1 Some linear algebra exercises

Problem 1

· The product of two upper (lower) triangular matrices is upper (lower) triangular.

⇒ Suppose we have two upper triangular matrices A and B with n x n sizes. By the definition of upper triangular matrix, each entry of A and B is 0 if $a_{ij} = 0$ for $i > j$. Therefore, according to the row-column rule of matrix multiplication, let matrix $C = A \cdot B$, then entries of C $(C_{ij}) = 0$ if $i > j$ and C becomes an upper triangular matrix. By without loss of generality, we can see that the product of two lower triangular matrices are also lower triangular.

· The inverse of an upper (lower) triangular matrix is upper (lower) triangular.

⇒ Suppose we have an upper triangular matrix A. Assume that the inverse of A (A^{-1}) is not an upper triangular matrix. That means A^{-1} has non zero entries in its first row and first columns. However, according to the property of an inverse matrix, $A \cdot A^{-1} = I$. By the row-column rule for matrix multiplication, when we multiply rows and columns of A and A^{-1}, if inverse of A is not upper triangular matrix, then it is impossible to have 0 entries for the lower entries of I_{ij}. This contradicts to the fact that $A \cdot A^{-1} = I$, with all the entries are 0 except the diagonal. Therefore, we can conclude that the inverse of an upper triangular matrix is upper triangular and without the loss of generality it applies same for the lower triangular matrix.

· The product of two unit upper (lower) triangular matrices is unit upper (lower) triangular.

⇒ Suppose we have two unit upper triangular matrices A and B with the size of n x n. Since all the diagonal entries of unit triangular matrix are 1, we have

$$a_{11} = a_{22} = a_{33} = \cdots = a_{nn} = b_{11} = b_{22} = \ldots b_{nn} = 1$$

Therefore, the product of A and B will have

$$a_{11}b_{11} = a_{22}b_{22} = \cdots = a_{nn}b_{nn} = 1$$

Hence, we can see that the product of two unit upper triangular matrices will also have all the diagonal entries of 1, which is a unit upper triangular matrix.

· The inverse of a unit upper (lower) triangular matrix is unit upper (lower) triangular.

⇒ Suppose we have a unit upper triangular matrix A with size of n x n and let B = $[b_{ij}]$ denotes the inverse of the matrix A. Then we have

$$A \cdot B = I$$

, where I denotes the identity matrix. Then, we can express $AB = I$ as

$$AB = I \Rightarrow det(AB) = det(I)$$

$$\Rightarrow ab_{ii} = 1$$

, here ab_{ii} refers to the ith row and ith column of multiple of the matrices A and B.

$$\Rightarrow \sum_{k=1}^{n} a_{ik}b_{kj} = 1$$

Since $a_{ik}b_{kj} = 1$ only if i=k=j, $a_{ii}b_{ii} = 1$. Therefore, $b_{ii} = 1$. This proves that the matrix B, the inverse matrix of A, is also an upper unit triangular matrix.

· An orthogonal upper (lower) triangular matrix is diagonal.

⇒ Suppose we have an upper triangular matrix A with the size of n x n and assume that the matrix A is an orthogonal matrix. By the definition of orthogonal matrix, the matrix A is invertible and $A^{-1} = A^T$. Therefore, since the inverse of A is same as transpose of A, we can see that the inverse of matrix A is a lower triangular matrix. However, as we proved in the above bullet points, the inverse of an upper triangular matrix is also an upper triangular matrix. Therefore, the inverse of matrix A should be both lower and upper triangular matrix, which means that it is a diagonal matrix. As a result, $A^{-1} = A^T$ is a diagonal matrix which refers that $(A^T)^T = A$ is a diagonal matrix.

Problem 2

· The Sherman-Morrison formula

Suppose $Y = A + uu'$ and $X = (A + uu')^{-1}$ such that XY = YX = I.

