點} \middle| \text{手上有 13 點}\right) = \frac{19}{C_1^{50}} = \frac{19}{50} = 0.38$$

(c) Explain when you will take a card if the face-up cards appear.

	莊家亮牌>6	莊家亮牌=6	莊家亮牌<6
莊家是否要牌	7,8,9,10, J, Q, K, A	6	2,3,4,5
要牌 (莊家總點數<17)	可能底牌 5,6,7,8,9 可能總點數範圍 11-16	可能底牌 2, 3,,.K 可能點數範圍 8-16	可能底牌 A,2, 3,,.K 可能點數範圍 3-16
不要牌 (莊家總點數≥17)	可能底牌 10,J,Q,K **可能總點數範圍** **17-20**	可能底牌 A 可能點數範圍 17	沒有可能底牌

要牌策略：當莊家亮出的牌大於 6，而且莊家選擇不繼續要牌，而延續 b 小題的前提，我手上的牌有 13 點，我會選擇繼續要牌。

策略說明：如果莊家亮出的牌大於 6，而且莊家選擇不繼續要牌，則莊家手上的總點數很有可能介於 17-20 點，此時我手上總點數只有 13 點，若不繼續要牌，在最終計算點數時，很有可能會輸給莊家，因為距離 21 點的距離，有很大的機會會比莊家的牌來得大。

2. Basic Statistics in Roulette *(10 points)*

(a) $E(X) = 2\left(\frac{12}{38}\right) + (-1)\left(\frac{26}{38}\right) = \frac{-1}{19} = -0.05263$

$$E(X^2) = 2^2\left(\frac{12}{38}\right) + (-1)^2\left(\frac{26}{38}\right) = \frac{37}{19} = 1.94376$$

$$Var(X) = E(X^2) - [E(X)]^2 = 1.94459$$

(b) Expectations and variances

- 每次賭本很低，但玩很多次的玩法

	Payout	Bet Amount	Expectation (A)	Variance (B)	Real Bet (D)	Times (C)	(A)*(CD)	(B)*(CD)^2
Red or Black	1-to-1	A multiple of 8	-0.053	0.997	8	12	-5.088	9188.35
Odd or Even	1-to-1	A multiple of 8	-0.053	0.997	8	12	-5.088	9188.35
1 to 18 or 19 to 36	1-to-1	A multiple of 8	-0.053	0.997	8	12	-5.088	9188.35
Dozen (1 to 12, …)	2-to-1	A multiple of 4	-0.053	1.945	4	25	-5.30	19450
Column (on the right)	2-to-1	A multiple of 4	-0.053	1.945	4	25	-5.30	19450

Single Number	35-to-1	A multiple of 1	-0.053	33.21	1	100	-5.30	332100

- 一次賭本比較高，玩最少次的玩法

	Payout	Bet Amount	Expectation (A)	Variance (B)	Real Bet (D)	Times (C)	(A)*(CD)	(B)*(CD)^2
Red or Black	1-to-1	A multiple of 8	-0.053	0.997	96	1	-5.088	9188.35
Odd or Even	1-to-1	A multiple of 8	-0.053	0.997	96	1	-5.088	9188.35
1 to 18 or 19 to 36	1-to-1	A multiple of 8	-0.053	0.997	96	1	-5.088	9188.35
Dozen (1 to 12, ...)	2-to-1	A multiple of 4	-0.053	1.945	100	1	-5.30	19450
Column (on the right)	2-to-1	A multiple of 4	-0.053	1.945	100	1	-5.30	19450
Single Number	35-to-1	A multiple of 1	-0.053	33.21	100	1	-5.30	332100

在思考賭場測略時，基本上有兩種思考方向，每種玩法的期望值都是一樣的-0.053，但每種玩法的變異數會有不同，那相對應的每種玩法每次要下的最低賭本會有不一樣。第一種可以思考的玩法可以是每次玩的時候，都下最低賭本。而另一種玩法則是希望可以玩越少次越好，然後每次下的賭本就出到最大。兩種邏輯可以列出上面的兩張表。後來仔細一想，兩種玩法算出來的期望值及變異數，也會一樣，但在玩法上有些細微的不同。

(1) 期望值最大化的策略：當我們整理出上面的表格後，可以看出其實賭場內不同玩法間的期望值會是一樣的，都是-0.053 元，這個時候就要考慮當我們手上的賭本只有 100 元時，各種玩法可以進行的次數及每次可以下的賭本。現在假設前提是手上的 100 元需要全部用盡的情境下。在追求期望值最大化的策略中，基本上可以從 Red or Black，Odd or Even，1 to 18 or 19 to 36 這三種玩法中任選一種玩 12 次，每次都下最小賭本 8 元，或是乾脆一次下 96 元，就玩一次，兩種玩法的期望值基本上是一樣的，都是-5.088 元，這時候還剩下 4 元，可以考慮再從 Dozen 或 Column 兩種玩法中，任選一種來玩，把剩下的 4 元賭本出完，最後的期望值算出來是-5.088-0.053*4=-5.3。發現其實跟單純一次 Single Number 然後，一次就壓 100 元的期望值是一樣的。

並且其實當我們採用其他的玩法，例如 Dozen 或 Column，不管每次壓最小賭本 4 元玩 25 次，或是一次壓 100 元論輸贏，期望值都是-5.30 元。

因次在追求期望值最大的目標上，不管選哪種玩法，其實期望值都一樣。

(2) 變異數最小化的策略：雖然各種玩法經過上面的試算後，發現基本上都一樣。但是各種玩法間的變異數卻都有差異。變異數最小的玩法依然有三種，分別是 Red or Black，Odd or Even，1 to 18 or 19 to 36，不管是採用每次都最小賭本然後玩很多次，或是採用最高賭本，然後只玩一次的玩法，三種玩法在花了 96 元後，變異數都是 9188.35，還有剩下 4 元，可以考慮玩一次賭本 4 元的玩法，即 Dozen 或 Column，玩一次的變異數為 1.945*4^2，這樣就花了 100 元，變異數合計為 9188.35+1.945*4^2=9219.472，可以獲得最低的變異數。

(c) 程式模擬結果

(1) 期望值最大策略：

第一把賭 Red or Black，賭 Red，押 96 元

第二把賭 Dozen，賭 1-12，押 4 元

程式	

```
import numpy as np
import random
def spins():
    slots = {'00': 'green', '0': 'green', '1': 'red', '2': 'black',
```

	'3': 'red', '4': 'black', '5': 'red', '6': 'black', '7': 'red','8': 'black', '9': 'red', '10': 'black', '11': 'red', '12': 'black', '13': 'red', '14': 'black', '15': 'red', '16': 'black', '17': 'red', '18': 'black', '19': 'red', '20': 'black', '21': 'red', '22': 'black', '23': 'red', '24': 'black', '25': 'red', '26': 'black', '27': 'red', '28': 'black', '29': 'red', '30': 'black', '31': 'red', '32': 'black', '33': 'red', '34': 'black', '35': 'red', '36': 'black'} result = random.choice(list(slots.keys())) return result result = {} for i in range(2): result[i] = spins() result
程式模擬結果	{0: '18', 1: '34'} 第一把結果是黑色，不是紅色，所以 96 元賠掉。 第二把結果是 34，沒有落在 1-12 之間，所以 4 元也賠掉。
Average Winning per Bet	-100 元/2 把 = -50 元/把

