

ONE DRIVE LINK:

[RC11 23155219 FINAL ASSIGNMENTS - OneDrive](#)

[\(sharepoint.com\)](#)

RC11_23155219

CEEL ASSIGNMENT

[github-codebook-feijiannughb/Ceel/final_ceel.py at main · UD-Skills-2023-24/github-codebook-feijiannughb](#)

Exercise one

requirements

1. Implement this algorithm in Python. Use the NumPy ndarray object for your matrices;
2. Give the asymptotic time complexity of the above algorithm or your implementation (they should be the same).
Justify and explain your answer.

Code:

```
✓ 0 秒 import numpy as np

def square_matrix_multiply(A, B):
    A = np.array(A)
    B = np.array(B)
    n = len(A)
    C = np.zeros((n, n))
    for i in range(n):
        for j in range(n):
            for k in range(n):
                C[i][j] += A[i][k] * B[k][j]
    return C
```

Explain:

1. Import numpy
2. By defining the function 'square_matrix_multiply' so that it accepts the arguments 'A' and 'B', A and B represent the two

matrices to be multiplied. matrices to be multiplied.

3. Convert 'A' and 'B' to numpy nparray objects.

4. Code: `n=len(A)`, used to get the dimension of matrix 'A'.

5. Code: `C=np.zeros((n,n))`, used to create an all-zero matrix of size (n,n), store the results of the calculation

6. The next three lines of for... code are triple nested loops used to compute each element of the matrix multiplication; their number of iterations determines the time complexity of this code, since i, j, and k represent the outer, middle, and innermost loops, and the number of iterations for each loop is the dimension n of the matrix, i.e., the total number of iterations is $n \times n \times n = n^3$, and the overall algorithm's The time complexity is $O(n^3)$

Suggestions:

1. Implement the algorithm with nested lists and compare the algorithms on different sizes of matrices

```
# compare different sizes of matrices
sizes = [2, 3, 4] # different sizes
for size in sizes:
    A = [[1]*size for _ in range(size)]
    B = [[2]*size for _ in range(size)]
    start_time = time.time()
    result = matrix_multiply_nested_list(A, B)
    end_time = time.time()
    duration = end_time - start_time
    print("Size: {}, Time taken: {:.10f} seconds".format(size, duration))
```

```
Size: 2, Time taken: 0.0000162125 seconds
Size: 3, Time taken: 0.0000159740 seconds
Size: 4, Time taken: 0.0000293255 seconds
```

Explain:

- (1) Define sizes, using 3 matrices of different sizes
- (2) Generate matrices A and B of size 'size × size' using elements 1 and 2, respectively, and create nested lists using list derivatives
- (3) Record the current time
- (4) Use custom function 'result...' to multiply matrices A and B, save the result in variable 'result'.
- (5) Record the time after running multiplication
- (6) Calculate the multiplication running time

2. Compare with built-in multiplication functions



```

sizes = [2, 3, 4] # different sizes
for size in sizes:
    A = np.ones((size, size))
    B = np.ones((size, size))
    start_time_builtin = time.time()
    result_builtin = A @ B
    end_time_builtin = time.time()
    duration_builtin = end_time_builtin - start_time_builtin

    A_list = [[1]*size for _ in range(size)]
    B_list = [[1]*size for _ in range(size)]
    start_time_custom = time.time()
    result_custom = matrix_multiply_nested_list(A_list, B_list)
    end_time_custom = time.time()
    duration_custom = end_time_custom - start_time_custom

    print("Size: {}, Built-in Time: {:.10f} seconds, Custom Time: {:.10f} seconds".format(size, duration_builtin, duration_custom))

```

Size: 2, Built-in Time: 0.0000529289 seconds, Custom Time: 0.0000131130 seconds
 Size: 3, Built-in Time: 0.0000319481 seconds, Custom Time: 0.0000183582 seconds
 Size: 4, Built-in Time: 0.0000221729 seconds, Custom Time: 0.0000333786 seconds

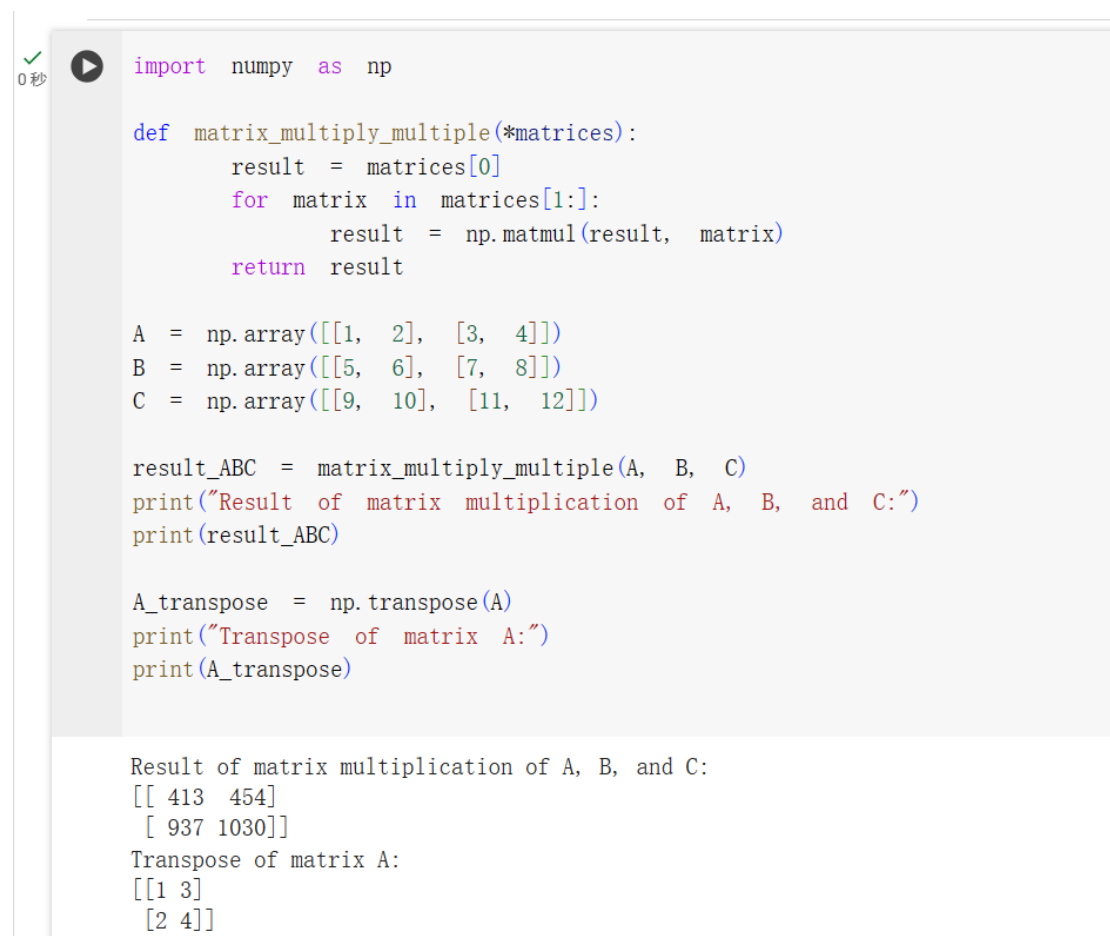
Explain:

- (1) Continuing the definition of sizes and matrix sizes mentioned in Recommendation 1
- (2) Unlike the previous code, here all the elements of matrices A and B have the value of 1 (just for simplicity and ease of understanding the code)

(3) Similarly perform the multiplication of matrices A and B and record the time before and after running the built-in multiplication, save the result and calculate the actual time taken by the built-in multiplication.

(4) Similarly, use a custom algorithm and calculate the time taken.

3. Look at multiplying more than two matrices, or at other operations on matrices



```
import numpy as np

def matrix_multiply_multiple(*matrices):
    result = matrices[0]
    for matrix in matrices[1:]:
        result = np.matmul(result, matrix)
    return result

A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])
C = np.array([[9, 10], [11, 12]])

result_ABC = matrix_multiply_multiple(A, B, C)
print("Result of matrix multiplication of A, B, and C:")
print(result_ABC)

A_transpose = np.transpose(A)
print("Transpose of matrix A:")
print(A_transpose)
```

Result of matrix multiplication of A, B, and C:

```
[[ 413  454]
 [ 937 1030]]
```

Transpose of matrix A:

```
[[1 3]
 [2 4]]
```

Explain:

(1) Define the function 'matrix...' to accept any number of

matrices and perform the practice matrix multiplication operations

(2) Create three matrices and use the function defined above, save the result in the variable 'result_ABC' and perform successive matrix multiplication operations on matrix ABC

(3) Transpose the matrix A using the function 'np.transpose()' (show other algorithms for matrices)

Exercise two

requirements

1. Describe and explain the algorithm. It should contain at least the following: recursiveness: How is it recursive? What is (the criteria for) the base case? How does the recursion step reduce to the base case? divide-and-conquer: How does this algorithm fit into the divide-andconquer approach? Explain each step of divide, conquer, and combine for this algorithm (as in slide 8 / pdf page 16 of the lecture slides).
2. Implement the recursive algorithm in Python. Reflect on which steps of the pseudocode were straightforward to implement and which hid a lot of complexity behind their language.
3. Do a complexity analysis for the SMMRec algorithm. First comment on the complexity of the base case, divide step, conquer step, and combine step separately, then put it all together.

Code:

```
✓ 0秒 ▶ def split_matrix(A, n):
    A11 = [row[:n] for row in A[:n]]
    A12 = [row[n:] for row in A[:n]]
    A21 = [row[:n] for row in A[n:]]
    A22 = [row[n:] for row in A[n:]]
    return A11, A12, A21, A22

def matrix_add(A, B):
    n = len(A)
    return [[A[i][j] + B[i][j] for j in range(n)] for i in range(n)]
```

```
✓ 0秒 ▶ def combine_matrix(A, B, C, D, n):
    top = [A[i] + B[i] for i in range(n)]
    bottom = [C[i] + D[i] for i in range(n)]
    return top + bottom

def square_matrix_multiply_recursive(A, B):
    n = len(A)
    if n == 1:
        return [[A[0][0] * B[0][0]]]
    else:
        mid = n // 2
        A11, A12, A21, A22 = split_matrix(A, mid)
        B11, B12, B21, B22 = split_matrix(B, mid)

        # Recursive multiplications
        C11 = matrix_add(square_matrix_multiply_recursive(A11, B11),
                          square_matrix_multiply_recursive(A12, B21))
        C12 = matrix_add(square_matrix_multiply_recursive(A11, B12),
                          square_matrix_multiply_recursive(A12, B22))
        C21 = matrix_add(square_matrix_multiply_recursive(A21, B11),
                          square_matrix_multiply_recursive(A22, B21))
        C22 = matrix_add(square_matrix_multiply_recursive(A21, B12),
                          square_matrix_multiply_recursive(A22, B22))

        # Combining quarters
        C = combine_matrix(C11, C12, C21, C22, mid)
        return C

# Test the function
A = [[1, 2], [3, 4]]
B = [[5, 6], [7, 8]]
result = square_matrix_multiply_recursive(A, B)
print("Result of matrix multiplication:")
for row in result:
    print(row)
```

```
Result of matrix multiplication:
[19, 22]
[43, 50]
```

Explain:

1. Recursiveness:

‘square_matrix_multiply_recursive’

(1) How is it recursive?