(⇒) For XY = I,

$$XY = (A^{-1} - \frac{A^{-1}uu'A^{-1}}{1 + u'A^{-1}u})(A + uu')$$
$$= A^{-1}A + A^{-1}uu' - \frac{A^{-1}uu'A^{-1}A}{1 + u'A^{-1}u} - \frac{A^{-1}uu'A^{-1}uu'}{1 + u'A^{-1}u}$$

$$= A^{-1}A + A^{-1}uu' - \frac{(A^{-1}uu'A^{-1}A + A^{-1}uu'A^{-1}uu')}{1+u'A^{-1}u}$$

$$= A^{-1}A + A^{-1}uu' - \frac{(A^{-1}uu' + A^{-1}uu'A^{-1}uu')}{1+u'A^{-1}u}$$

$$I + A^{-1}uu' - \frac{A^{-1}u(u' + u'A^{-1}uu'}{1+u'A^{-1}u}$$

$$= I + A^{-1}uu' - \frac{A^{-1}u(1+u'A^{-1}u)u'}{1+u'A^{-1}u}$$

$$= I + A^{-1}uu' - A^{-1}uu' = I$$

($\Leftarrow$) For YX = I,

$$YX = (A+uu')(A^{-1} - \frac{A^{-1}uu'A^{-1}}{(1+u'A^{-1}u)})$$

$$= AA^{-1} + uu'A^{-1} - \frac{AA^{-1}uu'A^{-1}}{1+u'A^{-1}u} - \frac{uu'A^{-1}uu'A^{-1}}{1+u'A^{-1}u}$$

$$= I + uu'A^{-1} - \frac{uu'A^{-1}}{1+u'A^{-1}u} - \frac{uu'A^{-1}uu'A^{-1}}{1+u'A^{-1}u}$$

$$I = uu'A^{-1} - \frac{u(u'A^{-1} + u'A^{-1}uu'A^{-1})}{1+u'A^{-1}u}$$

$$= I + uu'A^{-1} - \frac{u(1+u'A^{-1}u)u'A^{-1}}{1+u'A^{-1}u}$$

$$= I + uu'A^{-1} - uu'A^{-1} = I$$

Hence, since XY = YX = I, we have proved the Sherman-Morrison Formula.

· The Woodbury formula

To prove this, we multiply $(A + UV')$ in both sides.

$$(A+UV')(A+UV')^{-1} = (A+UV')(A^{-1} - A^{-1}U(I_m + V'A^{-1}U)^{-1}V'A^{-1})$$

Then, the right hand side becomes I, so we need to show that left hand side becomes I.

$$(A+UV')(A^{-1} - A^{-1}U(I_m + V'A^{-1}U)^{-1}V'A^{-1})$$

$$= AA^{-1} - UV'A^{-1} + AA^{-1}U(I_m - V'A^{-1}U)^{-1}V'A^{-1} - UV'A^{-1}U(I_m - V'A^{-1}U)^{-1}V'A^{-1}$$

$$= I_m - UV'A^{-1} + U(I_m - V'A^{-1}U)^{-1}V'A^{-1} - UV'A^{-1}U(I_m - V'A^{-1}U)^{-1}V'A^{-1}$$

$$= I_m - U(I_m - (I_m - V'A^{-1}U^{-1})^{-1} + V'A^{-1}U(I_m - V'A^{-1}U)^{-1})V'A^{-1}$$
$$= I_m - U(I_m - (I_mV'A^{-1}U)^{-1}(I_m - V'A^{-1}U))V'A^{-1}$$
$$= I_m - U(I_m - I_m)V'A^{-1} = I$$

Therefore, since R.H.S is equal to L.H.S., we have proved the Woodbury formula.

· The binomial inversion theorem

If we multiply $(A + UBV)$ on both sides,

$$I = UBVA^{-1}(A^{-1} - A^{-1}UB(B + BVA^{-1}UB)^{-1}BVA^{-1})$$
$$= I + UBVA^{-1} - U(B + BVA^{-1}UB)(B + BVA^{-1}UB)^{-1}BVA^{-1}$$
$$= I + UBVA^{-1} - UBVA^{-1} = I$$

Thus, since R.H.S and L.H.S are same, we have proved the binomial inversion theorem.