(2) 變異數最小策略：

前 12 把賭 Red or Black，賭 Black，每把押 8 元

第 13 把賭 Dozen，賭 13-24，押 4 元

程式	import numpy as np import random def spins(): slots = {'00': 'green', '0': 'green', '1': 'red', '2': 'black', '3': 'red', '4': 'black', '5': 'red', '6': 'black', '7': 'red', '8': 'black', '9': 'red', '10': 'black', '11': 'red', '12': 'black', '13': 'red', '14': 'black', '15': 'red', '16': 'black', '17': 'red', '18': 'black', '19': 'red', '20': 'black', '21': 'red', '22': 'black', '23': 'red', '24': 'black', '25': 'red', '26': 'black', '27': 'red', '28': 'black', '29': 'red', '30': 'black', '31': 'red', '32': 'black', '33': 'red', '34': 'black', '35': 'red', '36': 'black'} result = random.choice(list(slots.keys())) return result

	result = {} for i in range(13): result[i] = spins() result
程式模擬結果	{0: '22', 1: '12', 2: '15', 3: '15', 4: '3', 5: '3', 6: '36', 7: '23', 8: '10', 9: '30', 10: '13', 11: '17', 12: '4'} 前 12 把中，有 5 把為 Black，7 把為 Red，前 12 把結果為-16 元。 第 13 把結果為 4，沒有落在 13-24 中，再賠掉 4 元。
Average Winning per Bet	-20 元/13 把 = -1.538 元/把

(d) 真實生活中，Steve 若採用不斷加碼的策略，首先會面對的問題就是每個人在賭場內的成本通常都是有限的。如果 Steve 身上剛好就只有帶 100 元，這種不斷加碼的策略，當他連續輸掉三把：-8-16-32=-56，當下他的口袋其實只剩下 44 元，已經無法再加碼到 64 元了。當發生連續輸錢好幾次的情形時，基本上就無法採用題目中 Steve 原先預想的策略，而且現實中當人的運氣不好時，的確會有機會連續輸很多局。

3. Monthly Record-Breaking Temperature in California I: Matrix Calculation *(25 points)*

(a) Run LU factorization on X to obtain matrix L and U.

```
import pandas as pd

data = pd.read_csv('/content/drive/MyDrive/臺大資料科學博士班/資料科學計算/作業/CAmaxTemp.txt', sep = '   ').reset_index(drop=False)
data['Station'] = data['index'].map(lambda x:x.split('\t')[0])
data['Period'] = data['index'].map(lambda x:x.split('\t')[1])
data.drop('index', axis=1, inplace=True)
data.columns = ['JAN','FEB','MAR','APR','MAY','JUN','JUL','AUG','SEP','OCT','NOV','DEC','MAX','Station','Period']
data = data[['JAN','FEB','MAR','APR','MAY','JUN','JUL','AUG','SEP','OCT','NOV','DEC']]

def LU_decomposition(a):
  dim = a.shape #先知道送進來矩陣的維度
  lower = np.identity(dim[0])  #產生 Identity Matrix 當作下三角的初始
  upper = np.zeros(dim) #產生 Identity Matrix 當作上三角的初始

  upper[0, :] = a[0, :]
  lower[:, 0] = a[:, 0]/upper[0,0]

  for i in range(1, dim[0]):
    for j in range(1, dim[0]):
```

```
      Z = sum(lower[i, k]*upper[k, j] for k in range(i))
      if i <= j:
        upper[i,j] = a[i,j] - Z
      elif i > j:
        lower[i,j] = (a[i,j] - Z)/upper[j,j]

  return lower, upper

L,U = LU_decomposition(data.to_numpy())
print(L)
print(U)
```

Lower triangular matrix L 結果如下

```
print(L)
```

```
[[  1.          0.          0.          0.          0.
    0.          0.          0.          0.          0.
    0.          0.        ]
 [  0.93902439  1.          0.          0.          0.
    0.          0.          0.          0.          0.
    0.          0.        ]
 [  0.92682927 10.98245614  1.          0.          0.
    0.          0.          0.          0.          0.
    0.          0.        ]
 [  0.95121951 -0.35087719 -0.95588235  1.          0.
    0.          0.          0.          0.          0.
    0.          0.        ]
 [  1.13414634  9.59649123 -2.98529412 -6.24731183  1.
    0.          0.          0.          0.          0.
    0.          0.        ]
 [  1.15853659  8.33333333  0.27941176 -2.44086022  0.30160259
    1.          0.          0.          0.          0.
    0.          0.        ]
 [  0.97560976  2.70175439  0.23529412  0.21505376 -0.02772817
  -10.32408736  1.          0.          0.          0.
    0.          0.        ]
 [  0.92682927  3.78947368 -4.76470588 -4.90322581  0.97748849
  -21.96987088  2.14427875  1.          0.          0.
    0.          0.        ]
 [  1.07317073  4.84210526  1.45588235 -2.09677419  0.10644797
   15.6169297  -1.48113472  0.28655739  1.          0.
    0.          0.        ]
 [  0.96341463  4.05263158 -1.32352941 -2.21505376  0.38831187
   -2.1157341   0.3010596   0.4644259   0.32886376  1.
    0.          0.        ]
 [  1.04878049  6.10526316 -2.64705882 -8.08602151  1.08847166
   -1.50374621  0.06609856  0.86240314  0.49464812 -0.63546906
    1.          0.        ]
 [  1.08536585  7.80701754 -2.41176471 -4.21505376  0.72704202
   -1.98102981  0.32833436  0.71029159 -0.04031333  2.88365162
   -1.41784205  1.        ]]
```

Upper triangular matrix U 結果如下

```
print(U)
```

```
[[ 8.20000000e+01  8.70000000e+01  9.40000000e+01  1.01000000e+02
   1.07000000e+02  1.14000000e+02  1.15000000e+02  1.12000000e+02
   1.12000000e+02  1.03000000e+02  9.40000000e+01  8.30000000e+01]
 [ 0.00000000e+00 -6.95121951e-01 -1.26829268e+00 -8.41463415e-01
   3.52439024e+00  1.95121951e+00  2.01219512e+00  2.82926829e+00
   1.82926829e+00  2.80487805e-01  7.31707317e-01  3.06097561e+00]
 [ 0.00000000e+00  0.00000000e+00 -1.19298246e+00 -2.36842105e+00
  -4.98771930e+01 -3.50877193e+01 -3.36842105e+01 -3.78771930e+01
  -3.08947368e+01 -1.05438596e+01 -1.71578947e+01 -3.55438596e+01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  1.36764706e+00
  -4.12205882e+01 -3.12941176e+01 -2.78823529e+01 -2.97500000e+01
  -2.44264706e+01 -5.95588235e+00 -1.55588235e+01 -3.48529412e+01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  -4.57591398e+02 -3.39268817e+02 -3.15483871e+02 -3.48107527e+02
  -2.78408602e+02 -7.71935484e+01 -1.61053763e+02 -3.55354839e+02]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
   0.00000000e+00 -5.89623085e-01 -5.49440737e+00 -5.37545822e+00
   9.79509352e-01 -1.97629006e+00  3.91437165e-01  2.36948961e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
   0.00000000e+00  0.00000000e+00 -5.11822095e+01 -5.17512355e+01
   1.67046070e+01 -1.40276981e+01  2.27443010e+00  1.92223816e+01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
   0.00000000e+00  0.00000000e+00  0.00000000e+00  4.27108289e+00
  -1.08681143e+01 -9.85147991e+00  2.15527337e-01  3.42030611e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
   0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
   1.79048924e+01  1.90945255e+01  2.27326051e+00 -8.91233352e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
   0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
   0.00000000e+00  2.79669600e+00 -2.86408797e+00 -2.05580848e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
   0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
   0.00000000e+00  0.00000000e+00 -4.66920170e+00 -3.40656688e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
   0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
   0.00000000e+00  0.00000000e+00  0.00000000e+00 -1.56278282e+00]]
```

