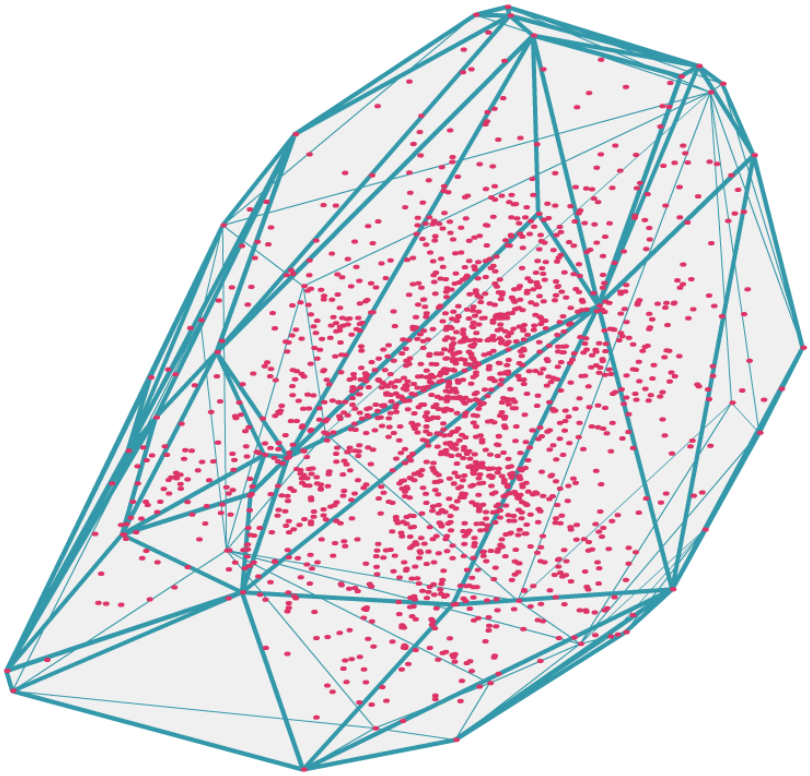
ALGORITHMS AND OPERATING SYSTEMS

PROJECT REPORT

3D CONVEX HULL AND ITS APPLICATIONS



Mohith Pavan Kumar-2018102016

Tejaswini Anuhya Suma-2018102018

Naveen Kumar-2019102037

Siva Durga-2019102038

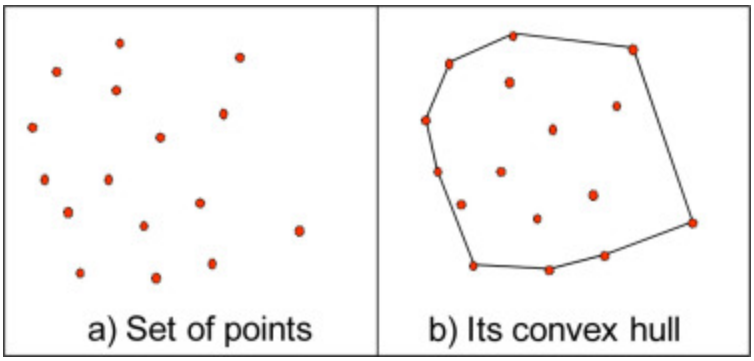
INTRODUCTION:-

A Region of surface bound by vertices , edges and faces is called a Polyhedron. For any 2 vertices p,q in a polyhedron, the line segment joining p,q will lie inside the polyhedron if it is convex. A convex polyhedron can also be called a polytop.

The convex hull of a set of points is the smallest convex set that contains the points. All faces of Convex Hull are extreme (and all extreme edges of Polyhedron are on the Convex Hull). We represent a convex hull with a set of facets and a set of adjacency lists giving the neighbors and vertices for each facet. The boundary elements of a facet are called ridges. Each ridge signifies the adjacency of two facets.

In 3D, the ridge is an edge and the facet is the face of the simplex. This project attempts to implement 3 Dimensional Convex Hull for the given 3D dataset points, using the QuickHull algorithm. In this project, we also investigate and compare the time and number of faces formed with brute force and random incremental algorithm.