In the 'square...' function, assuming that the dimension of the input matrix is 1, then the product of the elements

is computed directly and the result is returned. If the dimension of the input matrix is greater than 1, then the original matrix is decomposed into four sub-matrices and the multiplication of these sub-matrices is performed recursively.

Code:

```
else:
    mid = n // 2
    A11, A12, A21, A22 = split_matrix(A, mid)
    B11, B12, B21, B22 = split_matrix(B, mid)

    # Recursive multiplications
    C11 = matrix_add(square_matrix_multiply_recursive(A11, B11),
                     square_matrix_multiply_recursive(A12, B21))
    C12 = matrix_add(square_matrix_multiply_recursive(A11, B12),
                     square_matrix_multiply_recursive(A12, B22))
    C21 = matrix_add(square_matrix_multiply_recursive(A21, B11),
                     square_matrix_multiply_recursive(A22, B21))
    C22 = matrix_add(square_matrix_multiply_recursive(A21, B12),
                     square_matrix_multiply_recursive(A22, B22))
```

(2) The base case

In this code, the base case is to directly compute the product of the elements when the dimension of the matrix is reduced to 1, i.e. when the matrix is a single element, and return the result.

Code:

```
if n == 1:
    return [[A[0][0] * B[0][0]]]
```

(3) The recursion step reduce to the base case

-In the recursive step, the original matrix is decomposed

into four submatrices and multiplication is performed recursively on these submatrices.

- Each recursive call halves the size of the problem until the base case (matrix dimension of 1) is reached.

- When the dimension of the matrix drops to 1, the algorithm computes the product directly based on the base case and returns the result, thus reducing the recursive step to the base case.

Thus, this code exemplifies the process of recursion by recursively breaking the original problem into smaller subproblems and not stopping the recursion until the base case is reached.

2. Divide-and-conquer:

(1) Divide

In the 'square_matrix_multiply_recursive' function, the splitting step is implemented by calling the 'split_matrix' function, which splits the original matrix into four equal-sized sub-matrices according to the middle rows and columns

```
else:
    mid = n // 2
    A11, A12, A21, A22 = split_matrix(A, mid)
    B11, B12, B21, B22 = split_matrix(B, mid)
```

(2) Conquer

The conquest step computes the product of the submatrices by recursively calling the function to multiply the submatrices

```
C11 = matrix_add(square_matrix_multiply_recursive(A11, B11),
                  square_matrix_multiply_recursive(A12, B21))
C12 = matrix_add(square_matrix_multiply_recursive(A11, B12),
                  square_matrix_multiply_recursive(A12, B22))
C21 = matrix_add(square_matrix_multiply_recursive(A21, B11),
                  square_matrix_multiply_recursive(A22, B21))
C22 = matrix_add(square_matrix_multiply_recursive(A21, B12),
                  square_matrix_multiply_recursive(A22, B22))
```

(3) Combine

The combine step combines the product of the four sub-matrices into the product of the original matrix by calling the 'combine_matrix' function, so that the results of the four sub-matrices are combined into a single matrix in the order of top-left, top-right, bottom-left, bottom-right

```
# Combining quarters
C = combine_matrix(C11, C12, C21, C22, mid)
```

3. Reflect on which steps of the pseudocode were straightforward to implement and which hid a lot of complexity behind their language

In the divide-and-conquer of recursive algorithms, the three types of pseudo-code, dividing matrices, adding matrices, and merging matrices, can be implemented directly, with a great deal of complexity hidden behind

recursive multiplication.

Of the common types of recursive algorithms, tree traversal and backtracking can be implemented directly, dynamic programming, search algorithms, and branch delimitation are more complex.

4. Do a complexity analysis for the SMMRec algorithm

(1) Complexity of the base case:

In the base case, i.e., when the dimension of the matrix is reduced to 1, the algorithm directly computes the product of the matrix elements and returns the result. Thus, the complexity of the base case is of constant order, i.e., $O(1)$.

(2) Complexity of the segmentation step:

In the partitioning step, the original matrix is decomposed into four equal-sized submatrices. This step involves a slicing operation on the matrix and hence its complexity depends on the complexity of the slicing operation. Since the slicing operation extracts the elements according to the index range, its complexity is linear, i.e., $O(n)$.

(3) Complexity of conquest step:

In the conquest step, the multiplication operation is

performed recursively on the four submatrices. Assuming the dimension of the matrix is $n \times n$, the multiplication of each submatrix requires n multiplications and $n-1$ additions. Since there are four submatrices, the total number of multiplication and addition operations is $4n^2 - 4n$. hence the complexity of the conquest step is $O(n^2)$.

(4) Complexity of combination step:

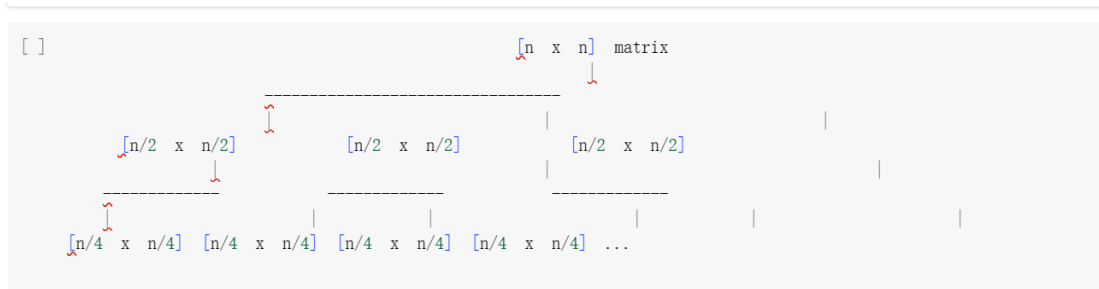
In the combination step, the results of the four submatrices are combined into the product of the original matrices. This step involves concatenation operations on lists and hence its complexity depends on the complexity of list concatenation. The complexity of list joining is linear i.e. $O(n)$.

Putting together the complexity of each of the above steps:

The total time complexity is the highest complexity among the segmentation step, the conquest step and the combination step, i.e., $O(n^2)$. Therefore, the total time complexity of the SMMRec algorithm is $O(n^2)$.

Suggestions:

1. Do a tree analysis



2. Test and compare the practical speed with the non-recursive algorithm

```

import numpy as np

def square_matrix_multiply_non_recursive(A, B):
    n = len(A)
    C = np.zeros((n, n), dtype=int)
    for i in range(n):
        for j in range(n):
            for k in range(n):
                C[i][j] += A[i][k] * B[k][j]
    return C

```

```
✓ 0 秒 ▶ import time
#Generate a random matrix of size n x n
def generate_matrix(n):
    return np.random.randint(1, 10, size=(n, n))
#Testing the execution time of a recursive algorithm
def test_recursive_algorithm(n):
    A = generate_matrix(n)
    B = generate_matrix(n)
    start_time = time.time()
    square_matrix_multiply_recursive(A, B)
    end_time = time.time()
    return end_time - start_time
#Testing the execution time of a non-recursive algorithm
def test_non_recursive_algorithm(n):
    A = generate_matrix(n)
    B = generate_matrix(n)
    start_time = time.time()
    square_matrix_multiply_non_recursive(A, B)
    end_time = time.time()
    return end_time - start_time
#Compare the execution time of the two algorithms
matrix_sizes = [2, 4, 8, 16, 32] # test a different size matrix
for size in matrix_sizes:
    print("Matrix size:", size)
    print("Recursive algorithm time:", test_recursive_algorithm(size))
    print("Non-recursive algorithm time:", test_non_recursive_algorithm(size))
    print("-----")

Matrix size: 2
Recursive algorithm time: 7.557868957519531e-05
Non-recursive algorithm time: 2.1457672119140625e-05
-----
Matrix size: 4
Recursive algorithm time: 0.0003025531768798828
Non-recursive algorithm time: 9.72747802734375e-05
-----
Matrix size: 8
Recursive algorithm time: 0.0018432140350341797
Non-recursive algorithm time: 0.0007159709930419922
-----
Matrix size: 16
```

Comments have been used in the code to explain what each part of the code does.

Exercise three

Requirements:

1. Reflect on the difference between (complexity of) addition/subtraction and multiplication on matrices.
2. Do a complexity analysis of the Strassen algorithm.
 - Instead of starting from scratch, you can also take your result from Exercise 2 and adapt to the optimisation; explain what changes in the complexity formula with these optimisations.

Code:

✓
0 秒



```
def matrix_sub(A, B):
    n = len(A)
    return [[A[i][j] - B[i][j] for j in range(n)] for i in range(n)]

def strassen_matrix_multiply(A, B):
    n = len(A)
    if n == 1:
        return [[A[0][0] * B[0][0]]]
    else:
        # Splitting and recombining matrices like in Exercise 2
        mid = n // 2
        A11, A12, A21, A22 = split_matrix(A, mid)
        B11, B12, B21, B22 = split_matrix(B, mid)

        # Strassen's multiplication operations
        P1 = strassen_matrix_multiply(matrix_add(A11, A22), matrix_add(B11, B22))
        P2 = strassen_matrix_multiply(matrix_add(A21, A22), B11)
        P3 = strassen_matrix_multiply(A11, matrix_sub(B12, B22))
        P4 = strassen_matrix_multiply(A22, matrix_sub(B21, B11))
        P5 = strassen_matrix_multiply(matrix_add(A11, A12), B22)
        P6 = strassen_matrix_multiply(matrix_sub(A21, A11), matrix_add(B11, B12))
        P7 = strassen_matrix_multiply(matrix_sub(A12, A22), matrix_add(B21, B22))

        # Combining results
        C11 = matrix_add(P1, P4)
        C11 = matrix_sub(C11, P5)
        C11 = matrix_add(C11, P7)
        C12 = matrix_add(P3, P5)
        C21 = matrix_add(P2, P4)
        C22 = matrix_add(P1, P3)
        C22 = matrix_sub(C22, P2)
        C22 = matrix_sub(C22, P6)

        C = combine_matrix(C11, C12, C21, C22, mid)
    return C
```

✓
0 秒



```
#test
def test_strassen_matrix_multiply():
    A = [[1, 2, 3, 4],
          [5, 6, 7, 8],
          [9, 10, 11, 12],
          [13, 14, 15, 16]]

    B = [[17, 18, 19, 20],
          [21, 22, 23, 24],
          [25, 26, 27, 28],
          [29, 30, 31, 32]]

    result = strassen_matrix_multiply(A, B)
    print("Result of matrix multiplication:")
    for row in result:
        print(row)

test_strassen_matrix_multiply()
```



```
Result of matrix multiplication:
[250, 260, 270, 280]
[618, -44, 670, -56]
[986, 1028, -210, -232]
[1354, 724, 190, -568]
```

Explain:

1. Addition and subtraction

'The matrix_sub function performs a subtraction operation on a matrix. This operation involves traversing all the elements of two matrices and performing a subtraction. Since only the elements at the corresponding positions need to be subtracted, its complexity is linear and proportional to the size of the matrices. Thus, the time complexity of subtraction of a matrix is $O(n^2)$, where n is the dimension of the matrix

2. Multiplication

In `strassen_matrix_multiply` function, we perform multiplication of matrices. This operation involves multiplication of two matrices and is more complex than addition and subtraction.