(b) Use Gram-Schmidt algorithm on X to obtain Q and R. Find the inverse of X.

```
# 3.b QR factorization using Gram-Schmidt
import pandas as pd

data = pd.read_csv('/content/drive/MyDrive/臺大資料科學博士班/資料科學計算/作業
/CAmaxTemp.txt', sep = '   ').reset_index(drop=False)
data['Station'] = data['index'].map(lambda x:x.split('\t')[0])
data['Period'] = data['index'].map(lambda x:x.split('\t')[1])
data.drop('index', axis=1, inplace=True)
data.columns =
['JAN','FEB','MAR','APR','MAY','JUN','JUL','AUG','SEP','OCT','NOV','DEC','MAX',
'Station','Period']
data =
data[['JAN','FEB','MAR','APR','MAY','JUN','JUL','AUG','SEP','OCT','NOV','DEC']]

def QR_factorization(a):
  dim = a.shape
  Q = np.zeros(dim)
  R = np.zeros((dim[1], dim[1]))

  R[0,0] = np.sqrt(sum(x**2 for x in a[:, 0].reshape(-1,1)))
```

```
  Q[:, 0] = a[:, 0]/R[0,0]

  for k in range(1, dim[1]):
    R[0:k, k] = [sum(Q[:, i]*a[:,k]) for i in range(k)]
    q = a[:, k] -sum(R[i,k]*Q[:, i] for i in range(k))
    R[k,k] = np.sqrt(sum([x**2 for x in q.reshape(-1,1)]))
    Q[:, k] = q/R[k,k]

  return Q, R

Q, R = QR_factorization(data.to_numpy())
print(Q)
print(R)
```

Q 矩陣的執行結果如下

```
print(Q)
```

```
[[ 0.28349274  0.41801549 -0.13338368 -0.01835086 -0.07573112  0.07374759
   0.16905809  0.00439808 -0.36293233  0.19234199  0.03495295  0.71906609]
 [ 0.2662066   0.30788157 -0.13083007  0.04098403  0.31203346 -0.2262477
  -0.19186116 -0.50551463 -0.07387586 -0.56608893  0.19760886 -0.09835931]
 [ 0.26274937 -0.54218285 -0.34721145  0.42559874  0.424057   -0.18208588
   0.14187859  0.23498874 -0.07862965 -0.06290474 -0.09241969  0.16822773]
 [ 0.26966383  0.42732454  0.03023034  0.33811576 -0.21994792 -0.36358812
   0.3378933   0.27688154 -0.08892105  0.06846716 -0.17884271 -0.46200749]
 [ 0.32152226 -0.33820587  0.27972407 -0.04885911 -0.27608368 -0.21470518
   0.45881806 -0.50856167  0.27741834  0.10696881  0.09145163  0.11563433]
 [ 0.32843671 -0.22109016 -0.24637749 -0.04981402 -0.45894516  0.55502911
   0.07332271  0.0286386  -0.27916561 -0.36194726  0.00561264 -0.21496518]
 [ 0.27657828  0.17912953 -0.18339567  0.13350371  0.35307318  0.501664
   0.01605225 -0.2606104   0.26059304  0.50768821 -0.01454634 -0.26379784]
 [ 0.26274937  0.06666835  0.62859452  0.21957393  0.17117488  0.28100356
  -0.01381842  0.37140488  0.19681299 -0.23415777  0.36070264  0.11448294]
 [ 0.30423611  0.03874117 -0.4064652  -0.41829542 -0.12265774 -0.22505431
  -0.13231794  0.3525947   0.44663327  0.05918758  0.39400962  0.00101998]
 [ 0.27312106  0.05968656  0.06370716  0.06237169 -0.16298815  0.00958687
  -0.39054003  0.00709196  0.40521669 -0.15473484 -0.70426069  0.22480693]
 [ 0.29732166 -0.07837454  0.25569077 -0.66263503  0.39607115 -0.04283362
   0.14233755  0.13171313 -0.28280528  0.01491   -0.31405823 -0.16076444]
 [ 0.30769334 -0.20712657  0.20312517  0.11694412 -0.17146927 -0.20164377
  -0.62743832 -0.07427211 -0.37885851  0.38885659  0.17501643 -0.11338671]]
```

R 矩陣的執行結果如下

```
print(R)
```

```
[[ 2.89249028e+02  2.94777136e+02  3.13657061e+02  3.37978664e+02
   3.52180959e+02  3.74690974e+02  3.71990879e+02  3.65470546e+02
   3.74386738e+02  3.57719439e+02  3.20187766e+02  2.88059741e+02]
 [ 0.00000000e+00  8.21218709e+00  1.44991512e+01  1.45658254e+01
   2.29056467e+01  2.40700375e+01  2.63747123e+01  2.45536515e+01
   2.28135939e+01  1.30888564e+01  9.85940740e+00  1.65297080e+00]
 [ 0.00000000e+00  0.00000000e+00  7.35000049e+00  5.77051464e+00
   9.23032076e+00  1.13653023e+01  9.66301574e+00  7.83340281e+00
   2.33716284e+00  7.75436412e+00  2.57050229e+00  8.36385739e-03]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  7.27739031e+00
   1.38025434e+01  1.35022829e+01  1.58146848e+01  1.88125345e+01
   1.14718111e+01  6.19247454e+00 -2.89623144e+00 -3.18405937e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
   1.22961098e+01  1.48308903e+01  1.96232825e+01  2.00754331e+01
   9.16113917e+00  4.89332443e+00  2.14524191e+00  3.07842531e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
   0.00000000e+00  8.17740773e+00  6.01005340e+00  3.75425854e+00
   4.67362807e+00  1.75971170e+00 -2.40993109e+00 -3.66019171e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
   0.00000000e+00  0.00000000e+00  6.74451837e+00  4.44467776e+00
   8.86461121e-01 -2.23070907e+00  9.46514981e-01 -7.90217891e-01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
   0.00000000e+00  0.00000000e+00  0.00000000e+00  2.29174413e+00
   1.74354541e+00  1.97911418e+00  8.03536747e-01 -2.02888240e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
   0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
   8.15170897e+00  7.27381736e+00  1.30103711e+00 -3.28854308e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
   0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
   0.00000000e+00  2.67676599e+00 -2.36581097e-01 -7.47972944e-01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
   0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
   0.00000000e+00  0.00000000e+00  2.62504329e+00  1.64167244e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
   0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
   0.00000000e+00  0.00000000e+00  0.00000000e+00  1.77198798e-01]]
```

有了 Q,R 矩陣後，就可以運用 $X^{-1} = R^{-1}Q^T$ 來得到 X 的反矩陣

```
np.dot(np.linalg.inv(R), Q.T)
```

```
array([[-3.21819249e+00,  6.34210633e-01, -6.05174035e-01,
         1.87378226e+00, -4.26867098e-01,  1.00983246e+00,
         8.59157162e-01, -3.73852764e-01, -1.26172151e-01,
        -6.87830287e-01,  7.41058340e-01,  2.36531391e-01],
       [-1.99027728e+00,  1.97654172e-02, -5.93892622e-01,
         1.50129684e+00, -4.68932570e-01,  5.17866485e-01,
         1.25234283e+00, -7.53959287e-01,  3.22952060e-02,
        -8.68117940e-01,  5.37765641e-01,  6.82403054e-01],
       [ 2.25949419e+00, -3.04169769e-01,  6.15626602e-01,
        -1.43131343e+00,  2.53557735e-01, -6.22464229e-01,
        -1.20869984e+00,  4.73847477e-01, -1.04119552e-01,
         1.16047377e+00, -2.58091079e-01, -6.86347069e-01],
       [ 2.13589992e+00, -3.58447738e-01,  4.32000831e-01,
        -1.27783168e+00,  4.32222953e-01, -5.53365296e-01,
        -6.93560114e-01,  4.13175381e-01,  5.62329981e-02,
         3.53403636e-01, -6.93268481e-01, -1.96565872e-01],
       [-4.51661587e+00,  8.53436390e-01, -1.03995764e+00,
         2.78344255e+00, -6.22072998e-01,  1.21245807e+00,
         1.61097284e+00, -8.37364048e-01, -9.05238896e-02,
        -1.23296261e+00,  9.98957718e-01,  7.13113017e-01],
       [-4.58293324e-01,  1.00269845e-01, -1.56769094e-01,
         1.99263179e-01, -1.60331336e-01,  2.25202930e-01,
         2.04937441e-01,  1.97526828e-02,  3.18738634e-02,
        -2.51370667e-01,  4.81086453e-02,  1.65927792e-01],
       [-7.99212049e-01,  1.15321180e-01, -2.64909448e-01,
         5.20647687e-01,  1.16023687e-01,  1.66783516e-01,
         5.08064620e-01, -2.73587842e-01, -7.70325691e-02,
        -3.61796961e-01,  1.53473314e-01,  1.33182538e-01],
       [ 2.77993382e+00, -5.70730287e-01,  7.68731524e-01,
        -1.63265072e+00,  1.80213725e-01, -7.62304507e-01,
        -1.18450278e+00,  5.65646191e-01,  7.51494927e-02,
         9.04917962e-01, -5.05004996e-01, -4.75575973e-01],
       [ 1.11861710e+00, -6.61493224e-03,  3.07582775e-01,
        -8.08050845e-01,  1.87933915e-01, -2.81938573e-01,
        -5.87348144e-01,  2.65331171e-01,  1.01200255e-03,
         5.49947134e-01, -2.86225510e-01, -3.86023267e-01],
       [ 9.82656638e-01, -3.29254296e-01,  1.86197292e-01,
        -5.64885922e-01,  1.89319311e-01, -4.06961620e-01,
        -1.44531128e-01,  6.94884454e-02,  3.66678834e-02,
         2.02863809e-01, -2.08372099e-01,  7.72872542e-03],
       [-2.52448879e+00,  4.22418381e-01, -6.28933989e-01,
         1.56243618e+00, -3.73270754e-01,  7.60815932e-01,
         9.25481824e-01, -2.66637016e-01,  1.46496608e-01,
        -1.06169773e+00,  4.47747586e-01,  4.66848167e-01],
       [ 4.05796257e+00, -5.55078865e-01,  9.49372858e-01,
        -2.60728342e+00,  6.52568378e-01, -1.21313002e+00,
        -1.48871123e+00,  6.46070632e-01,  5.75613381e-03,
         1.26867076e+00, -9.07254692e-01, -6.39884179e-01]])
```