The binomial inversion theorem is useful if $A = I_n$, $B = I_m$, and m =1 because in this case, U is a column vector, denoted as u, and V is a row vector, denoted as v^T. This implies that

$$(A + uv^T)^{-1} = A^{-1} - \frac{A^{-1}uv^TA^-1}{1 + v^TA^{-1}u}$$

Then, it becomes useful when we have a matrix A and suppose we know the inverse of A. Then if we need to invert matrices of the form $A + uv^T$, we can convert it efficiently.

For instance, when $A = I_n$ and $B = I_m$ and m = 1,

$$(I + uv^T)^{-1} = I - \frac{uv^T}{1 = v^Tu}$$

Problem 3

Show the matrix determinant lemma

$$det(A + UV') = det(A)det(I_m + V'A^{-1}U)$$

When A $= I_m$, we can convert the formula above as

$$\begin{bmatrix} I_m & 0 \\ V' & 1 \end{bmatrix} \begin{bmatrix} I_m + UV' & V \\ 0 & 1 \end{bmatrix} \begin{bmatrix} I_m & 0 \\ -V' & 1 \end{bmatrix} = \begin{bmatrix} I_m & U \\ 0 & 1 + V'U \end{bmatrix}$$

Here, in the left hand side, the first and the third matrix will have determinant of 1 because they both are unit lower triangular matrix. Therefore, eventually, the determinant of the second matrix and the right hand side matrix will have same determinant. As a result,

$$det(I_m + UV') = (1 + V'U)$$

Next, for the general case,

$$det(A + UV') = det(A)det(I_m + (A^{-1}U)V')$$

$$= det(A)(1 + V'(A^{-1}U))$$

Thus, we have shown that the matrix determinant lemma is true.

2. Programming exercises

Problem 4

Let a = 0.7, b = 0.2, and c = 0.1

In [ ]:
```
a = 0.7
b = 0.2
c = 0.1
```

In [ ]:
```
# Test (a+b)+c
(a+b)+c
```

Out[ ]:
```
0.9999999999999999
```

In [ ]:
```
# Test a+(b+c)
a+(b+c)
```

Out[ ]:
```
1.0
```

In [ ]:
```
# Test (a+c)+b
(a+c)+b
```

Out[ ]:
```
1.0
```

We were able to get exact value of 1.0 as an answer except when we compute (a+b)+c. Since the float type in Python follows double precision of the IEEE Standard 754 model, Python stores float type number in to the memory as slightly different value. Therefore, when we print out 0.7, 0.2, and 0.1 up to 10^-56 places, we would get the values below.

In [ ]:
```
# Print out the actual decimal points of each floats.
print(format(0.7, '.56f'))
print(format(0.2, '.56f'))
print(format(0.1, '.56f'))
```

```
0.69999999999999995559107901499373838305473327636718750000
0.20000000000000001110223024625156540423631668090820312500
0.10000000000000000555111512312578270211815834045410156250
```

The numbers above are the actual float point that Python uses for 0.7, 0.2, 0.1. Due to this, when we add 0.7 and 0.2 first and print out the result up to 10^-56 places, we get the number very slightly lower than 0.9. These subtle differences make Python to show us `0.9999...` when we add 0.7 and 0.2 first and add 0.1 later.

Problem 5

Create the vector v = (969,971,972,...,1022,1023), which has 54 elements

In [ ]:
```
import numpy as np

# Creat the vector v.
vector_v = np.array(range(969, 1024))

# delete 970
vector_v = np.delete(vector_v, [1])

print(vector_v)
```

```
[ 969  971  972  973  974  975  976  977  978  979  980  981  982  983
  984  985  986  987  988  989  990  991  992  993  994  995  996  997
  998  999 1000 1001 1002 1003 1004 1005 1006 1007 1008 1009 1010 1011
 1012 1013 1014 1015 1016 1017 1018 1019 1020 1021 1022 1023]
```