2D CONVEX HULL



Ref - ScienceDirect - Convex hull 2D

For two dimensional points, it may suffice more simply to list the points that are vertices, in their cyclic order around the hull.

Naive solution to implement would be:(complexity being polynomial -here O(n^3))

Let P be the set of N points in the plane. For now let us assume that no three points from P are collinear or lie on the same line.

- Two points from the set P define an edge of the convex hull, if and only if all the other points from P lie on the same side from the line through this segment.
- Let E denote a set that will store the edges of the convex hull. First, we initialize E with an implicit. Subsequently, for each ordered pair, AB, of points from P, we check whether all the other points from P lie on the left of the direct line through the points A and B.
- If this is the case, we place the segment AB in the set E, and as a final set, we generate from the set E a list of vertices of the convex hull of our point set P.

The algorithm may be easily modified to deal with collinearity, including the choice whether it should report only vertices of the convex hull or all points that lie on the convex hull.

The algorithm spends linear time per each pair of points. The total number of pairs of points is quadratic. Hence, we conclude that the time complexity of our algorithm is cubic in the number of points in the set P. There are various algorithms like Gift Wrapping, Graham scan ,Quick hull, incremental and divide and conquer approach better than the original algorithm with best complexity of O(n log n)

3D CONVEX HULL :-

For 3D convex hull computation, it is assumed that the input points are in general position(not satisfying more affine dependencies than absolutely necessary-like no 2 vertices at same point, no 3 vertices on the same line etc., depending upon the dimension) so that their convex hull is a simplicial.

Naive Way is to compute a convex hull for given 3D points is as follows:-

- Consider every triplet of three points ($\vec{a}$,$\vec{b}$,$\vec{c}$), and check if they create a triangular face on the convex hull. In order for it to be a face, the remaining points must "see" the same

side of the triangle. In other words, if we consider the plane containing this triangle, the remaining points should lie on the same side of the plane.

- We can first take a vector $\vec{q} = (\vec{b} - \vec{a}) \times (\vec{c} - \vec{a})$ orthogonal to the plane and the sign of $(\vec{p} - \vec{a}) \cdot \vec{q}$ tells us the side of the plane.
- For each triplet, we perform this check with all points, so the total time complexity is O(n^4).

As an improvement on the naive Algorithm, the convex hull can be implemented using an extension of incremental algorithm and also QuickHull Algorithm for 3D. Quickhull algorithm is preferred over incremental algorithm because the worst-case complexity of an incremental algorithm is O(n^2) whereas that of quickhull is O(n log n).

3D QUICK HULL ALGORITHM:

The QuickHull algorithm computes the convex hull of a set of n-dimensional points using a divide and conquer approach similar to that of quicksort. N-dimensional QuickHull is an extension of planar or 2D QuickHull Algorithm. A d-dimensional convex hull is represented by faces and d - 1 facets. There are 2 advantages of using the QuickHull Algorithm. They are

1) Quickhull reuses the memory occupied by old facets and hence is optimized in terms of memory.
2) Quick hull algorithm is output-sensitive and it is important for convex hull algorithms because the output size can be much smaller than the worst-case input size.

Algorithm for the 3D convex hull is as follows:-

Initial phase

Step 1:- Calculate the maximum and minimum points on all the axes. Choose the 2 furthest points and join a line.

Step 2:- Find the point with maximum distance from above line and make a triangle. Similarly, find the most distant point from this plane and make a tetrahedron which will serve as the base unit of the 3D convex hull.

Step 3:- Take the dot product of the clockwise normal of the plane(any of the 4 faces) to the line joining any vertex of the plane and the point which is the distance of the point from the plane.

Step 4:- If the distance is positive, add them to a to_do list(consisting of outside points for each plane) and if not remove them from the set of points(since they are internal points and need not be processed therefore can be eliminated).

Final phase

Step 5:- Continue the program till there is a face of the polytope who has a non-zero to_do list which is computed using step 3,4.

Step 6:- Find the horizon of this point(the vertices to which this point will connect) where this point is the furthest for that set of vertices/horizon.