(c) Use power iteration method to find the largest eigenvalue-eigenvector pair of X.

```
# 3.c Power Iteration
def power_iteration(A, num_simulations: int):
    # Ideally choose a random vector
    # To decrease the chance that our vector
    # Is orthogonal to the eigenvector
    b_k = np.random.rand(A.shape[1])
```

```
    for _ in range(num_simulations):
        # calculate the matrix-by-vector product Ab
        b_k1 = np.dot(A, b_k)

        # calculate the norm
        b_k1_norm = np.linalg.norm(b_k1)

        # re normalize the vector
        b_k = b_k1 / b_k1_norm

    eig_val = np.dot(np.dot(b_k.T,A),b_k)/np.dot(b_k.T, b_k)
    eig_vec = b_k.reshape(A.shape[1],1)
    return eig_val, eig_vec

eig, eig_vec = power_iteration(data, 1000)
print(eig)
print(eig_vec)
```

可以找到最大的 eigenvalue 為 1167.7759013704458，而特徵向量的結果如下方截圖所示

```
print(eig_vec)

[[0.29756758]
 [0.28287902]
 [0.24957879]
 [0.29013861]
 [0.30377393]
 [0.30481108]
 [0.29475535]
 [0.28603568]
 [0.28885816]
 [0.27368297]
 [0.29154169]
 [0.29620859]]
```

(d) Use QR factorization to find all eigenvectors with REAL eigenvalues of X.

本題預設迭代的門檻值為 0.0001，迭代次數上限為 1000 次。怕會不收斂或是太早停止迭代，會將門檻值調成 0.00001，迭代次數上限調成 3000 次做嘗試。程式碼如下：

```
# 3.d Use QR factorization to find all eigenvectors with REAL eigenvalues of X
def QR_iteration(a, precision=0.0001, iter_ceil=1000):
    dim = a.shape
    real_value = []
    for i in range(dim[0]):
        times = 1
        a_0 = a.copy()
        Q_eigen = np.identity(dim[0])  # identity matrix

        while True:
            Q, R = QR_factorization(a_0)
```

```
            Q_eigen = np.dot(Q_eigen, Q) # converges to eigenvector
            ak = np.dot(R,Q) # converges to eigenvalue

            if np.abs(ak[i][i] - a_0[i][i]) < precision:
                print(f"Eigenvalue number {i+1} is found. Iteration:", times)
                real_value.append(ak[i][i])
                break
            elif times > iter_ceil:
                print(f"Eigenvalue number {i+1} is unable to converge.")
                break
            else:
                times = times + 1
                a_0 = ak

    return Q_eigen, real_value

Q_eigen, eig_real = QR_iteration(data.to_numpy(), precision=0.00001,
iter_ceil=3000)
```

程式執行的結果如下方所示，可以順利找到題目中說的 6 個實數 eigenvalue。其他 6 個就無法順利收斂。

```
Q_eigen, eig_real = QR_iteration(data.to_numpy(), precision=0.00001, iter_ceil=30
```

```
Eigenvalue number 1 is found. Iteration: 5
Eigenvalue number 2 is unable to converge.
Eigenvalue number 3 is unable to converge.
Eigenvalue number 4 is found. Iteration: 69
Eigenvalue number 5 is unable to converge.
Eigenvalue number 6 is unable to converge.
Eigenvalue number 7 is found. Iteration: 49
Eigenvalue number 8 is unable to converge.
Eigenvalue number 9 is unable to converge.
Eigenvalue number 10 is found. Iteration: 34
Eigenvalue number 11 is found. Iteration: 34
Eigenvalue number 12 is found. Iteration: 15
```

6 個實數的 eigenvalue 如下所示：

```
eig_real
```

```
[1167.7759013778389,
 -8.000987760051268,
 5.338919448003369,
 1.1173128943659278,
 -0.751429802863104,
 -0.3694889418458691]
```

4. Monthly Record-Breaking Temperature in California II: PCA and SVD *(30 points)*

(a) Standardize the data and compute the variance-covariance matrix

```
#%% 4. PCA and SVD
#read txt file as pandas dataframe
import pandas as pd
```

```
data = pd.read_csv('/content/drive/MyDrive/臺大資料科學博士班/資料科學計算/作業
/CAmaxTemp.txt', sep = '   ').reset_index(drop=False)
data['Station'] = data['index'].map(lambda x:x.split('\t')[0])
data['Period'] = data['index'].map(lambda x:x.split('\t')[1])
data.drop('index', axis=1, inplace=True)
data.columns =
['JAN','FEB','MAR','APR','MAY','JUN','JUL','AUG','SEP','OCT','NOV','DEC','MAX','Station'
,'Period']
data = data[['JAN','FEB','MAR','APR','MAY','JUN','JUL','AUG','SEP','OCT','NOV','DEC']]
# Standardize the data
df_standardized = (data - data.mean())/data.std()
# Compute the variance-covariance matrix
df_cov1 = df_standardized.cov()
```

下方為標準化過後的資料表

```
df_standardized
```

	JAN	FEB	MAR	APR	MAY	JUN	JUL	AUG	SEP	OCT	NOV	DEC
0	−0.185730	0.332858	0.510055	0.545400	0.915795	0.811345	1.014074	0.997747	0.636142	−0.041342	0.251580	0.011995
1	−0.928651	−0.625772	−0.486331	−0.545400	0.375326	0.084767	0.314713	0.341694	−0.212047	−1.033542	−0.467220	−0.275886
2	−1.077236	−1.903945	−2.621445	−2.415344	−2.507178	−2.385598	−1.783371	−1.462451	−2.586976	−2.521844	−2.048582	−1.139529
3	−0.780067	−0.306229	0.083032	0.389572	0.915795	0.230083	0.734329	0.997747	0.466504	−0.206708	−0.323460	−0.851648
4	1.448696	1.131716	1.079419	1.168715	0.375326	0.084767	0.174840	−0.150345	0.466504	1.281593	1.257901	1.307460
5	1.745865	1.611031	1.079419	1.324543	0.015013	0.520714	0.034968	−0.150345	0.805779	0.785492	1.114141	1.307460
6	−0.482899	−0.306229	−0.343991	−0.233743	0.735639	1.247292	1.433691	1.489787	1.314693	0.454759	−0.467220	−0.707708
7	−1.077236	−1.105087	−0.343991	−0.389572	0.555482	0.956661	0.874202	0.997747	0.127228	0.124025	−0.754741	−1.283470
8	0.705775	0.812173	0.367714	0.077914	−0.885769	−1.077757	−1.223882	−1.298438	0.466504	0.620125	1.114141	0.731698
9	−0.631483	−0.625772	−0.486331	−0.545400	−0.885769	−0.787126	−1.223882	−1.298438	−0.381685	−0.206708	−0.898501	−0.995589
10	0.408607	0.332858	0.510055	−0.233743	−0.165143	0.084767	0.174840	−0.150345	−0.720961	−0.041342	0.682861	0.875638
11	0.854359	0.652401	0.652396	0.857057	0.555482	0.230083	−0.524521	−0.314359	−0.381685	0.785492	0.539101	1.019579