In [ ]:
```
ans = 0

# change elements to float and compute when i = 1
for i in range(len(vector_v)):
    ans += 2**float(vector_v[i])

print(ans)
```

```
1.7976931348623157e+308
```

In [ ]:
```
ans = 0

# compute when i = 2
for i in range(1, len(vector_v)):
    ans += 2**float(vector_v[i])

print(ans)
```

```
1.7976931348623157e+308
```

In [ ]:
```
ans = 0

# compute 2^v1 + vector sum
for i in range(1, len(vector_v)):
    ans += 2**float(vector_v[i])

2**float(vector_v[0]) + ans
```

Out[ ]:
```
1.7976931348623157e+308
```

We were able to get the same result for all three questions above, even if there were large difference in computing the formular. For instance, although we did not add `2**969` in the second bullet problem, we got the same result as the first bullet point. This happened because of the overflow error. The result is too large so that it exceeded the limit that Python could represent.

Problem 6

Create the vector x = (0.988,0.989,0.990,⋯1.010, 1.011, 1.012).

In [ ]:
```
# create a vector of range(0.988, 1.012)

vector_x = np.arange(0.988, 1.012, 0.001)
print(vector_x)
```

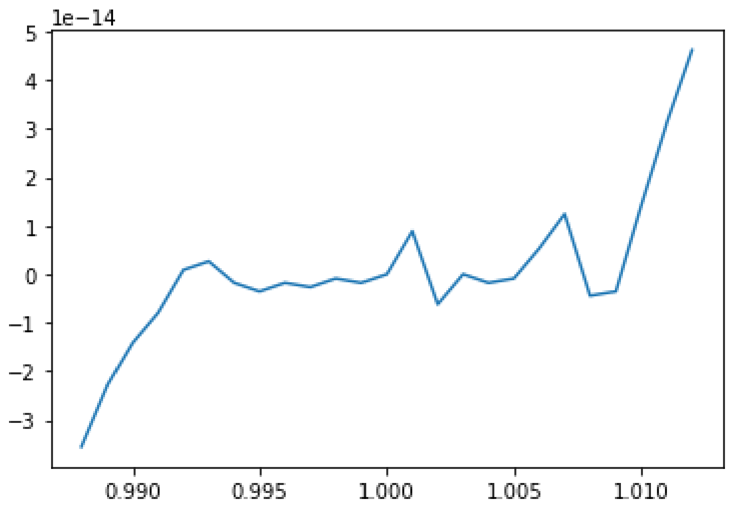
```
[0.988 0.989 0.99  0.991 0.992 0.993 0.994 0.995 0.996 0.997 0.998 0.999
 1.    1.001 1.002 1.003 1.004 1.005 1.006 1.007 1.008 1.009 1.01  1.011
 1.012]
```

In [ ]:
```
# Import matplot library module to use plot function
import matplotlib.pyplot as plt

# Formula
yval = vector_x**7 - 7*vector_x**6 + 21*vector_x**5 - 35*vector_x**4 + 35*vector_x*

# x represents each value in vector_x and y represents the y value of each correspondi
plt.plot(vector_x, yval)
```

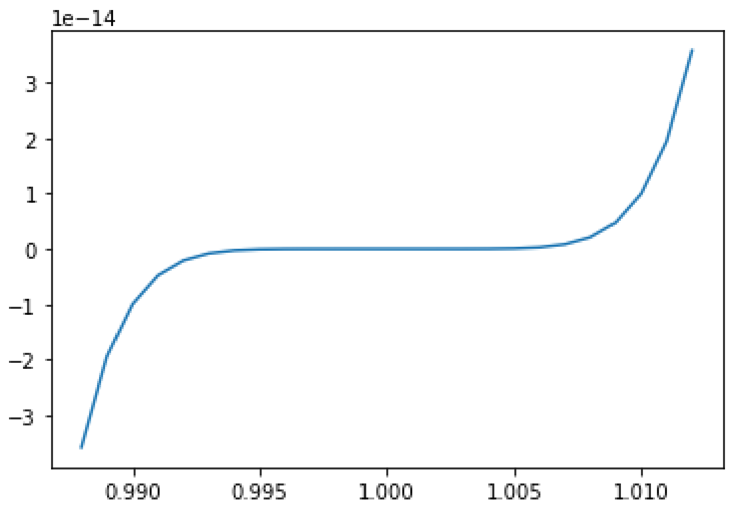
Out[ ]:
```
[<matplotlib.lines.Line2D at 0x26fe42d26d0>]
```