For normal matrix multiplication, the time complexity is usually $O(n^3)$, because each element of the result matrix needs to be multiplied by the corresponding row and column elements and then added together. But Strassen's algorithm reduces the number of multiplications to $O(n^{\log_2 7})$ by recursively

3. Strassen algorithm

When analysing the Strassen algorithm in Complexity Analysis Exercise 3, the main focus is on event complexity.◦

(1) Addition, subtraction and splitting of matrices are of linear time complexity, i.e. $O(n^2)$

(2) Due to the recursive splitting of the original matrix into smaller submatrices and their multiplication, at each level of recursion, seven multiplication operations (P1-P7) are run, all involving matrices of size $1/4$ the size of the original matrix, and the number of multiplication operations grows

exponentially as the depth of recursion increases

(3) On top of the above, the process of combining the four sub-matrices, i.e. addition and subtraction processes, so the complexity is also $O(n^2)$

Overall complexity: each level of recursion quarter the size of the matrix, so the recursion depth is $\log_2 n$, and there are 7 multiplications in each level of recursion, plus splitting and combining of matrices, the overall complexity can be approximated as $O(n^{\log_2 7})$

Explain code:

1. 'matrix_sub(A, B)' function is used to compute the result of subtracting two matrices A and B. It first gets the size of matrix A, then uses list derivatives to compute the difference of the two matrices element by element, and returns the results

```
def matrix_sub(A, B):  
    n = len(A)  
    return [[A[i][j] - B[i][j] for j in range(n)] for i in range(n)]
```

2. 'strassen_matrix_multiply(A, B)' function is the implementation of matrix multiplication in 'Strassen' algorithm. 'A' and 'B' are the input matrices. It checks the size of the input matrices and returns their product directly

if the size is 1. Otherwise, it splits the input matrix into four submatrices (using 'split_matrix') and recursively calls the 'strassen_matrix_multiply' function to compute the seven intermediate matrices of the product (P1-P7). It then uses these intermediate matrices to compute the four sub-matrices 'C11' 'C12' of the result matrix according to the formulas defined in the 'Strassen' algorithm. 'C21' 'C22', these four sub-matrices are combined to form the final product matrix 'C'.

```
def strassen_matrix_multiply(A, B):  
    n = len(A)  
    if n == 1:  
        return [[A[0][0] * B[0][0]]]  
    else:  
        # Splitting and recombining matrices like in Exercise 2  
        mid = n // 2  
        A11, A12, A21, A22 = split_matrix(A, mid)  
        B11, B12, B21, B22 = split_matrix(B, mid)
```

Suggestions:

Compared to the pseudo-code, I have optimised the splitting and combining of matrices by adding some functions, instead of defining the intermediate matrices through the given variables S1-S10 in the pseudo-code

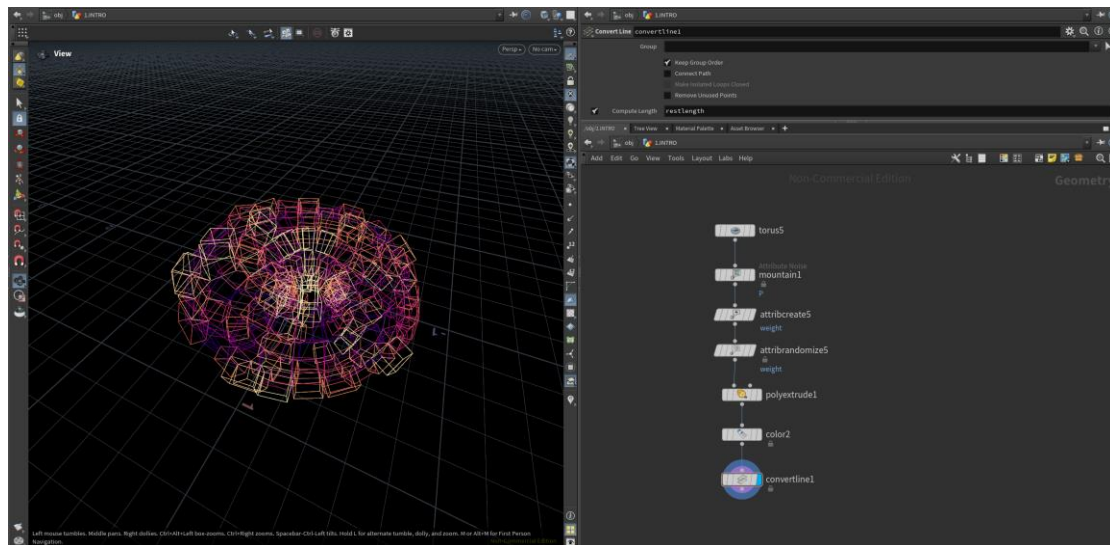
RC11_23155219

Houdini ASSIGNMENT

[github-codebook-feijiannughb/Joris_Houdini](https://github.com/feijiannughb/Joris_Houdini) at main · [UD-Skills-2023-24/github-codebook-feijiannughb](https://github.com/UD-Skills-2023-24/github-codebook-feijiannughb)

1 houdini fundamentals

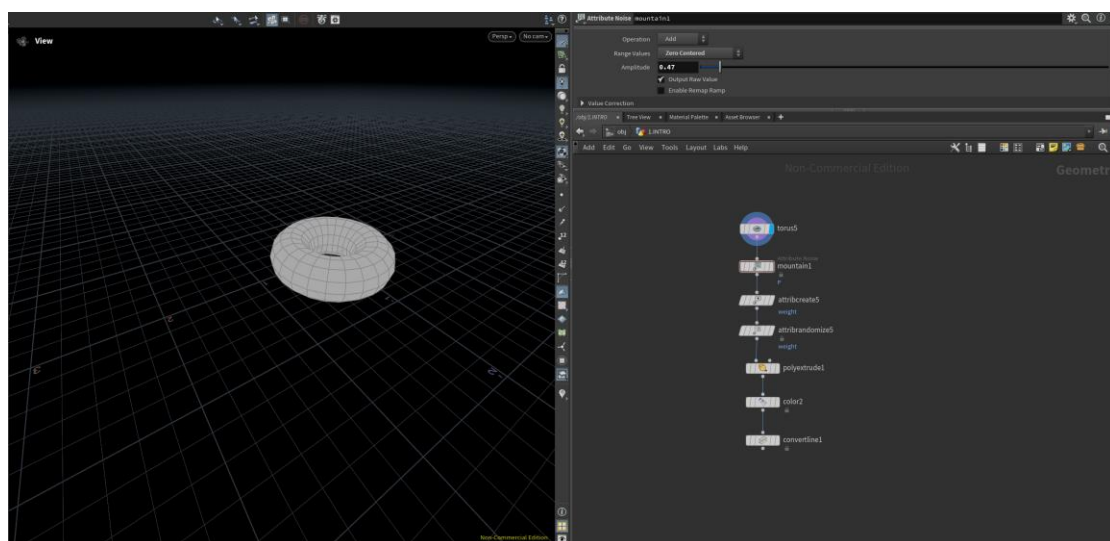
Example 1



Step:

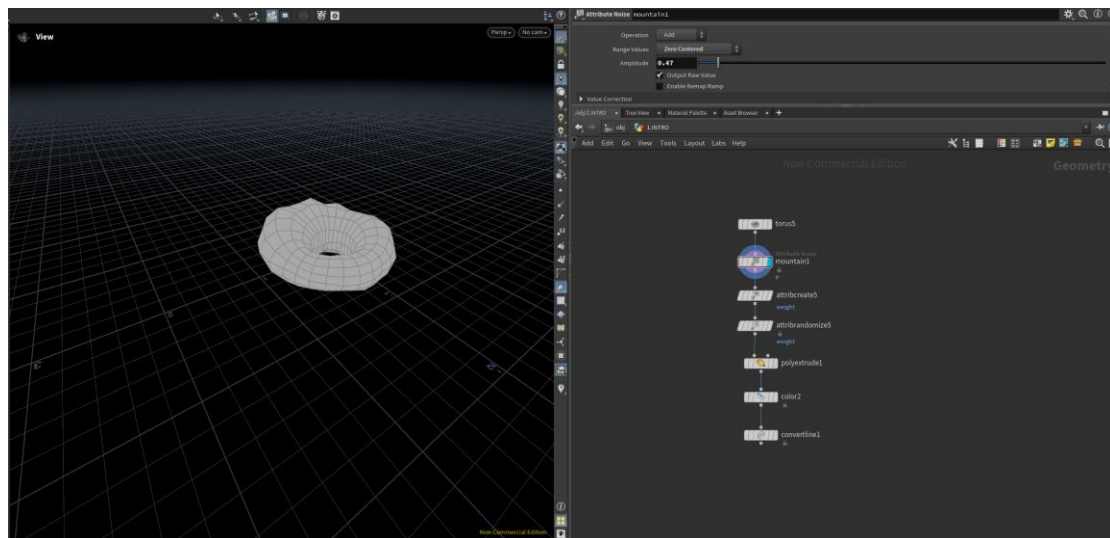
1. Create torus SOP

This SOP is for creating torus geometry.