下方為 12 個月份的共變異數矩陣 (Variance-Covariance Matrix)

```
df_cov
```

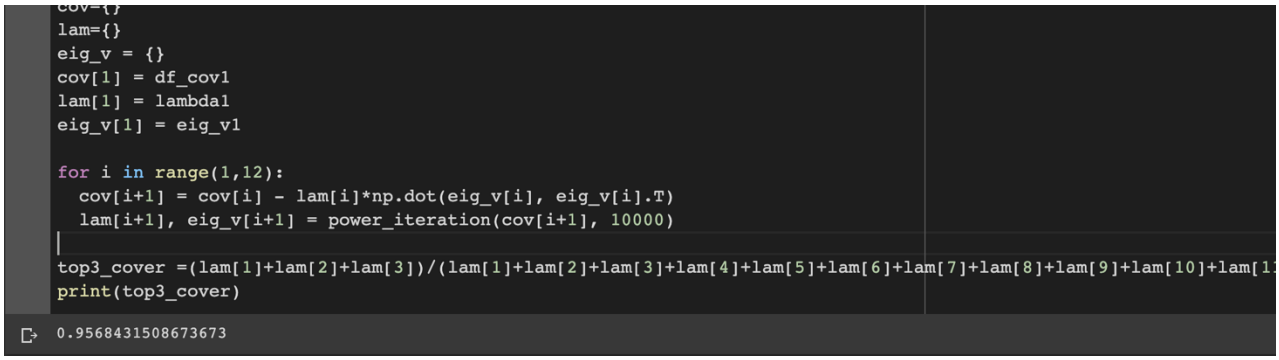
	JAN	FEB	MAR	APR	MAY	JUN	JUL	AUG	SEP	OCT	NOV	DEC
JAN	1.000000	0.932856	0.766674	0.752494	0.129583	0.138873	−0.083604	−0.218774	0.341993	0.722049	0.885975	0.947360
FEB	0.932856	1.000000	0.916745	0.895160	0.392290	0.361803	0.149831	0.011713	0.594425	0.824453	0.965211	0.909277
MAR	0.766674	0.916745	1.000000	0.958815	0.663043	0.610972	0.402263	0.268300	0.715063	0.902486	0.929669	0.766306
APR	0.752494	0.895160	0.958815	1.000000	0.700562	0.626836	0.421061	0.321798	0.748575	0.889026	0.858402	0.706547
MAY	0.129583	0.392290	0.663043	0.700562	1.000000	0.929176	0.876807	0.851297	0.770285	0.607348	0.419687	0.174254
JUN	0.138873	0.361803	0.610972	0.626836	0.929176	1.000000	0.918810	0.874082	0.786031	0.611136	0.367959	0.135801
JUL	−0.083604	0.149831	0.402263	0.421061	0.876807	0.918810	1.000000	0.977595	0.691876	0.369558	0.184172	−0.053536
AUG	−0.218774	0.011713	0.268300	0.321798	0.851297	0.874082	0.977595	1.000000	0.620323	0.254580	0.036975	−0.191190
SEP	0.341993	0.594425	0.715063	0.748575	0.770285	0.786031	0.691876	0.620323	1.000000	0.778456	0.576978	0.262491
OCT	0.722049	0.824453	0.902486	0.889026	0.607348	0.611136	0.369558	0.254580	0.778456	1.000000	0.832601	0.634566
NOV	0.885975	0.965211	0.929669	0.858402	0.419687	0.367959	0.184172	0.036975	0.576978	0.832601	1.000000	0.909077
DEC	0.947360	0.909277	0.766306	0.706547	0.174254	0.135801	−0.053536	−0.191190	0.262491	0.634566	0.909077	1.000000

(b) Find the top three principal components using power iteration. Calculate the cumulative percentage of the total eigenvalues that these three principal components cover.

```
def power_iteration(A, num_simulations: int):
    b_k = np.random.rand(A.shape[1])
```

```
    for _ in range(num_simulations):
        # calculate the matrix-by-vector product Ab
        b_k1 = np.dot(A, b_k)
        # calculate the norm
        b_k1_norm = np.linalg.norm(b_k1)
        # re normalize the vector
        b_k = b_k1 / b_k1_norm
    eig_val = np.dot(np.dot(b_k.T,A),b_k)/np.dot(b_k.T, b_k)
    eig_vec = b_k.reshape(A.shape[1],1)
    return eig_val, eig_vec


lambda1, eig_v1 = power_iteration(df_cov1, 10000)


cov={}
lam={}
eig_v = {}
cov[1] = df_cov1
lam[1] = lambda1
eig_v[1] = eig_v1


for i in range(1,12):
  cov[i+1] = cov[i] - lam[i]*np.dot(eig_v[i], eig_v[i].T)
  lam[i+1], eig_v[i+1] = power_iteration(cov[i+1], 10000)


top3_cover
=(lam[1]+lam[2]+lam[3])/(lam[1]+lam[2]+lam[3]+lam[4]+lam[5]+lam[6]+lam[7]+lam[8]+lam[9]+
lam[10]+lam[11]+lam[12])
print(top3_cover)
```

前三大的 eigenvalues 在所有 eigenvalues 的佔比為 95.68%，程式執行結果如下方截圖。

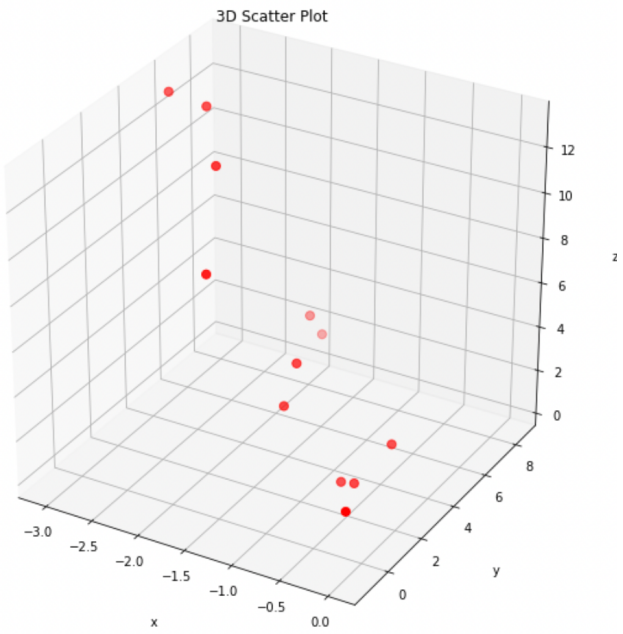


(c) Plot the data on a 3D space with three principal component axes. Provide the coordinates of the recast data.

```
#plot the data on a 3D space with three PC axes
data_centralized = data - data.mean()
```

```
eigv_top3 = np.concatenate((eig_v[1],eig_v[2],eig_v[3]), axis = 1)
eigv_proj = np.dot(eigv_top3.T, data_centralized).T

fig = plt.figure(figsize = (10,10))
ax = plt.axes(projection='3d')
ax.grid()

# ax.set_xlim([-4,1])
# ax.set_ylim([-1,9])
# ax.set_zlim([-1,14])

x = eigv_proj[:,0]
y = eigv_proj[:,1]
z = eigv_proj[:,2]

ax.scatter(x, y, z, c = 'r', s = 50)
ax.set_title('3D Scatter Plot')

# Set axes label
ax.set_xlabel('x', labelpad=20)
ax.set_ylabel('y', labelpad=20)
ax.set_zlabel('z', labelpad=20)

plt.show()
```