In [ ]:
```
# Formula
yval = (vector_x - 1)**7

plt.plot(vector_x, yval)
```

Out[ ]:
```
[<matplotlib.lines.Line2D at 0x26fe433b1f0>]
```



Compare to the second polynomial graph, the first polynomial graph has more deviation.

Problem 7

Basic exercises using numpy or R: let u = (1,2,3,3,2,1)'

In [ ]:
```
# Create a vector array with transpose
u = np.matrix([1,2,3,3,2,1]).T
```

In [ ]:
```
# a)
# Create an identity matirx
I = np.identity(6)

u = np.matrix([1,2,3,3,2,1]).T

# change the scalar value to float type to prevent error.
S = float(2/(u.T*u))
U = I - S*(u*u.T)

print(U)
```

```
[[ 0.92857143 -0.14285714 -0.21428571 -0.21428571 -0.14285714 -0.07142857]
 [-0.14285714  0.71428571 -0.42857143 -0.42857143 -0.28571429 -0.14285714]
 [-0.21428571 -0.42857143  0.35714286 -0.64285714 -0.42857143 -0.21428571]
 [-0.21428571 -0.42857143 -0.64285714  0.35714286 -0.42857143 -0.21428571]
 [-0.14285714 -0.28571429 -0.42857143 -0.42857143  0.71428571 -0.14285714]
 [-0.07142857 -0.14285714 -0.21428571 -0.21428571 -0.14285714  0.92857143]]
```

In [ ]:

```
# b)

# use numpy matmul function since it is matrix matrix multiplication
C = np.matmul(U,U)

lst = []

# append each element in the matrix C to lst variable except the diagonal
for i in range(len(C)):
    for j in range(len(C)):
        if C[i,j] <= 0:
            lst.append(C[i,j])

print("Matrix C: ")

print(C)
```

```
Matrix C:
[[ 1.00000000e+00 -2.46401539e-17 -5.55111512e-17 -4.38991247e-17
  -2.77555756e-17 -1.92589708e-17]
 [-2.46401539e-17  1.00000000e+00 -7.59030027e-17 -1.11022302e-16
  -4.48195902e-17 -2.77555756e-17]
 [-5.55111512e-17 -7.59030027e-17  1.00000000e+00 -7.22211406e-17
  -1.11022302e-16 -4.48903953e-17]
 [-4.38991247e-17 -1.11022302e-16 -7.22211406e-17  1.00000000e+00
  -8.97807905e-17 -8.32667268e-17]
 [-2.08166817e-17 -5.55111512e-17 -9.71445147e-17 -5.55111512e-17
   1.00000000e+00 -4.16333634e-17]
 [-1.38777878e-17 -2.77555756e-17 -5.55111512e-17 -8.32667268e-17
  -4.16333634e-17  1.00000000e+00]]
```

In [ ]:

```
print(f"The largest off-diagonal elements of C: {max(lst)}")
print(f"The smallest off-diagonal elements of C: {min(lst)}")
```

```
The largest off-diagonal elements of C: -1.3877787807814457e-17
The smallest off-diagonal elements of C: -1.1102230246251565e-16
```

In [ ]:

```
# c)
# Largest and Smallest diagonal elements of C

lst2 = []

# append each diagonal element in the lst2 variable
for i in range(len(C)):
    for j in range(len(C)):
        if C[i,j] > 0:
            lst2.append(C[i,j])

print(f"The largest diagonal elements of C: {max(lst2)}")
print(f"The smallest diagonal elements of C: {min(lst2)}")
```

```
The largest diagonal elements of C: 1.0
The smallest diagonal elements of C: 0.9999999999999998
```