Step 7:- Implement DFS from the face to which the point was a to_do list point. The base condition to terminate DFS will depend on whether the point in consideration is in the to_do list of the face or not.

Step 8:- Re-assign other vertices of the to_do list to the new faces of the cone of facets that is made. Do this until no points are left and then remove the old faces where this point is at positive distance and are the neighbours which are iterated in a recursive way from the list.

This will give a final set of edges which are the horizon set or set of vertices which make the convex hull. Quickhull is faster than randomized incremental algorithms because it processes fewer points. They are comparable because if the selection step for point(Note that Quickhull always selects the furthest point) is changed to random point from furthest point , it is a randomized incremental algorithm. **(source code submitted)**

Reasoning for Correctness of the Algorithm:-

Theorem:- d- Dimensional QuickHull algorithm computes/produces a convex hull in d Dimension.

Proof/ Reason:- Quickhull starts with the convex hull of d + 1 points(minimum). After initialization, Quickhull assigns each unprocessed point to an outside set. By definition, the corresponding facet is visible from the point since it is part of the outside set.

When Quickhull creates a cone of new facets, it builds new outside sets from the outside sets of the visible facets. This process is called partitioning. Quickhull partitions points into outside sets and picks furthest points for processing. If a point is above multiple new facets, one of the new facets is selected. If it is below all of the new facets, the point is inside the convex hull and can be discarded.

Quickhull merges the facet that minimizes the maximum distance of a vertex to the neighbor and it retains the outside points for every facet and discards the internal points. Therefore in the end, the set will have only the vertices of the convex hull enclosing all the points. Hence proved.

Reasoning for Complexity:-

Theorem:- If there are n input points and number of processed points is r, then the worst case complexity of a quick hull algorithm is O(n log r) for 3 or less than 3 dimensions.

Reason 1(geometrical):- Let's assume all points are processed in the worst case, therefore r=n.

There are two costs to Quickhull: adding a point to the hull and partitioning.

The dominant cost for adding a point is O(d^3) work to create hyperplanes. The dominant cost for partitioning is O(d) work to compute distances. The total cost for adding points is

proportional to the total number of new facets created and the cost of partitioning one point in one iteration is proportional to the number of new facets.

Using klee conjecture(maximum number of facets of r vertices is $O(r^{\lfloor d/2 \rfloor / \lfloor d/2 \rfloor !})$), the total cost is $O(d^3 * n * \Sigma 1/j)$, j=1 to r which gives O(n log r) or rather O(n log n) using the assumption r=n.

Reason 2(using code):- Using the Euler formula for polyhedra(Euler's formula: V - E + F = 2), we can say that edges and faces are linearly related to the number of vertices, i.e O(n). So for loop over edges or faces is of complexity O(n).

For 3D convex hull computation, while using the QuickHull algorithm, we use divide and conquer approach and divide the space into 4 parts(4 to_do lists for each face of the tetrahedron) and use recursion which costs 'n'.

So the cost equation would be T(n)=4T(n/4)+n.

Using Masters Theorem($f(n)=\Theta(n^{\log_b a})$), the complexity is O(n log n).

Pseudocode:-

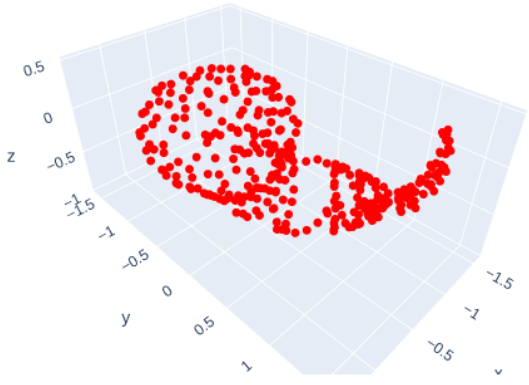
```
Create a tetrahedron simplex
for each face F:
        for each unassigned point P:
                if dist(P) >0 :
                        Add to to_do list
for each face F with a non-empty to_do list:
        select the furthest point Q of F's to_do list
        initialize the visible set V to F
        for all unvisited neighbors N of faces in V:
                if p is above N:
                        add N to V
        the boundary of V is the set of horizon edges H
```