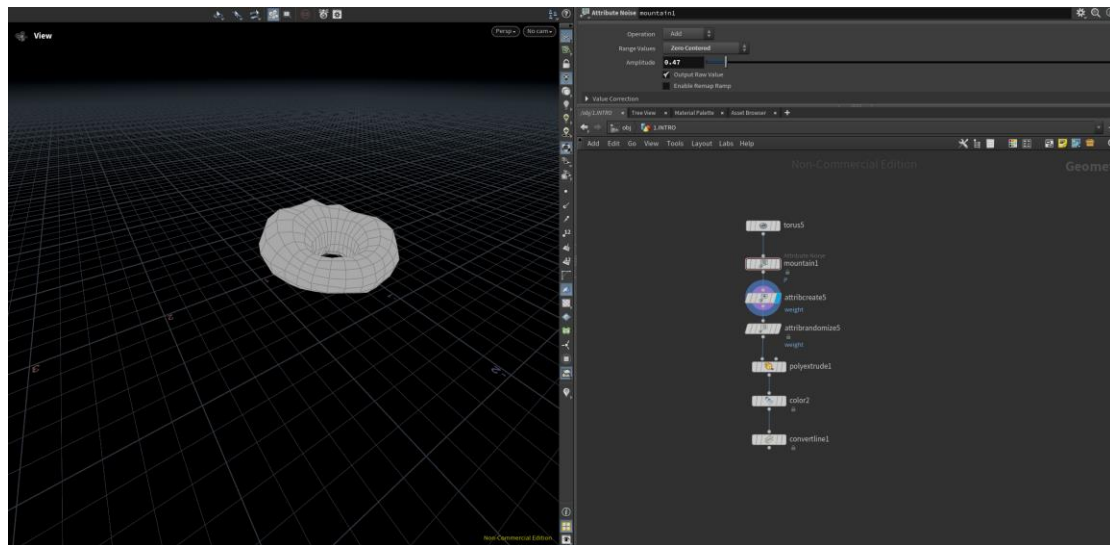
2. Mountain SOP

Mountains SOP is used to generate the geometry for the shape of the mountains. In this step, I create terrain features on the surface of the torus by adjusting the parameters to simulate a more natural rise.



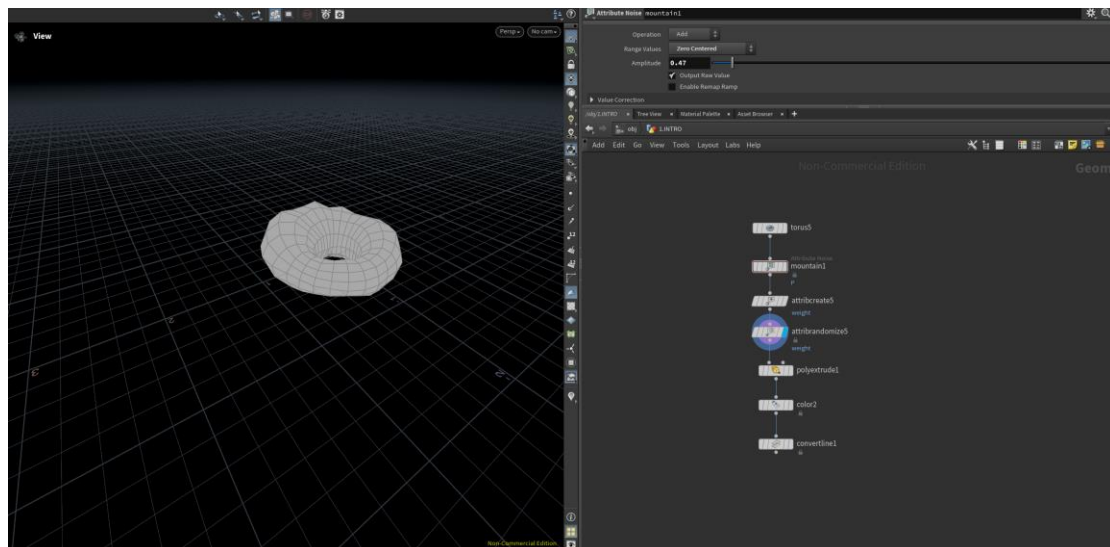
3. Attribute create SOP

This SOP is used to create new attributes on the geometry. Attributes are data associated with points, edges or faces on the geometry, they can contain information such as position, colour, normals, texture coordinates, and so on. I've added the attribute 'weight', which can be used for colouring and deforming downstream nodes.



4. Attribrandomize SOP

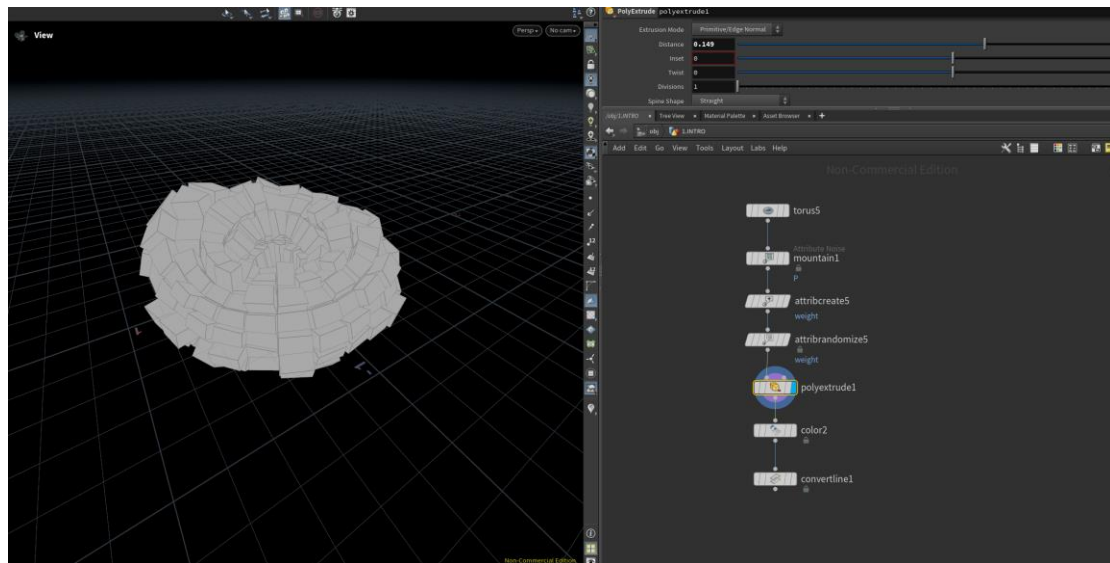
This node, used to randomise the values of the attributes on the geometry. I use the attribute 'weight' to give a new global value, which can be used for colouring and other functions.



5. Polyextrude SOP

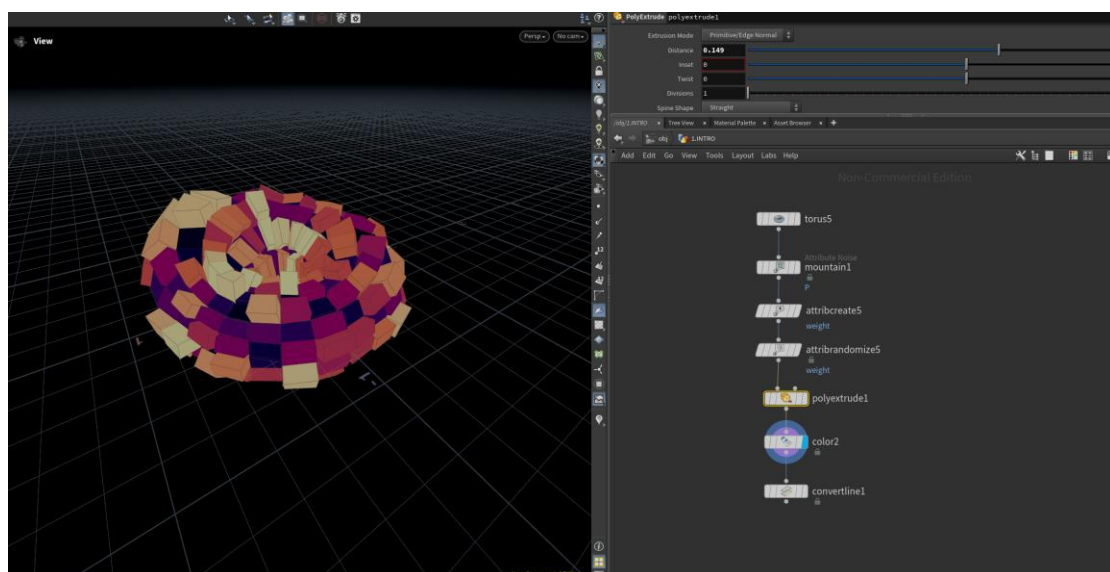
This SOP is used to create new geometry by extruding the surface of the geometry in the normal direction, I adjusted

the 'distance' parameter.



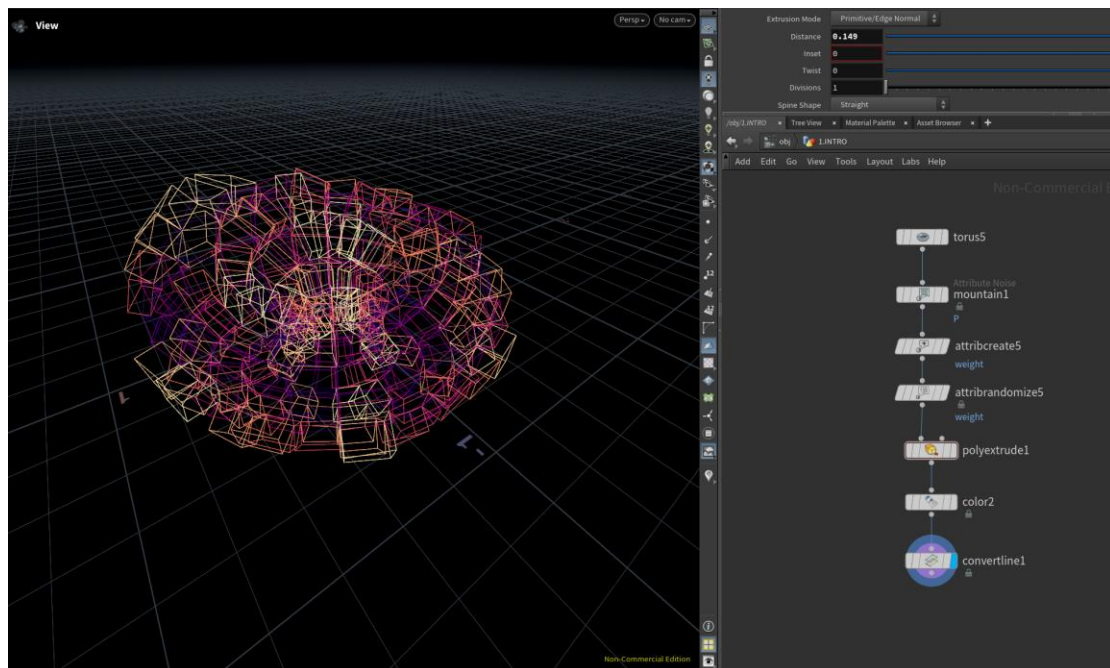
6. Color SOP

This node is used to add, modify or manipulate colour attributes on the geometry. I assign a colour to the geometry through the 'weight' property, give the geometry a colour based on the value of the property, and interpret the visualisation data.