In [ ]:

```
# e)
# Find maximum absolute row sum of the matrix U

# By using linalg.norm function in numpy,
# we can easily get the maximum absolute row sum
```

```
np.linalg.norm(U, np.inf) #inf-norm of matrix U
```

Out[ ]:
```
2.2857142857142856
```

In [ ]:
```
# f)
# Print the third row of U.

print(U[2])
```

```
[[-0.21428571 -0.42857143  0.35714286 -0.64285714 -0.42857143 -0.21428571]]
```

In [ ]:
```
# g)
# Print the elements of the second column below the diagonal

print(U[1,3])
```

```
-0.42857142857142855
```

In [ ]:
```
# h)
# compute P = AA'

# A => first three columns of U
A = U[0:3,].T

# Use numpy.matmul function
P = np.matmul(A,A.T)

print(P)
```

```
[[ 9.28571429e-01 -1.42857143e-01 -2.14285714e-01 -3.54025199e-17
  -2.88884563e-17 -1.44442281e-17]
 [-1.42857143e-01  7.14285714e-01 -4.28571429e-01 -1.52938886e-17
  -2.26576127e-18 -1.13288064e-18]
 [-2.14285714e-01 -4.28571429e-01  3.57142857e-01 -4.07837029e-17
  -4.30494642e-17 -2.15247321e-17]
 [-3.54025199e-17 -1.52938886e-17 -4.07837029e-17  6.42857143e-01
   4.28571429e-01  2.14285714e-01]
 [-2.88884563e-17 -2.26576127e-18 -4.30494642e-17  4.28571429e-01
   2.85714286e-01  1.42857143e-01]
 [-1.44442281e-17 -1.13288064e-18 -2.15247321e-17  2.14285714e-01
   1.42857143e-01  7.14285714e-02]]
```

In [ ]:
```
# i)
# Show that P is idempotent

PI = np.matmul(P,P) - P

# Calculate inf-norm of PP - P
print(np.linalg.norm(PI, np.inf))
```

```
2.43852557194454e-16
```

In [ ]:
```
# j)
# Let B the last three columns of U.
# Compute Q = BB'

# Last three columns of U
B = U[3:,].T

# Compute Q = BB'
Q = np.matmul(B,B.T)
print(Q)
```

```
[[ 7.14285714e-02  1.42857143e-01  2.14285714e-01 -2.38967009e-17
  -7.25751658e-18 -8.21338462e-18]
```

```
 [ 1.42857143e-01  2.85714286e-01  4.28571429e-01 -4.77934019e-17
  -1.45150332e-17 -1.64267692e-17]
 [ 2.14285714e-01  4.28571429e-01  6.42857143e-01 -2.83220159e-17
  -2.00378263e-17 -2.46401539e-17]
 [-2.38967009e-17 -4.77934019e-17 -2.83220159e-17  3.57142857e-01
  -4.28571429e-01 -2.14285714e-01]
 [-7.25751658e-18 -1.45150332e-17 -2.00378263e-17 -4.28571429e-01
   7.14285714e-01 -1.42857143e-01]
 [-8.21338462e-18 -1.64267692e-17 -2.46401539e-17 -2.14285714e-01
  -1.42857143e-01  9.28571429e-01]]
```

In [ ]:
```
# k)

# Show that Q is idempotent

QI = np.matmul(Q,Q) - Q

# Calculate inf-norm of QQ - Q
print(np.linalg.norm(QI, np.inf))
```

```
3.039660360150381e-16
```