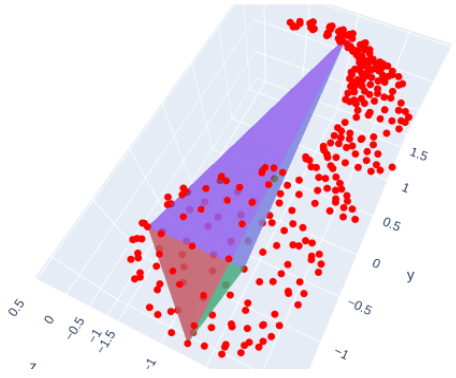
```
for each edge e in H:
        create a new face from e and p
        link the new face to its neighbors
for each new face F:
        for each unassigned point q in a to_do list of a face in V:
                if q is above F:
                        Add q to F's to_do list
delete the facets in V
```

RESULTS:-

In this project, we implemented the convex hull problem using the quick hull algorithm for 3D points, which works efficiently in both time and space complexities compared to computing the convex hull using the random incremental algorithm. The implemented codes and outputs for some of the test cases are attached, and the instructions for executing the codes are provided at the end of this report.

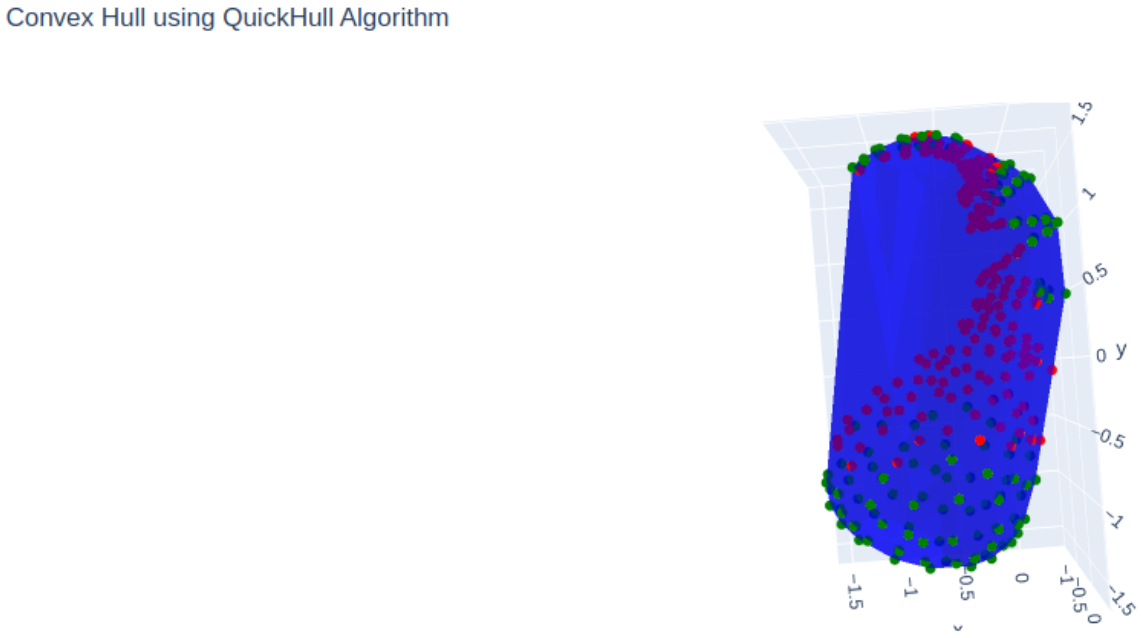
To test the implementation of the quick hull algorithm, we considered a set of 326 3D points; the overall shape of the input point set is similar to bottle gourd, as shown in the below left side figure.