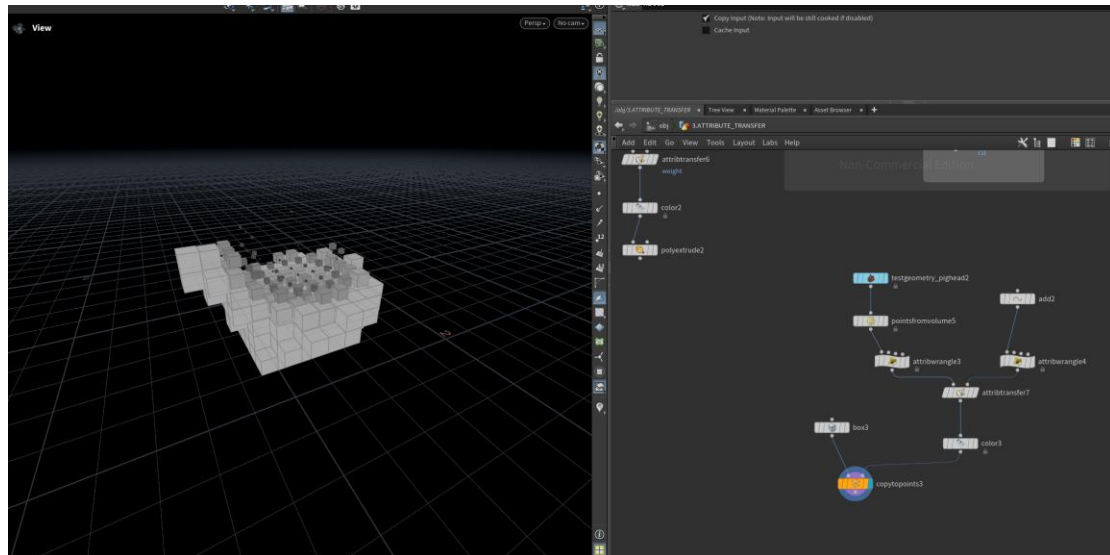
7. Convert line SOP

This node is used to transform line geometry into other types of geometry, here I have transformed the geometry into line segments to see more clearly the internal form of the extruded geometry.

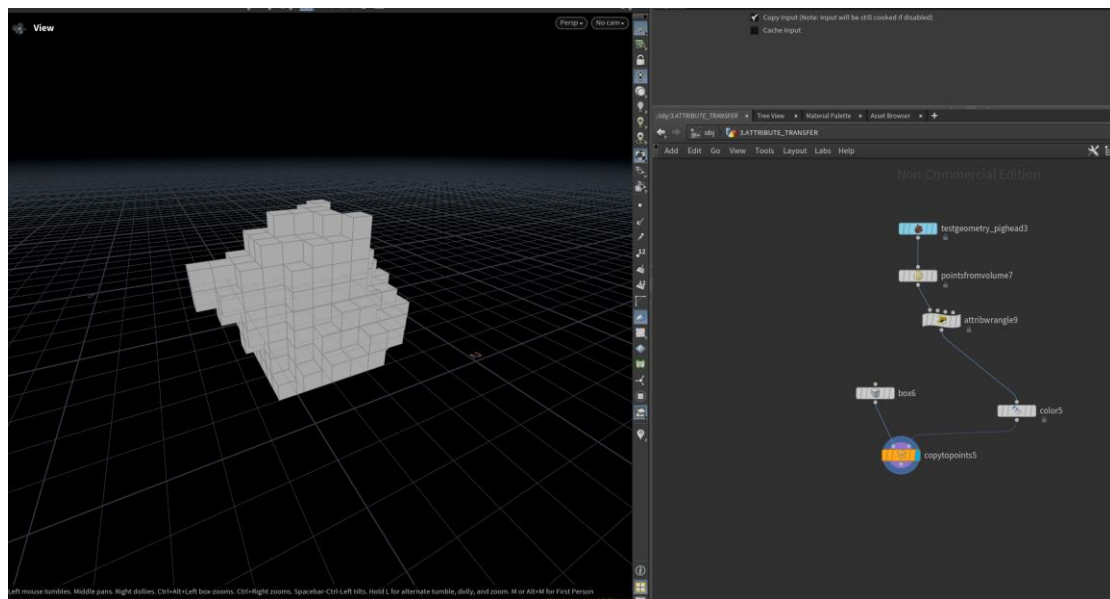


Example 2

initial model



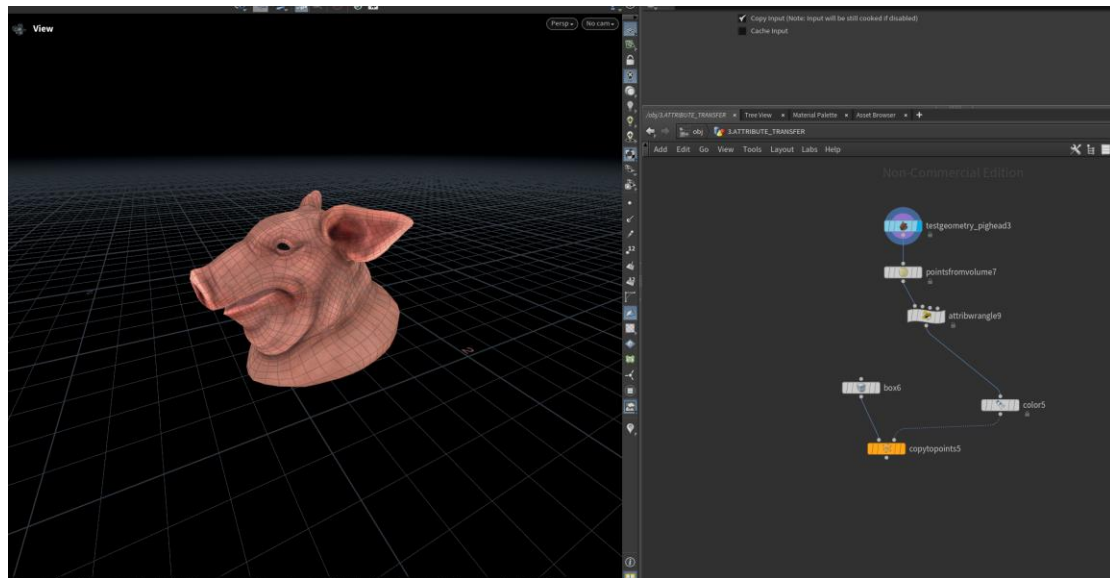
remove 2 nodes



Step:

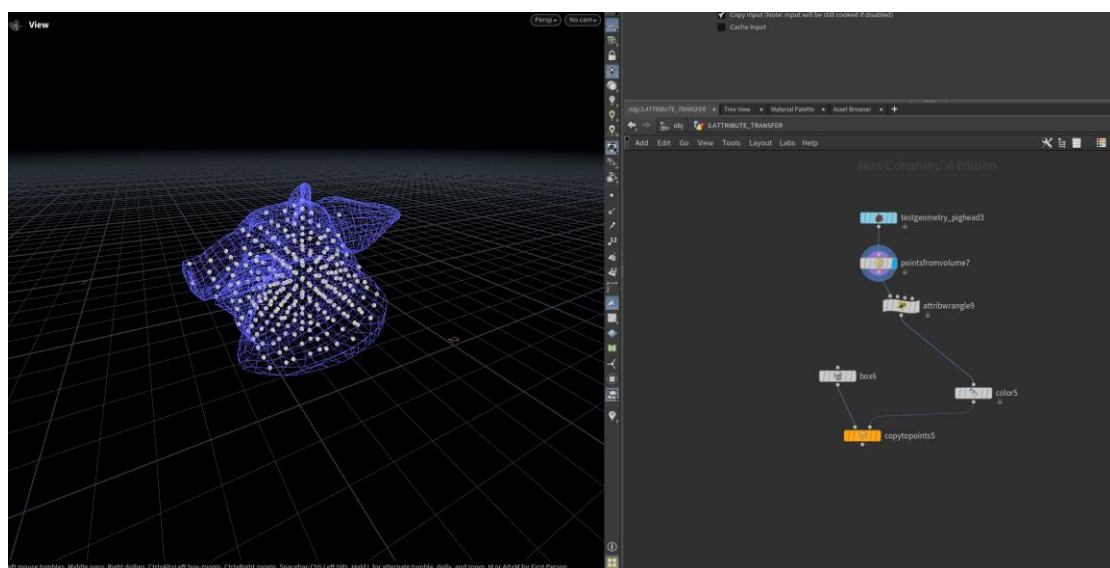
1. Test geometry SOP

This SOP is used to create a base piggyback model



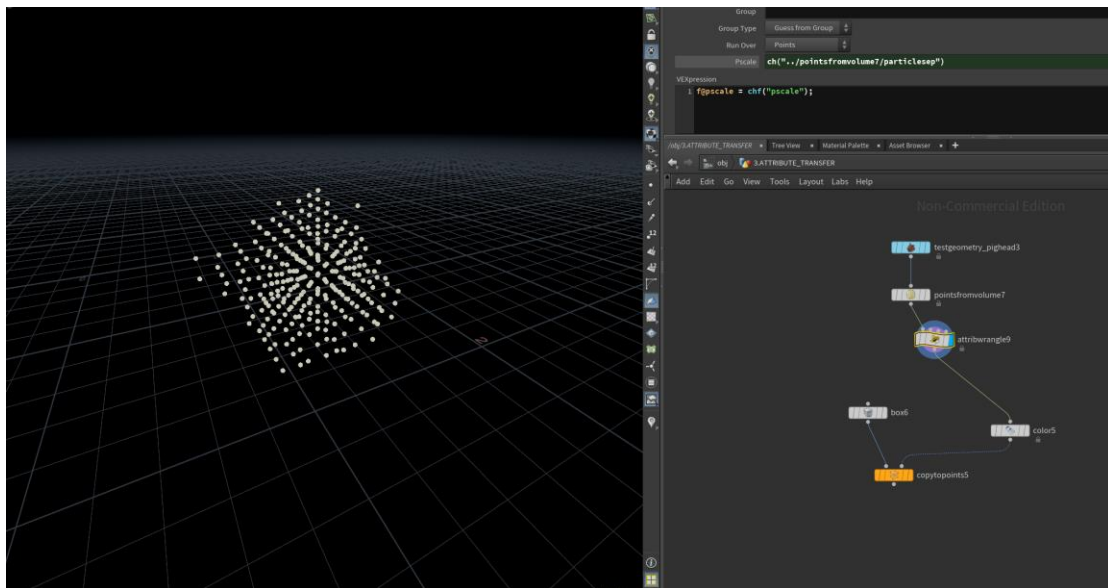
2. Points for volume

This SOP converts voxel data to point cloud data, and I used it to transform pig geometry into point cloud data