In [ ]:
```
# l)
# compute P + Q
print(P + Q)
```

```
[[ 1.00000000e+00 -2.77555756e-17 -5.55111512e-17 -5.92992209e-17
  -3.61459728e-17 -2.26576127e-17]
 [-2.77555756e-17  1.00000000e+00 -5.55111512e-17 -6.30872905e-17
  -1.67807944e-17 -1.75596499e-17]
 [-5.55111512e-17 -5.55111512e-17  1.00000000e+00 -6.91057189e-17
  -6.30872905e-17 -4.61648860e-17]
 [-5.92992209e-17 -6.30872905e-17 -6.91057189e-17  1.00000000e+00
  -1.11022302e-16 -5.55111512e-17]
 [-3.61459728e-17 -1.67807944e-17 -6.30872905e-17 -1.11022302e-16
   1.00000000e+00  0.00000000e+00]
 [-2.26576127e-17 -1.75596499e-17 -4.61648860e-17 -5.55111512e-17
   0.00000000e+00  1.00000000e+00]]
```

Problem 8

In [ ]:
```
# import pandas to read the data
import pandas as pd

# read oringp data
df = pd.read_csv("oringp.dat", header=None, sep=",")

# rename the column name
df.rename(columns={0:"flight_num", 1:"date", 2:"O_rings", 3:"failed_num", 4:"temp"},

df
```

Out[ ]:

	flight_num	date	O_rings	failed_num	temp
0	1	4/12/81	6	0	66
1	2	11/12/81	6	1	70
2	3	3/22/82	6	0	69
3	5	11/11/82	6	0	68
4	6	4/04/83	6	0	67
5	7	6/18/83	6	0	72
6	8	8/30/83	6	0	73

	flight_num	date	O_rings	failed_num	temp
7	9	11/28/83	6	0	70
8	41-B	2/03/84	6	1	57
9	41-C	4/06/84	6	1	63
10	41-D	8/30/84	6	1	70
11	41-G	10/05/84	6	0	78
12	51-A	11/08/84	6	0	67
13	51-C	1/24/85	6	3	53
14	51-D	4/12/85	6	0	67
15	51-B	4/29/85	6	0	75
16	51-G	6/17/85	6	0	70
17	51-F	7/29/85	6	0	81
18	51-I	8/27/85	6	0	76
19	51-J	10/03/85	6	0	79
20	61-A	10/30/85	6	2	75
21	61-B	11/26/85	6	0	76
22	61-C	1/12/86	6	1	58
23	61-I	1/28/86	6	NA	31

In [ ]:
```
# Drop the row that contains NA (Here, it is the last row)
df.drop([23],inplace=True)
df.tail()
```

Out[ ]:

	flight_num	date	O_rings	failed_num	temp
18	51-I	8/27/85	6	0	76
19	51-J	10/03/85	6	0	79
20	61-A	10/30/85	6	2	75
21	61-B	11/26/85	6	0	76
22	61-C	1/12/86	6	1	58

In [ ]:
```
# Since failed number column type is object,
# convert it to int64 type
df['failed_num'] = pd.to_numeric(df['failed_num'])

df.dtypes
```

Out[ ]:
```
flight_num    object
date          object
O_rings        int64
failed_num     int64
temp           int64
dtype: object
```

In [ ]:
```
# Compute correlation between number of failures and temperature at launch
corr = df['failed_num'].corr(df['temp'])
```

```
print(f"The correlation between number of failures and temperature at launch: {corr}")
```

```
The correlation between number of failures and temperature at launch: -0.5613284258418
356
```

Problem 9

In [ ]:
```
# Create a function solving linear algebra for matrix
# use numpy.linalg.solve function

def solve_power1(A,k,b):
    if k>1:
        temp = np.linalg.matrix_power(A,k) # linalg.matrix_power to make A**k
        ans = np.linalg.solve(temp,b)
    else:
        ans = np.linalg.solve(A,b) # Use linalg.solve function
    return ans
```