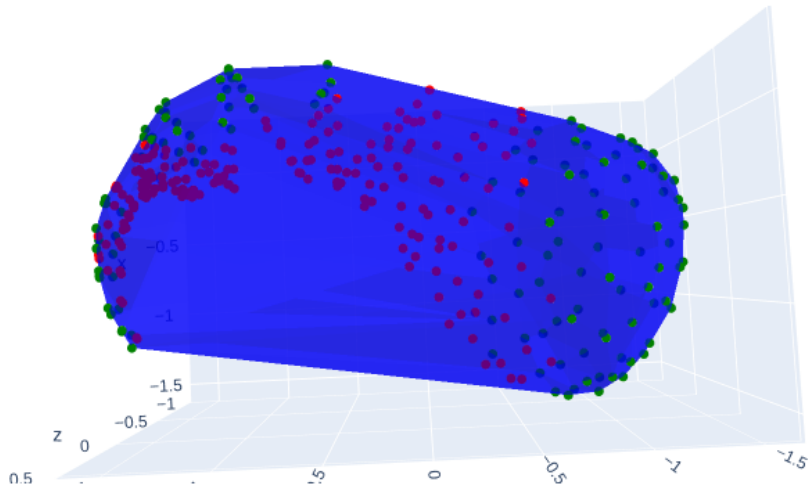


We passed this set as an input to our code and obtained a base simplex shown in the right side figure. This base simplex(in 3D input case) is a tetrahedron received due to the second step of the algorithm described above.

After the final execution, it gave 264 faces and 134 vertices as the faces and vertices of the convex hull. To verify the results of the quick hull algorithm, we found the convex hull of the above set using the scikit library. It also resulted in the same number of faces and vertices. The plots of the convex hull for both cases are as shown in the below figures [To get a better view, look at the plots in the quickhull.ipynb] notebook. We can notice that both the hulls are the same. In the figure, the red points are the interior points and the green are the boundary points of the convex hull.



Convex Hull [Implemented from inbuilt library]



To verify further, we considered various 3D input files and found the convex hulls using both methods and the results are in ‘test outputs’ and code ‘src’ folder. We further investigated the time and space complexity of the quick hull, compared it with a random incremental, brute force algorithm by generating random points in 3D space and analysed it.

Analysis :

The analysis is performed for 3 algorithms, brute force algorithm, quick hull algorithm and random incrementation algorithm to compute convex hull for 3-dimensional input. Time taken by each algorithm and number of faces (verified out of nc3 faces) is noted.

From the below graphs and tables, the following observations are drawn

- The number of faces covered by random incremental algo is more than the quick hull algorithm(Fig-2)(Reason:-random incremental algo chooses a random point instead of finding horizon so it unnecessarily checks the internal planes where it's not required.)
- Time taken by brute force is greater than the remaining 2 algorithms because it checks all the nc3 planes.(Fig-1)
- There is a minor difference in External planes and External points obtained from the quick hull and incremental algorithm because there is a possibility of planarity in the

points which leads to choosing a random point from the set. A little difference is tolerated and both can be treated as correct.(Fig 3)

Note:- Brute force code is tested only 10 times up to size 100 because of high time complexity.(Fig-4)