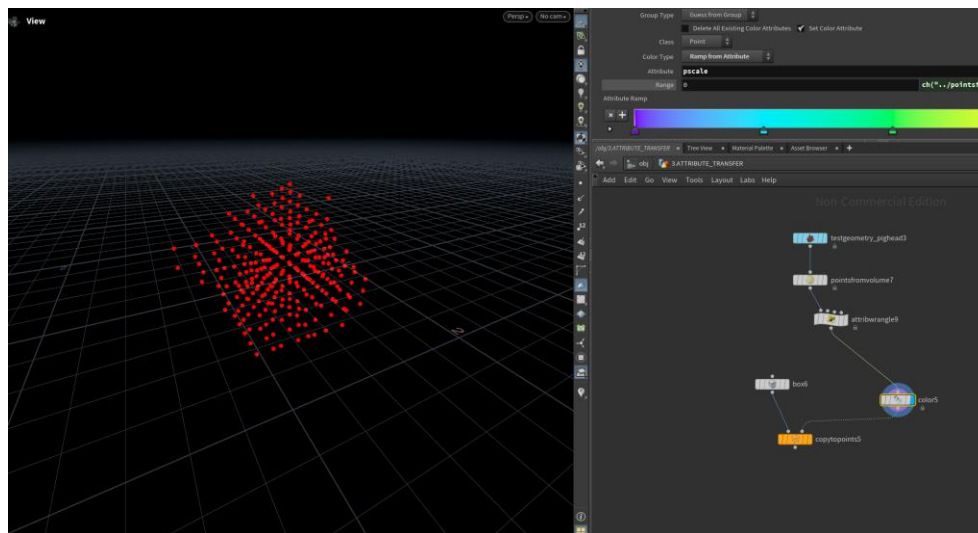
3. Attribute wrangle

This node is used for complex control, computation and manipulation of the properties of the geometry. I used one of the VEX to add the “pscale” attribute



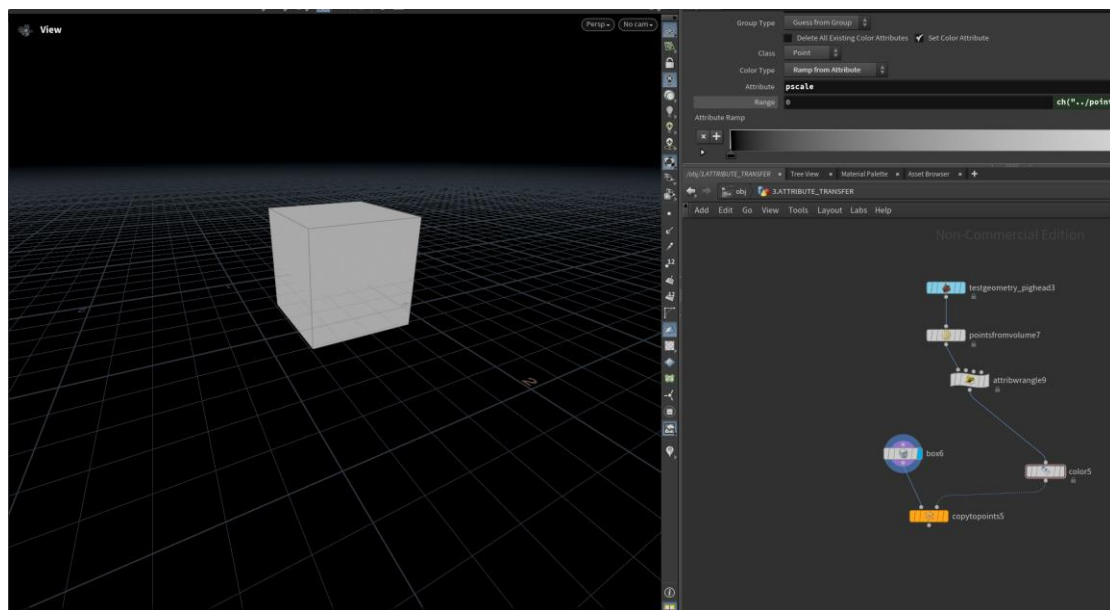
4. Color SOP

This node is used to give color to the geometry, I add the “pscale” attribute to give the color(Temporarily give the red colour to the display function)



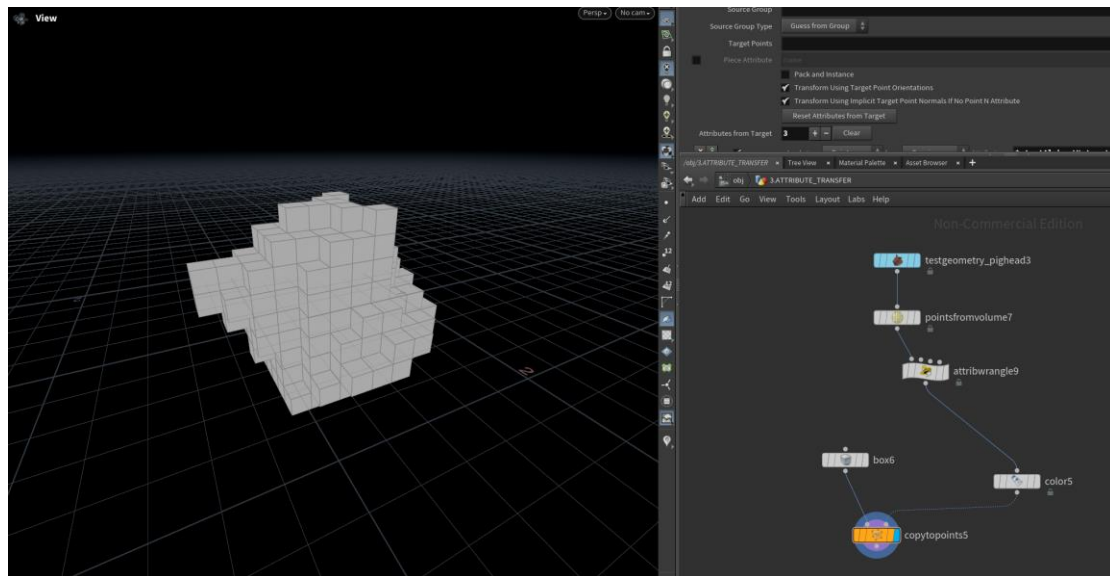
5. Box SOP

This node creates the box geometry



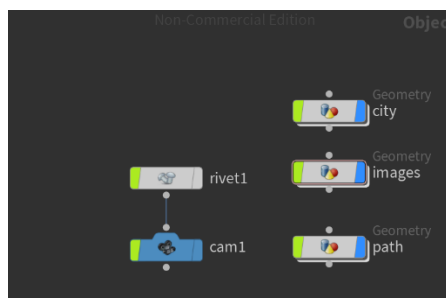
6. Copy to points

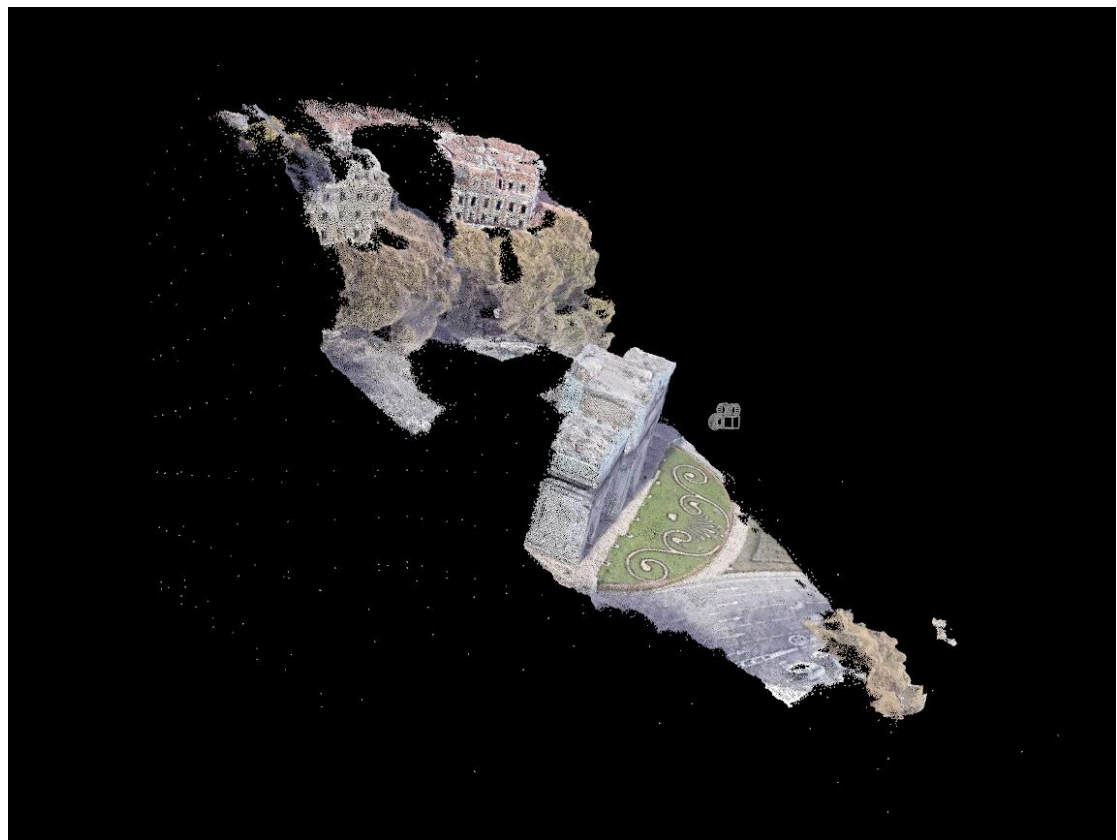
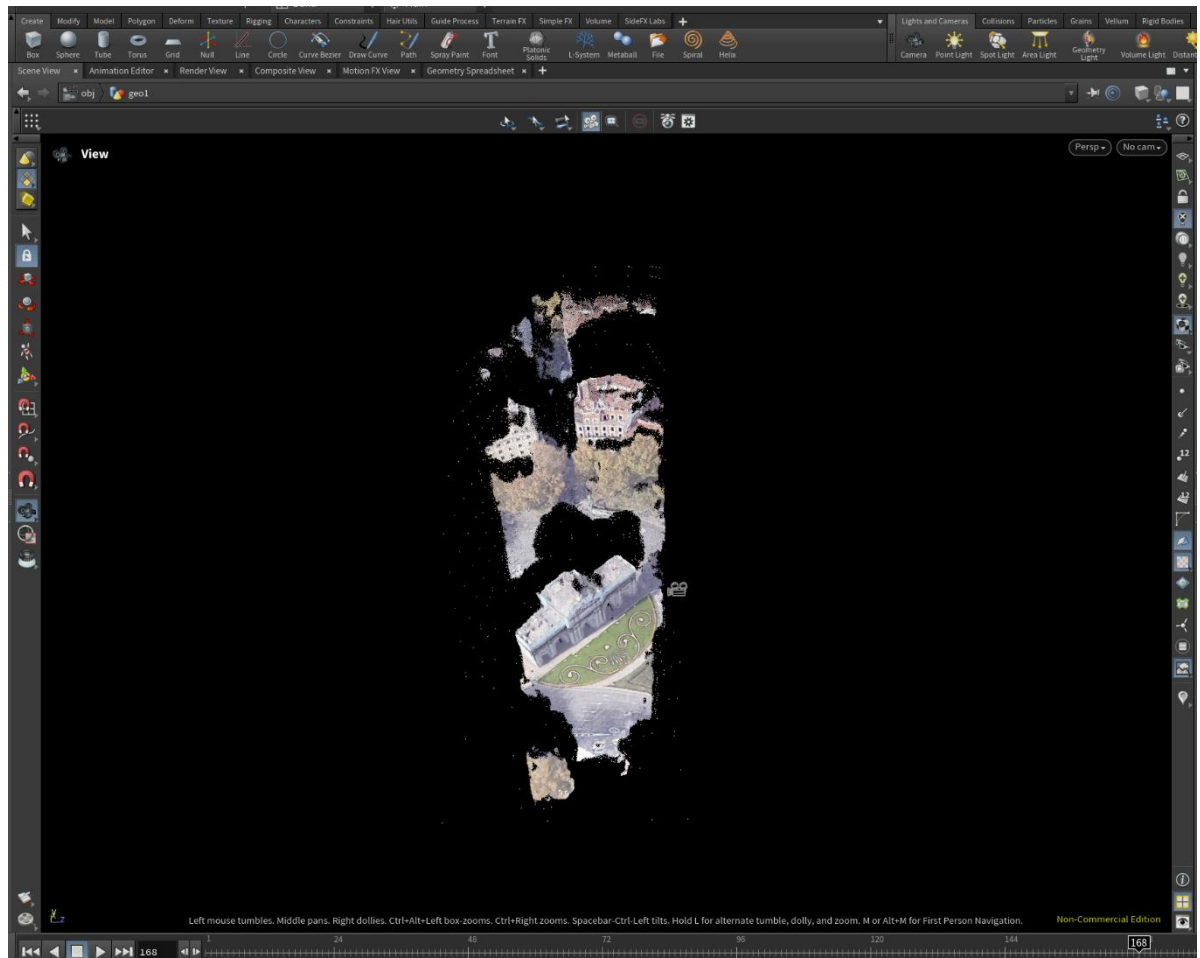
This node copies the object to the point cloud, I have used box geometry as input 1 and colour as input 2 to achieve changing the shape of the point cloud for the piggyback geometry



