Tetrahedron Finder Code Walkthrough

Introduction

The Tetrahedron Volume Calculation Code was initially designed to read a file containing points data and find the smallest tetrahedron formed by selecting four points with a sum of their 'n' values equal to 100. The code aimed to efficiently compute the volume of tetrahedrons and identify the smallest one.

Here is the explanation of key parts of the code.

- **Read points function:** - This function is used to process to read files containing points. It transforms raw data into a format where we can access the content inside the file. Points in the file are represented as tuples of coordinates (x, y, z, n) where value (n) Is an integer value ranging from 0 to 100.

```python
def read_points(file_path):
    points = []
    with open(file_path, 'r') as file:
        for index, line in enumerate(file):
            line = line.strip()
            if line:
                try:
                    x, y, z, n = map(float, line.strip('()\n').split(','))
                    points.append((x, y, z, int(n), index))
                except ValueError as e:
                    print(f"Error processing line: {line}")
                    print(e)
    return points
```

- **Find valid tetrahedron function: -** This function is responsible for finding all the valid combinations of four points where the sum of their ‘n’ attribute is equal to 100.

```python
def find_valid_tetrahedrons(points):
    points.sort(key=lambda p: p[3])  # Sort points based on the 'n' value
    n = len(points)
    valid_tetrahedrons = []


    for i in range(n):
        if i > 0 and points[i][3] == points[i-1][3]:
            continue
        for j in range(i + 1, n):
            if j > i + 1 and points[j][3] == points[j-1][3]:
                continue
            k, l = j + 1, n - 1
            while k < l:
                total = points[i][3] + points[j][3] + points[k][3] + points[l][3]
                if total == 100:
                    valid_tetrahedrons.append((points[i], points[j], points[k], points[l]))
                    # print("")
                    print(f" Found valid combination: {points[i][:4]}, {points[j][:4]}, {points[k][:4]}, {points[l][:4]} \n indexes for above valid combination : {points[i][4]},
                    while k < l and points[k][3] == points[k+1][3]:
                        k += 1
                    while k < l and points[l][3] == points[l-1][3]:
                        l -= 1
                    k += 1
                    l -= 1
                elif total < 100:
                    k += 1
                else:
                    l -= 1
    print(f"\n Total valid tetrahedrons found: {len(valid_tetrahedrons)}")
    # print(f"valid tetrahedrons found: {(valid_tetrahedrons)}")
    return valid_tetrahedrons
```

The initial implementation used a brute-force approach, using itertools generating all possible combinations of points and checking their sums, leading to exponential time complexity and impractical performance for large datasets as shown in the below screenshot.

Previous code:

```
def find_valid_tetrahedrons(points):
    valid_tetrahedrons = []
    for combination in itertools.combinations(range(len(points)), 4):
        if sum(points[i][3] for i in combination) == 100:
            valid_tetrahedrons.append(combination)
    return valid_tetrahedrons
```

The optimized implementation approach reduce redundancy and enhance performance using two-pointer technique to find valid combinations with time complexity of O(n^3)

- **Volume of tetrahedron function: -** Calculates the volume of the tetrahedron by taking tuple of the four points as input and calculates the volume based on their coordinates.

```
def volume_of_tetrahedron(combination):

    p1, p2, p3, p4 = combination
    AB = (p2[0] - p1[0], p2[1] - p1[1], p2[2] - p1[2])
    AC = (p3[0] - p1[0], p3[1] - p1[1], p3[2] - p1[2])
    AD = (p4[0] - p1[0], p4[1] - p1[1], p4[2] - p1[2])

    cross_product_x = AB[1] * AC[2] - AB[2] * AC[1]
    cross_product_y = AB[2] * AC[0] - AB[0] * AC[2]
    cross_product_z = AB[0] * AC[1] - AB[1] * AC[0]

    scalar_triple_product = (
        AD[0] * cross_product_x +
        AD[1] * cross_product_y +
        AD[2] * cross_product_z
    )

    volume = abs(scalar_triple_product) / 6.0

    return volume, combination
```

- **Find smallest tetrahedron function:** It finds smallest tetrahedron among the valid combinations. Utilized multiprocessing to parallelize the computation. It divides the workload among multiple processes to improve performance.

```
def find_smallest_tetrahedron( valid_tetrahedrons, count=4):
    with Pool(processes=multiprocessing.cpu_count()) as pool:
        results = pool.map(volume_of_tetrahedron, [combination for combination in valid_tetrahedrons])

    results.sort(key=lambda x: x[0])
    return results[:count]
    # min_volume, min_combination = min(results, key=lambda x: x[0])
    # return min_combination
```

Initially, we used a worker function to handle coordinates and their indices separately, which added complexity. The find_smallest_tetrahedron_worker function was used to manage this.

Previous Code

```
# def find_smallest_tetrahedron_worker(args):
#     combination = args
#     p1, p2, p3, p4 = combination
#     volume = volume_of_tetrahedron(p1, p2, p3, p4)
#     return volume, combination

def find_smallest_tetrahedron( valid_tetrahedrons, count=4):
    with Pool(processes=multiprocessing.cpu_count()) as pool:
        results = pool.map(volume_of_tetrahedron, [combination for combination in valid_tetrahedrons])

    results.sort(key=lambda x: x[0])
    return results[:count]
    # min_volume, min_combination = min(results, key=lambda x: x[0])
    # return min_combination
```

I have improved the code by adding an index field to the tuples, which preserved the original indices and allowed us to remove the unnecessary worker function. This streamlined the calculation process.