12 個點投影到三個特徵向量的 3D 散佈圖



12 個點的 X Y Z 座標以下方矩陣表示

```
eigv_proj
```

```
array([[-0.34434995,  3.73650354,  1.66731149],
       [-0.48428933,  2.29489698,  0.73704585],
       [-1.22593971,  2.26350039,  3.31513572],
       [-1.23240683,  3.03484537,  4.70184029],
       [-1.80917643,  1.03613105,  9.24096114],
       [-3.07888862,  5.59691376, 13.04341258],
       [-2.6217695 ,  5.35672582, 13.0129059 ],
       [-2.21575574,  3.74300159, 11.80328916],
       [-1.95743219,  8.59354144,  1.46709101],
       [-1.93901273,  7.7340607 ,  2.92871812],
       [-0.67191685,  2.57457494,  0.3573007 ],
       [ 0.04671243, -1.22633173,  2.7345607 ]])
```

(d) Find all principal components with their eigenvalues using SVD.

```python
# 4.e
def SVD_factorization(a):
  m,n = a.shape
  S = np.zeros([m,n])
  U = np.zeros([m,m])
  data = np.dot(a.T, a)
  eigenvals, eigenvecs = np.linalg.eig(data)
  eig_index = eigenvals.argsort()[::-1]
  eigenval_desc = eigenvals[eig_index]
  V = eigenvecs[:, eig_index]

  #把 S 矩陣算出來
  for i in range(m):
    for j in range(n):
        if i == j:
            S[i,j] = np.sqrt(eigenval_desc[i])

  #把 U 矩陣算出來
  for i in range(m):
    if np.diagonal(S)[i] != 0:
        u = (1/np.diagonal(S)[i])*np.dot(a, V[:,i].reshape(n,1))
        u = u.reshape(m,)
        U[:,i] = u
    else:
        U[:,i] = np.zeros(m)

  return U, S, V
```

```
U,S,V = SVD_factorization(data)
V
```

先以簡單的例子程式做出來 SVD 分解是正確的，如下圖所示，可以正確的將矩陣還原為原本的樣子。

```
a = np.array([[1,0,0,0,2],[0,0,3,0,0],[0,0,0,0,0],[0,2,0,0,0]])
U, S, V =SVD_factorization(a)
a_rebuilt = np.dot(U, S.dot(V.T))

print(a)
print(a_rebuilt)
```

```
[[1 0 0 0 2]
 [0 0 3 0 0]
 [0 0 0 0 0]
 [0 2 0 0 0]]
[[1. 0. 0. 0. 2.]
 [0. 0. 3. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 2. 0. 0. 0.]]
```

在進行 SVD 分解的時候，過程中得到的 V 矩陣，即為所有 Principal Component 的向量，V 的執行結果如下所示。

```
U,S,V = SVD_factorization(data)
V
```

```
array([[-0.24596734,  0.38039803, -0.23892106, -0.3413032 , -0.35697115,
        -0.09232829,  0.29126521, -0.23895372, -0.01380489, -0.04120992,
        -0.47493134,  0.34042684],
       [-0.25099717,  0.27755664, -0.00336995, -0.05545957,  0.20800546,
        -0.14073083,  0.30662376,  0.13169188, -0.1455818 ,  0.56184404,
         0.52775261,  0.25895498],
       [-0.26740722,  0.18050994,  0.23811455,  0.55021911,  0.15331841,
         0.17308457,  0.08898983, -0.16729338, -0.01969155,  0.387594  ,
        -0.47106915, -0.27308579],
       [-0.28820447,  0.12569739,  0.22311023,  0.27183826, -0.13043582,
        -0.58842026,  0.20493958, -0.21452036,  0.03876082, -0.45081186,
         0.26017771, -0.23187677],
       [-0.30102352, -0.22127372, -0.01023373,  0.27404673, -0.08559225,
        -0.36895133, -0.46970551,  0.29206279,  0.20796315,  0.1075777 ,
        -0.16675862,  0.50320107],
       [-0.32032224, -0.28617665,  0.0079096 ,  0.2052524 , -0.36180969,
         0.38927724,  0.39782148,  0.4728886 , -0.24049872, -0.21349525,
         0.05159862,  0.05380163],
       [-0.31823622, -0.4089893 , -0.28599273,  0.02371722,  0.20772217,
         0.23667411,  0.19846993, -0.43836293,  0.53195714, -0.01984931,
         0.15999602,  0.09701799],
       [-0.31264427, -0.41241923, -0.32065316, -0.19828718,  0.00490342,
        -0.22670817, -0.18283752, -0.19235621, -0.57556405,  0.18466622,
        -0.06404976, -0.31416335],
       [-0.31975289, -0.11878731,  0.44481156, -0.53161622,  0.41356271,
        -0.08101069,  0.11022727,  0.30785238,  0.1643962 , -0.09777116,
        -0.25009923, -0.13971263],
       [-0.30506417,  0.10621884,  0.4059485 , -0.23677708, -0.48397647,
         0.31148989, -0.42373501, -0.21845539,  0.09855698,  0.14647189,
         0.27298526, -0.10283044],
       [-0.27269418,  0.29660166, -0.04147222,  0.08847009,  0.44955239,
         0.30843927, -0.29975032, -0.12993827, -0.3615483 , -0.45258987,
         0.07585738,  0.2878246 ],
       [-0.24499316,  0.38508384, -0.53598472, -0.01351437, -0.00397104,
         0.04095247, -0.19485966,  0.39253692,  0.30647636, -0.04090201,
         0.05763632, -0.45995157]])
```

(e) SVD provides an extra information on U that PCA does not usually have. Is it any interpretation of this U matrix? If yes, please state it.

給定一資料矩陣 A，經 SVD 分解，可得 $A = U \cdot S \cdot V^T$。

V 為 $X^T X$ 的特徵向量矩陣 $(p \times p)$，U 為 XX^T 的特徵向量矩陣 $(n \times n)$。若 X 為一 $n \times p$ 的矩陣，表示有 p

個 feature 及 n 筆 sample，那麼 V 矩陣在描述的是 feature 與 feature 之間的相關性 (Covariance 有相關性的意涵存在)，而 U 矩陣在表達的則 sample 與 sample 之間的相關性，以本題而言，V 矩陣在描述的應該是月份與月份之間的關係，而 U 矩陣則是在描述各個地點與地點之間的關係。有些文獻會稱 U 矩陣為左奇異向量矩陣，V 矩陣則是右奇異向量矩陣。

(f) Conduct a rank-3 approximation (SVD version of X).

```
U,S,V = SVD_factorization(data)
U_3 = U[:,0:3]
S_3 = S[0:3,0:3]
V_3 = V.T[0:3,:]
A_3 = np.dot(U_3.dot(S_3), V_3)
```

Rank-3 approximation 的結果如下圖所示

```
[26] np.round(A_3)

array([[ 84.,  86.,  92., 100., 107., 114., 114., 112., 112., 106.,  93.,
         83.],
       [ 79.,  81.,  86.,  94., 102., 109., 111., 109., 106.,  99.,  88.,
         80.],
       [ 73.,  73.,  76.,  82.,  89.,  94.,  96.,  95.,  90.,  86.,  80.,
         75.],
       [ 79.,  82.,  90.,  97., 105., 113., 113., 111., 112., 104.,  90.,
         78.],
       [ 93.,  93.,  98., 104., 104., 110., 108., 105., 112., 110., 101.,
         92.],
       [ 93.,  94.,  98., 105., 104., 110., 108., 106., 112., 110., 102.,
         93.],
       [ 79.,  82.,  90.,  98., 108., 116., 117., 115., 114., 105.,  90.,
         78.],
       [ 75.,  80.,  88.,  95., 106., 113., 114., 112., 111., 102.,  87.,
         74.],
       [ 89.,  90.,  95., 101.,  98., 103.,  99.,  97., 108., 107.,  97.,
         87.],
       [ 78.,  81.,  88.,  94.,  96., 103., 100.,  98., 106., 101.,  88.,
         76.],
       [ 88.,  87.,  91.,  98., 102., 108., 108., 106., 106., 103.,  95.,
         89.],
       [ 89.,  90.,  95., 101., 102., 108., 106., 104., 109., 107.,  98.,
         89.]])
```