2 Visualizing GPS and Image

I used my trip to Venice as the theme. Because of some special original, some of the apple mobile phones that have used Chinese phone cards don't have a way to open google timeline, so I mapped out the rough route in google mymap based on the coordinates of the photos I took in Venice before. Then I converted the KML file to GPX file, downloaded the city map of Venice, and input the photos with coordinates. Showing my day in Venice

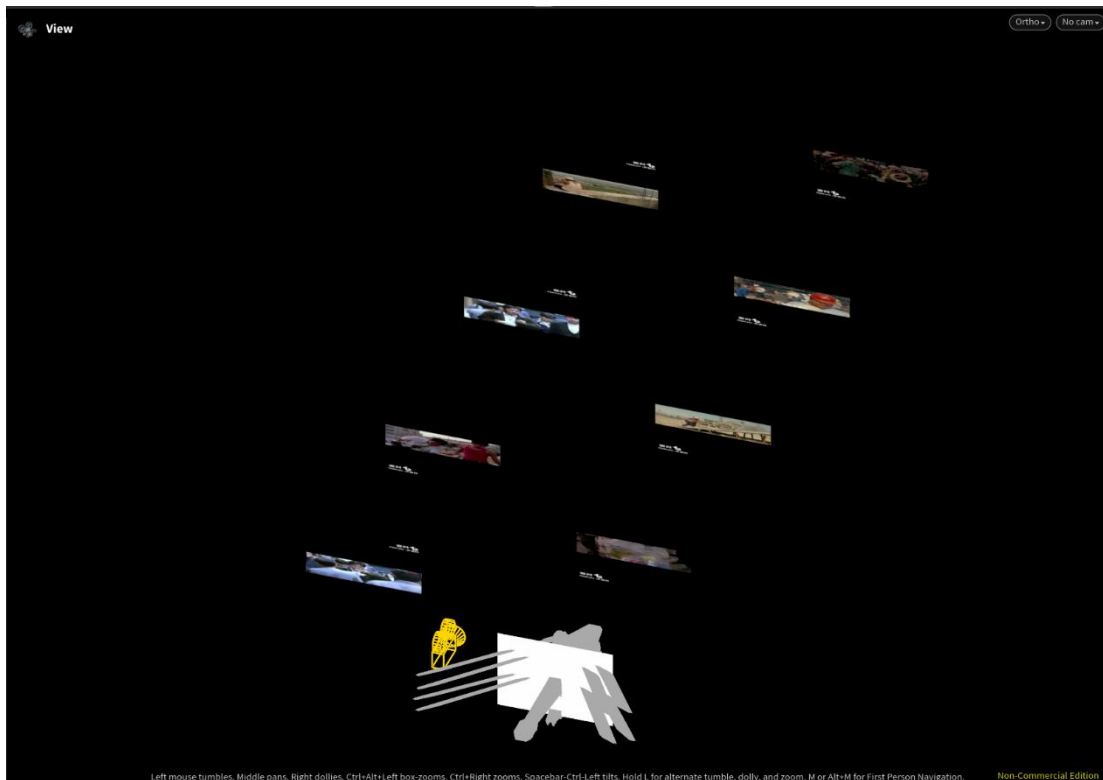
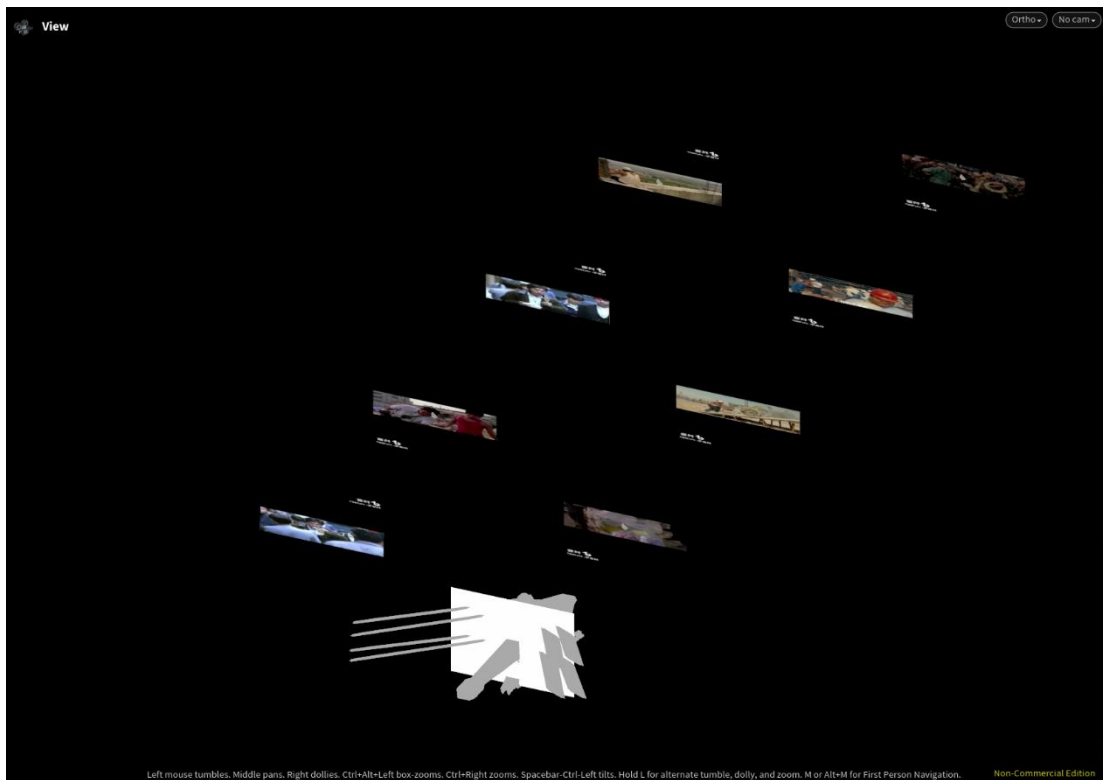




4. Visualizing JSON

I used python to get the video clips and frames, renamed the frames with the same subtitles starting from 1 and put them in a folder, and rewrote the corresponding parts of the json code to enable json using houdini, to generate the film clip projection gallery (Film clip from Tiktok)

```
renconstruct = {
  "folder": [
    {
      "paragraph": "A Day in the Life of the Underclass.",
      "filmID": "tiktok",
      "image_name": "frame",
      "model_path": "model.obj",
      "image_n": 20,
      "time": 100,
      "film_path": "tiktok.mp4"
    },
    {
      "paragraph": "Eat more.",
      "filmID": "tiktok",
      "image_name": "frame",
      "model_path": "",
      "image_n": 15,
      "time": 100,
      "film_path": "tiktok.mp4"
    },
    {
      "paragraph": "You're the boss.",
      "filmID": "tiktok",
      "image_name": "frame",
      "model_path": "",
      "image_n": 18,
      "time": 100,
      "film_path": "tiktok.mp4"
    },
    {
      "paragraph": "Let's go to work.",
      "filmID": "tiktok",
      "image_name": "frame",
      "model_path": "",
      "image_n": 26,
      "time": 100,
      "film_path": "tiktok.mp4"
    }
  ],
}
```

RC11_23155219

PYTHON ASSIGNMENT

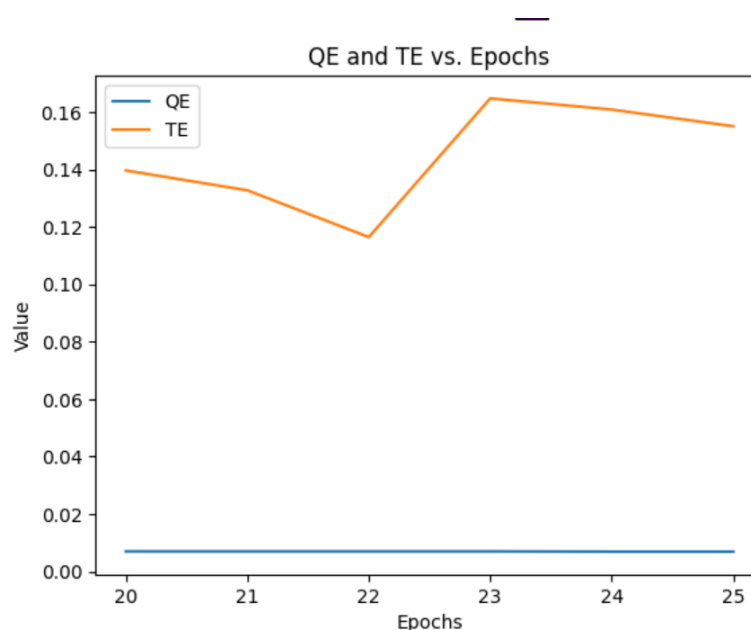
Since I didn't succeed in making a search engine in the end, I can only explain how I used the dataset; the comparison of the two vectorisation methods; and the use of PCA in the SOM again