Output:

Output for points_small.txt file

```
PS E:\Urva Surti\Coding Task> python Tetrahedron_test.py

 Number of points in file: 100

 Found valid combination: (459.59, 481.89, 392.24, 0), (167.25, 157.89, 25.73, 0), (191.55, 158.11, 476.93, 0), (161.
27, 323.97, 130.23, 100)
 indexes for above valid combination : 9, 34, 57, 94

 Found valid combination: (459.59, 481.89, 392.24, 0), (167.25, 157.89, 25.73, 0), (76.79, 467.27, 423.67, 10), (463.
34, 196.1, 392.33, 90)
 indexes for above valid combination : 9, 34, 5, 74

 Found valid combination: (459.59, 481.89, 392.24, 0), (167.25, 157.89, 25.73, 0), (274.35, 408.58, 448.18, 20), (81.
63, 93.18, 457.45, 80)
 indexes for above valid combination : 9, 34, 0, 98

 Found valid combination: (459.59, 481.89, 392.24, 0), (167.25, 157.89, 25.73, 0), (468.95, 468.41, 167.5, 30), (109.
6, 88.61, 25.47, 70)
 indexes for above valid combination : 9, 34, 4, 89

 Found valid combination: (459.59, 481.89, 392.24, 0), (167.25, 157.89, 25.73, 0), (28.63, 95.35, 492.75, 40), (195.5
1, 312.52, 417.82, 60)
 indexes for above valid combination : 9, 34, 8, 95

 Found valid combination: (459.59, 481.89, 392.24, 0), (167.25, 157.89, 25.73, 0), (364.86, 142.27, 490.77, 50), (367
.61, 266.31, 70.12, 50)
 indexes for above valid combination : 9, 34, 1, 80

 Found valid combination: (459.59, 481.89, 392.24, 0), (76.79, 467.27, 423.67, 10), (64.99, 231.57, 22.93, 10), (81.6
3, 93.18, 457.45, 80)
 indexes for above valid combination : 9, 5, 25, 98

 Found valid combination: (459.59, 481.89, 392.24, 0), (76.79, 467.27, 423.67, 10), (274.35, 408.58, 448.18, 20), (10
9.6, 88.61, 25.47, 70)
 indexes for above valid combination : 9, 5, 0, 89

 Found valid combination: (459.59, 481.89, 392.24, 0), (76.79, 467.27, 423.67, 10), (468.95, 468.41, 167.5, 30), (195
.51, 312.52, 417.82, 60)
 indexes for above valid combination : 9, 5, 4, 95
```

```
 Found valid combination: (76.79, 467.27, 423.67, 10), (468.95, 468.41, 167.5, 30), (107.91, 28.35, 84.87, 30), (344.
7, 14.94, 364.25, 30)
 indexes for above valid combination : 5, 4, 14, 92

 Found valid combination: (274.35, 408.58, 448.18, 20), (493.87, 361.72, 343.12, 20), (154.68, 418.89, 383.69, 20), (
482.17, 248.2, 357.77, 40)
 indexes for above valid combination : 0, 3, 26, 84

 Found valid combination: (274.35, 408.58, 448.18, 20), (493.87, 361.72, 343.12, 20), (468.95, 468.41, 167.5, 30), (3
44.7, 14.94, 364.25, 30)
 indexes for above valid combination : 0, 3, 4, 92


 Total valid tetrahedrons found: 23

 Starting to find the smallest tetrahedron...

Smallest Tetrahedron 1 for file: [0, 8, 9, 84] with volume: 37597.5256616668

Smallest Tetrahedron 2 for file: [4, 9, 34, 89] with volume: 145763.12774066566

Smallest Tetrahedron 3 for file: [0, 3, 9, 95] with volume: 146027.64878000008

Smallest Tetrahedron 4 for file: [5, 25, 31, 89] with volume: 222302.80649166647
```

Output for points_large.txt file.

```
Found valid combination: (199.88, 37.94, 169.4, 24), (13.71, 50.69, 374.7, 24), (191.2, 382.29, 238.5, 26), (407.76, 380.8, 36.35, 26)
indexes for above valid combination : 54, 79, 334, 1473

Found valid combination: (199.88, 37.94, 169.4, 24), (210.14, 189.56, 119.06, 25), (340.72, 348.83, 419.03, 25), (407.76, 380.8, 36.35, 26)
indexes for above valid combination : 54, 59, 93, 1473

Found valid combination: (210.14, 189.56, 119.06, 25), (340.72, 348.83, 419.03, 25), (409.91, 47.18, 223.91, 25), (188.9, 417.55, 393.05, 25)
indexes for above valid combination : 59, 93, 221, 1289


Total valid tetrahedrons found: 8037

Starting to find the smallest tetrahedron...

Smallest Tetrahedron 1 for file: [1, 28, 51, 1448] with volume: 13.39574533328414

Smallest Tetrahedron 2 for file: [1, 4, 149, 1329] with volume: 330.2920669999876

Smallest Tetrahedron 3 for file: [14, 73, 112, 1492] with volume: 368.7653936667678

Smallest Tetrahedron 4 for file: [15, 66, 108, 1369] with volume: 1002.3228285000272
```

Results: The optimized code demonstrated a significant improvement in performance compared to the initial implementation. For both small and large datasets, the optimized solution provided faster computation times, with negligible impact on accuracy.

Conclusion: Through strategic optimization techniques, including preprocessing, parallel processing, and code refactoring, we successfully improved the efficiency of tetrahedron volume calculation. The optimized code exhibited reduced time complexity, making it suitable for processing large datasets efficiently.