與跟原始的氣溫 data 比較，可以發現矩陣中各個的資料幾乎都可以還原的非常接近，顯示取前 3 個 rank 的 rank-3 approximation 效果已經非常好。

```
[27] data.to_numpy()

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]])
```

5. Monthly Record-Breaking Temperature in California III: ICA *(20 points)*

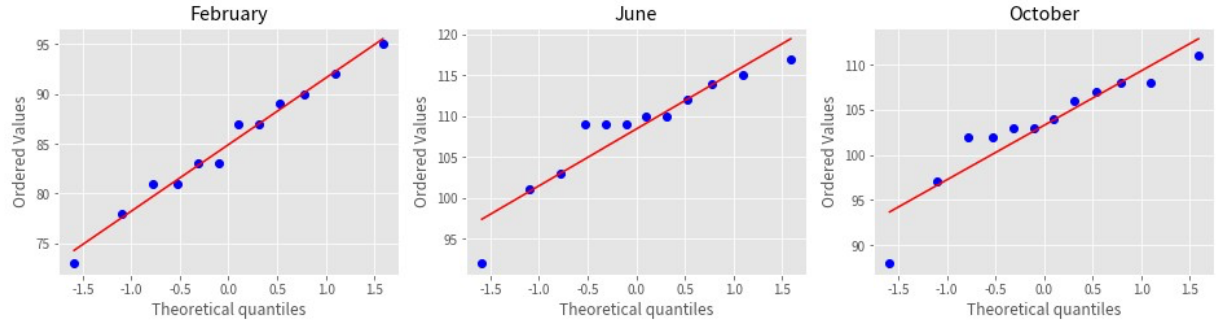
(a) Explain why the data is unlikely to be Gaussian.

```
# 5.a
import scipy.stats as stats
from scipy.stats import kurtosis
import pandas as pd
import numpy as np

data = pd.read_csv('/Users/wujhejia/Documents/Python/CAmaxTemp.txt', sep = '
').reset_index(drop=False)
data['Station'] = data['index'].map(lambda x:x.split('\t')[0])
data['Period'] = data['index'].map(lambda x:x.split('\t')[1])
data.drop('index', axis=1, inplace=True)
data.columns =
['JAN','FEB','MAR','APR','MAY','JUN','JUL','AUG','SEP','OCT','NOV','DEC','MAX','Station
','Period']
data = data[['JAN','FEB','MAR','APR','MAY','JUN','JUL','AUG','SEP','OCT','NOV','DEC']]
#data = data.to_numpy()

data_prime = data[['FEB','JUN','OCT']]
results = {}
months = {0:'FEB',1:'JUN',2:'OCT'}
for i in range(3):
  results[i] = kurtosis(data_prime[months[i]])
```

運用 Kurtosis 是否等於 0，來判斷三個月的變數是否符合常態分佈

```
✓ results ···
{0: -0.6224382359992937, 1: 0.7578802109307698, 2: 1.4866569765083826}
```

從程式執行的結果可以發現，三個月的資料都不等於 0，因此都不符合常態分佈。



上圖則是另外透過 quantile-quantile plot 來判斷是否接近常態，可以發現只有二月比較接近常態分佈，但六月跟十月都沒有接近常態，因為偏離紅色的標準線。

(b) Run the three preprocessing steps of ICA on X'.

```
#5.b
#Preprocessing Steps - Step1:Centering
def centering(data):
  output = data - data.mean()
  return output

D = centering(data_prime)
```

```
[69] print(D)
```

	FEB	JUN	OCT
0	2.083333	5.583333	-0.25
1	-3.916667	0.583333	-6.25
2	-11.916667	-16.416667	-15.25
3	-1.916667	1.583333	-1.25
4	7.083333	0.583333	7.75
5	10.083333	3.583333	4.75
6	-1.916667	8.583333	2.75
7	-6.916667	6.583333	0.75
8	5.083333	-7.416667	3.75
9	-3.916667	-5.416667	-1.25
10	2.083333	0.583333	-0.25
11	4.083333	1.583333	4.75

```
#Preprocessing Steps - Step2:Decorrelation
def decorrelation(D):
  D = D.to_numpy()
  D_cov = np.cov(D.T)
  eigs, V = np.linalg.eigh(D_cov)
  lam = np.diag(eigs)
  lam_inv = np.sqrt(np.linalg.inv(lam))
  U = np.dot(V, D.T).T
  return lam_inv, U

lam_inv, U = decorrelation(D)
```

```
[68] print(lam_inv)
     print(U)
```

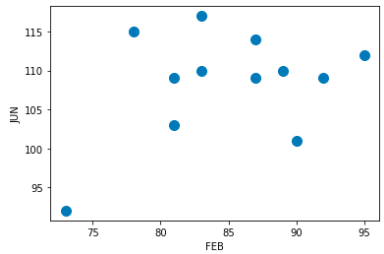
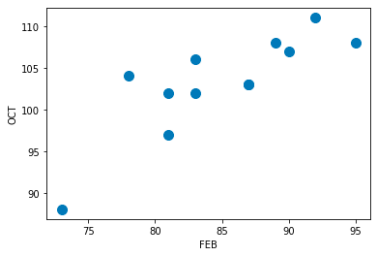
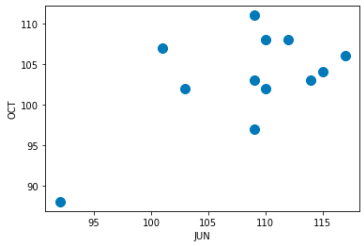
```
[[0.4606596  0.         0.        ]
 [0.         0.18537881 0.        ]
 [0.         0.         0.10582929]]
[[ 2.20727495 -4.7362095   2.87622074]
 [ 6.10880231  4.08216669  0.87264254]
 [ 5.77154135 24.49711166 -3.26453969]
 [ 2.75238599 -0.0530789  -0.40572102]
 [-8.11234596 -6.63036578  0.89692891]
 [-6.44271102 -7.99599217  5.62423086]
 [ 4.66324574 -7.81961427 -1.42038519]
 [ 7.52942497 -3.88351344 -4.4686842 ]
 [-9.43808036  2.40911977  0.16882914]
 [-0.20068225  5.91418475 -3.3504031 ]
 [-0.74121971 -0.82428161  1.87462229]
 [-4.09763603 -4.9595272   0.59625873]]
```

```
#Preprocessing Steps - Step3:Whitening
Z = np.dot(lam_inv, U.T)
Z.T
```

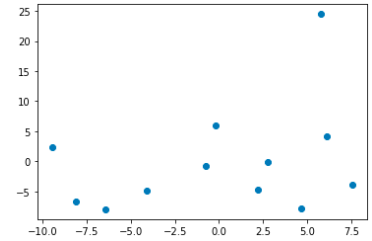
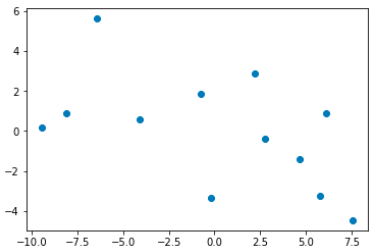
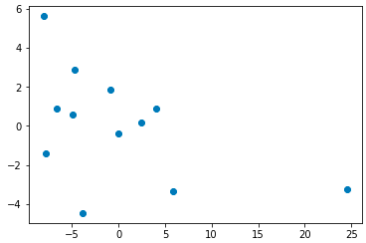
```
array([[ 1.0168024 , -0.87799288,  0.30438841],
       [ 2.81407842,  0.7567472 ,  0.09235114],
       [ 2.65871593,  4.54124542, -0.34548393],
       [ 1.26791303, -0.0098397 , -0.04293717],
       [-3.73703004, -1.22912932,  0.09492135],
       [-2.96789668, -1.48228752,  0.59520838],
       [ 2.14816891, -1.44959079, -0.15031836],
       [ 3.46850189, -0.7199211 , -0.4729177 ],
       [-4.34774232,  0.44659976,  0.01786707],
       [-0.0924462 ,  1.09636453, -0.3545708 ],
       [-0.34144997, -0.15280434,  0.19838995],
       [-1.88761537, -0.91939125,  0.06310164]])
```

(c) Provide the graphical illustration on the transformation, like the one in Lecture 04-2.