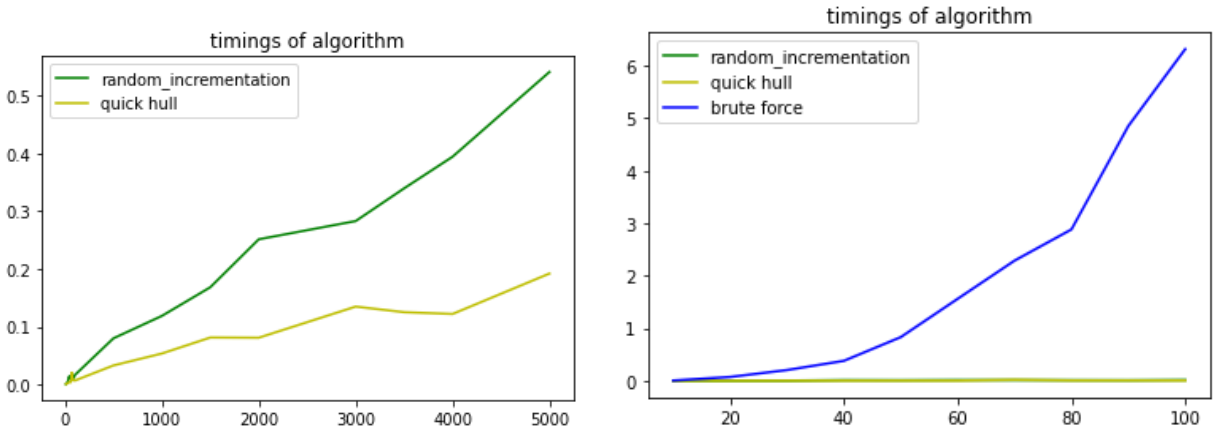


Fig:1 Graphical Representation of time taken by all 3 algorithms

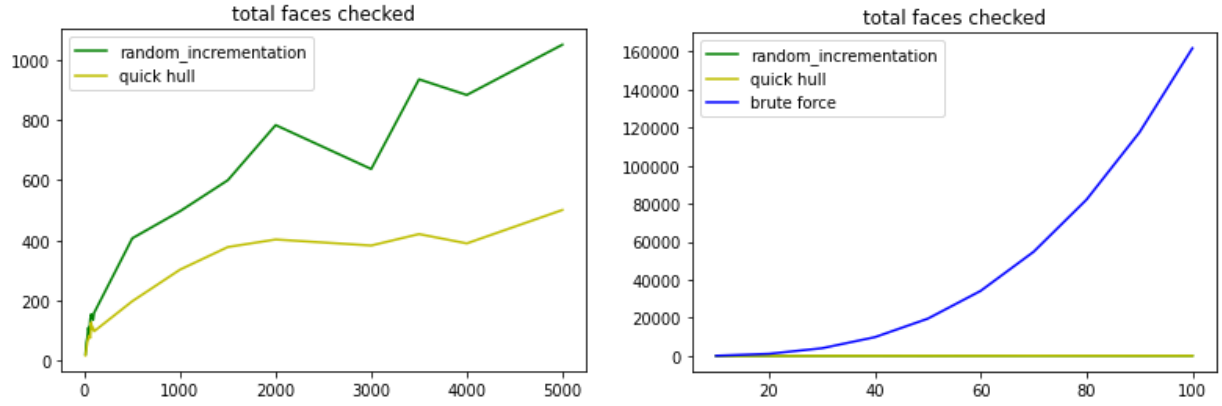


Fig:2 Graphical Representation of number of faces/planes checked by all 3 algorithms

```
Number of planes -  12 Number of vertices - 8
Number of planes -  22 Number of vertices - 13
Number of planes -  32 Number of vertices - 18
Number of planes -  38 Number of vertices - 21
Number of planes -  42 Number of vertices - 23
Number of planes -  42 Number of vertices - 23
Number of planes -  60 Number of vertices - 32
Number of planes -  54 Number of vertices - 29
Number of planes -  56 Number of vertices - 30
Number of planes -  48 Number of vertices - 26
Number of planes -  86 Number of vertices - 45
Number of planes -  138 Number of vertices - 71
Number of planes -  164 Number of vertices - 84
Number of planes -  178 Number of vertices - 91
Number of planes -  168 Number of vertices - 86
Number of planes -  188 Number of vertices - 96
Number of planes -  176 Number of vertices - 90
Number of planes -  222 Number of vertices - 113
```

```
Number of planes -  12 Number of vertices - 8
Number of planes -  26 Number of vertices - 15
Number of planes -  34 Number of vertices - 19
Number of planes -  42 Number of vertices - 23
Number of planes -  38 Number of vertices - 21
Number of planes -  46 Number of vertices - 25
Number of planes -  50 Number of vertices - 27
Number of planes -  58 Number of vertices - 31
Number of planes -  46 Number of vertices - 25
Number of planes -  58 Number of vertices - 31
Number of planes -  112 Number of vertices - 58
Number of planes -  116 Number of vertices - 60
Number of planes -  150 Number of vertices - 77
Number of planes -  172 Number of vertices - 88
Number of planes -  170 Number of vertices - 87
Number of planes -  196 Number of vertices - 100
Number of planes -  168 Number of vertices - 86
Number of planes -  252 Number of vertices - 128
```