1. Text_SOM

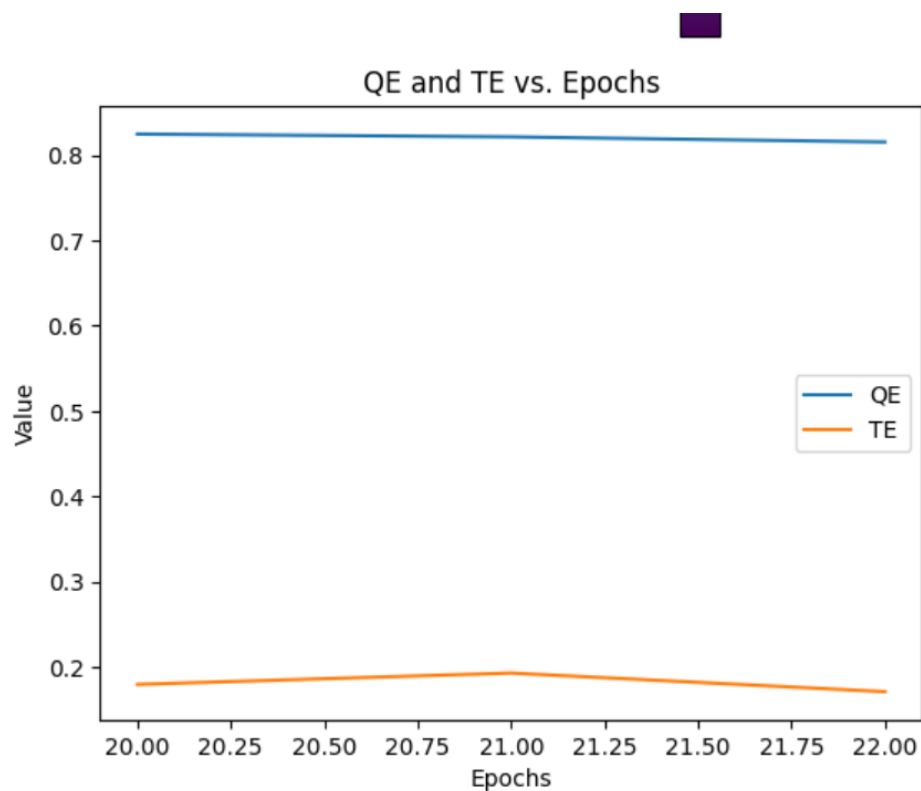
[github-codebook-feijiannughb/Julian_python/final_text_som.py](https://github.com/feijiannughb/Julian_python/final_text_som.py) at main · UD-Skills-2023-24/github-codebook-feijiannughb

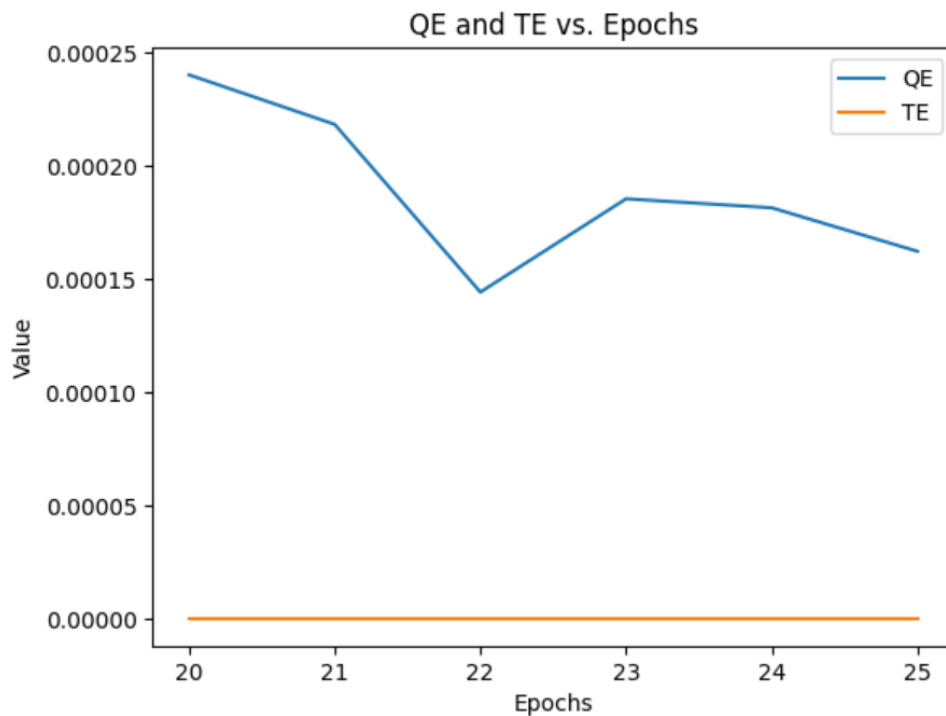
- (1) Text preprocessing call to epub file
- (2) py file in the textmodel
- (3) Training text SOM

The smaller the difference between QE and TE, the better the SOM, according to the line graph, at epochs22, the best SOM



(4) Compare TF-IDF and Doc2Vec, plotting QE and TE line graphs in SOM training, I subsequently chose TF-IDF, because TF-IDF's line graphs are smoother, indicating that its training results are more stable. And TF-IDF focuses on the frequency and rarity of words, which is more suitable for keyword-based characters, I intend to use keyword extraction for both subsequent activation of SOM and searching, and in the case of a large amount of data, the choice of TF-IDF can produce faster run results;.

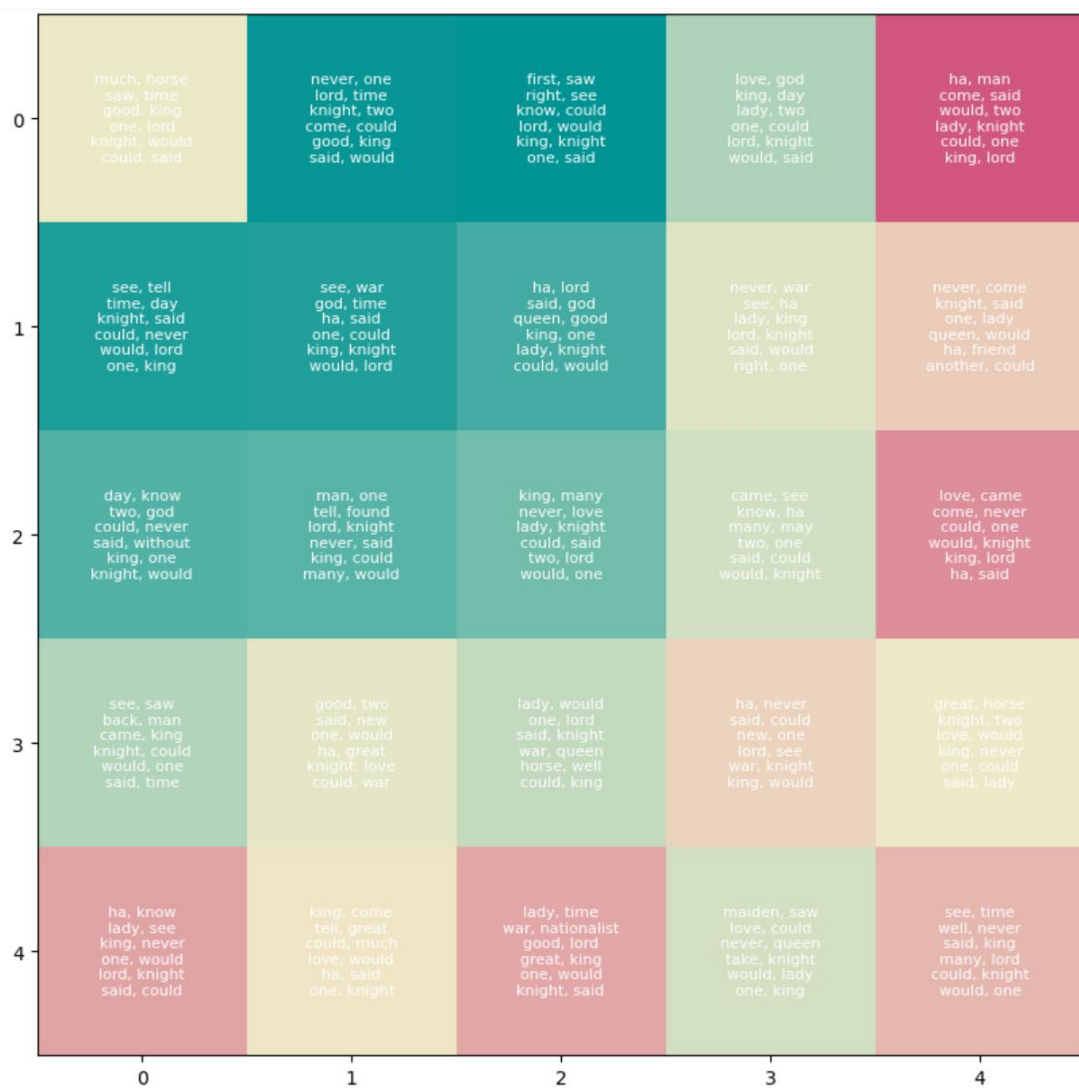




(5) Each cell of the SOM generates RGB colour values based on the three principal components of PCV, and remapping the SOM shows two different results, which may be due to the differences in the distribution of the data in different cells leading to different results in the PCA analysis and the interpretation of the variance of PCA itself, with the primary principal component explaining the largest variance, and the differences in cell colour reflecting the different proportions of variance accounted for by the principal components in different cells.



(6) Mapping text messages to corresponding SOM cells



2. IMAGE SOM

[github-codebook-feijiannughb/Julian_python/final_image_som.py](https://github.com/feijiannughb/Julian_python/final_image_som.py) at main · UD-Skills-2023-24/github-codebook-feijiannughb

[github-codebook-feijiannughb/Julian_python/final_image_fuyutext_som.py](https://github.com/feijiannughb/Julian_python/final_image_fuyutext_som.py) at main · UD-Skills-2023-24/github-codebook-feijiannughb

Similar steps to text som, extract image features, train to generate SOM, map the image to the corresponding SOM node, and then use FUYU model to generate text information of the image content from the corresponding image, and map the text information to the SOM