In [ ]:
```
# create a random matrix A and random vector b
A =  np.random.randint(1,10,(3,3))
b =  np.random.randint(1,10,(3,1))
k = np.random.randint(1,10)

print(f"The vector x is for k as {k}: ")
print(solve_power(A,k,b))
print('₩n')

# To check if the answer is correct:
print("This is the vector b by multiplying the matrix A and the vector x:")
print(np.dot(np.linalg.matrix_power(A,k),solve_power(A,k,b)))

print("This is the actual value for vector b:")
print(b.astype(np.float))
```

```
The vector x is for k as 3:
[[-1.61319987]
 [-1.30403646]
 [ 2.87548828]]


This is the vector b by multiplying the matrix A and the vector x:
[[3.]
 [1.]
 [2.]]
This is the actual value for vector b:
[[3.]
 [1.]
 [2.]]
```

Problem 10

Since the question reveals that the expected value of an independent normal erros and random effects are 0,

$$E(\gamma) = 0 \; and \; E(e_i) = 0$$

Therefore,

$$E(y_i) = E(x_i'\beta + z_i'\gamma + \varepsilon_i) = x_i'\beta + z_i'E(\gamma) + E(e_i) = x_i'\beta + 0 + 0$$

and since $\mu_i = x_i'\beta$

The exepected value of the mixed effect model y is μ

For variance,

Since Z = $(z_1, \ldots, z_n)'$

$$Var(y_i) = Var(x_i'\beta + z_i'\gamma + \varepsilon_i) = Var(Z\gamma + \varepsilon_i)$$

$$= Var(Z\gamma) + Var(\varepsilon_i)$$

$$= ZVar(\gamma)Z' + Var(\varepsilon_i)$$

$$= Z(\sigma_1^2 I_q)Z' + \sigma_0^2 I_n$$

Therefore, the variance of the mixed effect model y is $(\sigma_1^2)ZZ' + \sigma_0^2 I_n$

Thus, we can say $y\ N(\mu, (\sigma_1^2)ZZ' + \sigma_0^2 I_n)$

The p.d.f of the normal distribution is

$$\frac{1}{\sigma\sqrt{2\pi}}exp(-\frac{1}{2}(\frac{x-\mu}{\sigma}))^2$$

Thus, if we substitue μ(Expected value) as μ and σ(variacne) as $(\sigma_1^2)ZZ' + \sigma_0^2 I_n$ for the mixed effect model, we would get the formula of negative log likelihood function.

$$L(y, \beta, \gamma) = -\frac{1}{2}\{nlog(2\pi) + log|Var(\gamma)| + (y - X\beta)'(V(\gamma))^{-1}(y - X\beta)\}$$

Thus, if we substitue our variance and expected value to the formula, we would get

$$-\frac{1}{2}\{nlog(2\pi) + log|(\sigma_1^2)ZZ' + \sigma_0^2 I_n| + (y-\mu)'((\sigma_1^2)ZZ' + \sigma_0^2 I_n)^{-1}(y-\mu)\}$$
$$= -\frac{n}{2}log(2\pi) - \frac{1}{2}log(det((\sigma_1^2)ZZ' + \sigma_0^2 I_n)) - \frac{1}{2}(y-\mu)'((\sigma_1^2)ZZ' + \sigma_0^2 I_n)^{-1}(y-\mu)$$

Hence, we proved that the log density function of the normal distribution of the mixed effect model is equal to

$$= -\frac{n}{2}log(2\pi) - \frac{1}{2}log(det((\sigma_1^2)ZZ' + \sigma_0^2 I_n)) - \frac{1}{2}(y-\mu)'((\sigma_1^2)ZZ' + \sigma_0^2 I_n)^{-1}(y-\mu)$$

In [ ]:
```
np.random.seed(777)

def dmvnorm_lowrank(y, mu, Z, sigma0, sigma1, log = False):
    pass
```

Problem 11

In `numpy`, linear algebra functions rely on `BLAS` and `LAPACK`. For example the function I have used above in the question 9 is `numpy.linalg.matrix_power(a,n)`. This helps user to calculate the square matrix to the power n. Since this function uses matrix multiplication to get the result, it is `level 3 BLAS` subroutine which does matrix-matrix operations.