Fig:3 Number of planes and vertices obtained in every iteration for QuickHull and Randomized Incremental Algorithms respectively

points	quick hull	random incremental	brute force	final external points	final external faces
10	(22,0.00055)	(26,0.00074)	(120,0.00653)	9	14
20	(45,0.00190)	(50,0.00204)	(1140,0.07363)	14	24
30	(62,0.00258)	(68,0.00286)	(4060,0.20542)	18	32
40	(67,0.00546)	(104,0.00895)	(9880,0.38078)	20	36
50	(103,0.00609)	(118,0.01044)	(19600,0.83230)	27	50
60	(79,0.00429)	(139,0.010295)	(34220,1.56199)	23	42
70	(92,0.00618)	(125,0.01164)	(54740,2.28932)	27	50
80	(93,0.00969)	(128,0.01004)	(82160,2.88316)	26	48
90	(152,0.01526)	(148,0.01547)	(117480,4.85289)	37	70
100	(143,0.01702)	(203,0.02552)	(161700,6.31663)	34	64
500	(221,0.03119)	(509,0.1255)	()	54	104
1000	(294,0.05998)	(548,0.15100)	()	72	140
1500	(340,0.07090)	(528,0.14523)	()	75	146
2000	(406,0.10750)	(825,0.34044)	()	87	170
3000	(399,0.11971)	(672,0.28369)	()	91	178
3500	(347,0.10741)	(9540,0.40385)	()	79	154
4000	(471,0.15421)	(909,0.42963)	()	101	198
5000	(442,0.18248)	(1035,0.56535)	()	99	194

Fig:4 Time taken, final external points and faces for different input test sizes

APPLICATIONS OF CONVEX HULL:-

- **Collision avoidance**: If the convex hull of a vehicle avoids collision with obstacles then so does the vehicle. Hence it used to plan routes and paths.
- **Smallest box**: For a 3D dimensional object, a convex hull is calculated to estimate the minimum rectangle or box enclosing the object.
- **Shape analysis**: Shapes can be determined using their convex hulls which helps in classification by matching their "convex deficiency trees"(trees representing polygons bounded by linear and curved edges that are subsets of convex curves).

INSTRUCTIONS TO RUN :

There are 3 codes going to be submitted :

1. QUICKHULL - main code :
    a. Run all the cells in ‘quickhull. ipynb’ file with paths corrected, currently, the paths are for collab and all the test datasets used were submitted in the folder test so, put the paths to the read input function wherever it is called.
2. Analysis - analysis code :
    a. It contains both quickhull and random incremental algorithms. To run quickhull, uncomment lines 11 to 18 and comment the line 19 and use restart and run all. To run incremental algorithm comment lines 11 to 18 and uncomment line 19 and use restart and run all. The tests here are implemented randomly through NumPy so no need for a test data set for this.
3. Test-Quick hull :
    a. Datasets like teacup set points etc., are used in this code. Since there are highly complex 3-d pictures,(in rotated form) the input output bound increases. We tried to test this for 20 different datasets but were successful only for 10 tests because of RAM issues in the colab. So it’s better to collapse other cells and open and run only 1 cell at a time to avoid system hang issues.

Directory Structure :

Src : have all the 3 codes

Test outputs : the images that are saved (convex hulls of 10-different data sets)

Obj_files : datasets used for testing

Report : Report which has theory, results and analysis parts.

References:

Quick hull - https://www.cise.ufl.edu/~ungor/courses/fall06/papers/QuickHull.pdf

Data collection - https://github.com/leomccormack/convhull_3d

Works done :

- Algorithm working, and its correctness - understood by everyone.
- Coding part implementation

  Siva and Mohit - final phase and random incremental and brute force analysis part.

  Naveen and tejaswini : initial phase and dataset collection + plotting.

- Report writing :

  The theory part - Tejaswini and Naveen.

  Results and the analysis part : Mohit and Siva.

Further exploration or extension of the algorithm :

The algorithm remains same for n dimensional except the geometry and hence we can extend our implementation to n dimensional by making necessary changes.