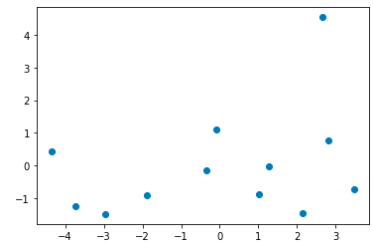
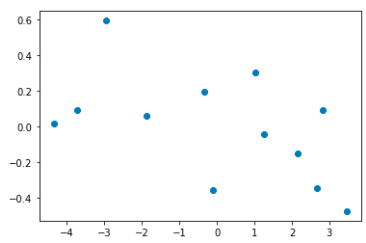
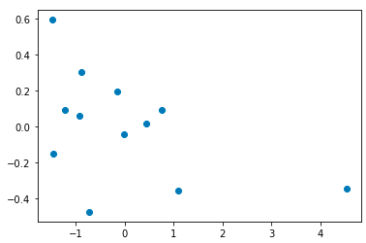
原始資料 Scatter Plots

FEB vs JUN	FEB vs OCT	JUN vs OCT
		

投影到 PCA Space

1st PC vs 2nd PC	1st PC vs 3rd PC	2nd PC vs 3rd PC
		

經過 Whitening 之後，其實整體資料呈現的分佈，與投影到 PCA 空間差不多，但整體尺度有變小。

1st column vs 2nd column	1st column vs 3rd column	2nd column vs 3rd column
		

(d) Run the fast ICA on Kurtosis Maximization to find the three independent components.

```
```

```
def fast_ICA1(a, precision=0.1**15, iter_ceil=100000):
    m,n = a.shape
    w0 = np.ones(n)
    #w0 = np.random.random(n)
    W = np.zeros(m)
    Z = np.zeros([m,n])
    times = 1
    while True:
        for i in range(m):
            W[i] = (w0.dot(a[i,]))**3
            Z[i,] = a[i,]*W[i]
        w1 = np.mean(Z, axis = 0) - 3*w0
        w1 = w1/np.linalg.norm(w1)
        if np.sum(np.abs(w1 - w0)) < precision:
            print('w converges. Iteration times:', times)
            break
        elif times > iter_ceil:
            print('w cannot converge.')
            break
        else:
            times = times + 1
            w0 = w1
    return w1

def fast_ICA2(a, u1, precision=0.1**15, iter_ceil=100000):
    m,n = a.shape
    w0 = np.ones(n)
    #w0 = np.random.random(n)
    W = np.zeros(m)
    Z = np.zeros([m,n])
    u1 = u1
    times = 1
    while True:
        for i in range(m):
            W[i] = (w0.dot(a[i,]))**3
            Z[i,] = a[i,]*W[i]
        w1 = np.mean(Z, axis = 0) - 3*w0
        w1 = w1/np.linalg.norm(w1)
        w1 = w1 - np.dot(w1,u1)*u1
        w1 = w1/np.linalg.norm(w1)
        if np.sum(np.abs(w1 - w0)) < precision:
        #if np.linalg.norm(w1 - w0) < precision:
```

```
            print('w converges. Iteration times:', times)
            break
        elif times > iter_ceil:
            print('w cannot converge.')
            break
        else:
            times = times + 1
            w0 = w1
    return w1

def fast_ICA3(a, u1, u2, precision=0.1**15, iter_ceil=100000):
    m,n = a.shape
    w0 = np.ones(n)
    #w0 = np.random.random(n)
    W = np.zeros(m)
    Z = np.zeros([m,n])
    u1 = u1
    u2 = u2
    times = 1
    while True:
        for i in range(m):
            W[i] = (w0.dot(a[i,]))**3
            Z[i,] = a[i,]*W[i]
        w1 = np.mean(Z, axis = 0) - 3*w0
        w1 = w1/np.linalg.norm(w1)
        w1 = w1 - np.dot(w1,u1)*u1 - np.dot(w1,u2)*u2
        w1 = w1/np.linalg.norm(w1)
        if np.sum(np.abs(w1 - w0)) < precision or np.sum(np.abs(w1)-np.abs(w0)) <
precision:
        #if np.linalg.norm(w1 - w0) < precision:
            print('w converges. Iteration times:', times)
            break
        elif times > iter_ceil:
            print('w cannot converge.')
            break
        else:
            times = times + 1
            w0 = w1
    return w1
```

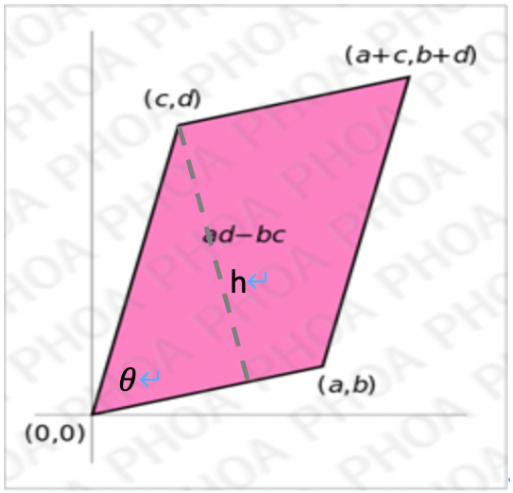
計算三組 Independent Components 的執行程式如下

```
u1 = fast_ICA1(Z.T)
u2 = fast_ICA2(Z.T, u1)
u3 = fast_ICA3(Z.T, u1, u2)
W = np.array([u1,u2,u3])
print(W)
```

執行結果如下

```
✓ u1 = fast_ICA1(Z.T) ···

w converges. Iteration times: 117
w converges. Iteration times: 25
w converges. Iteration times: 1
[[ 8.24050849e-01  5.62674082e-01 -6.58640639e-02]
 [-5.63882549e-01  8.25854942e-01  2.92690789e-04]
 [-5.45588522e-02 -3.68984042e-02 -9.97828562e-01]]
```

6. Determinant and Parallelogram *(Bonus 10 points)*



Proof.

令 $\tilde{x} = (a, b)$, $\tilde{y} = (c, d)$，$\tilde{x}$ 及 $\tilde{y}$ 的夾角為 θ，平行四邊形的高為 h

$$\cos\theta = \frac{\tilde{x} \cdot \tilde{y}}{\|\tilde{x}\|\|\tilde{y}\|} = \frac{ac + bd}{\sqrt{a^2 + b^2}\sqrt{c^2 + d^2}}$$

假設平行四邊形的面積為 $Area = \|\tilde{x}\| \cdot h = \|\tilde{x}\|\|\tilde{y}\| \sin\theta = \|\tilde{x}\|\|\tilde{y}\|\sqrt{1 - (\cos\theta)^2}$

$$= \sqrt{a^2 + b^2}\sqrt{c^2 + d^2}\sqrt{1 - \frac{(ac + bc)^2}{(a^2 + b^2)(c^2 + d^2)}}$$

$$= \sqrt{(a^2 + b^2)(c^2 + c^2) - (ac + bd)^2}$$

$$= \sqrt{a^2c^2 + a^2d^2 + b^2c^2 + b^2d^2 - a^2c^2 - b^2d^2 - 2abcd}$$

$$= \sqrt{(ad - bc)^2} = ad - bc$$

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