Quickhull Algorithm

ECM3428 - Algorithms that Changed the World

680033128 - 161678
Word Count: 1486

December 16, 2021

Abstract

1 Introduction and Motivation

The convex hull of a set of points s is the smallest convex set containing all the points in s. The convex set forms the convex polygon that encloses all the points on the plane.

We call a shape convex, if for any two points that are inside or lie on the boundary of the shape, the line between these two points is also inside the shape. If you look at concave case you see that it is easy to find two points inside the shape where the line has to leave and then enter the shape again.

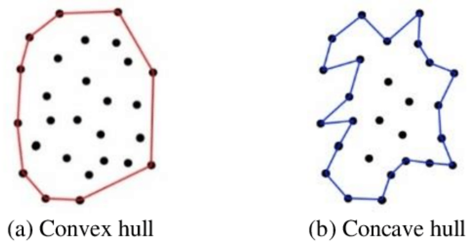


Figure 1: Classification of convex and concave hull [1]

2 Main Principles

Quickhull is a method of computing the convex hull of a finite set of points in the plane. It uses a divide and conquer approach similar to that of quicksort, from which its name is derived. Its average case complexity is considered to be $O(n \log(n))$, whereas in the worst case it takes $O(n^2)$ [2].

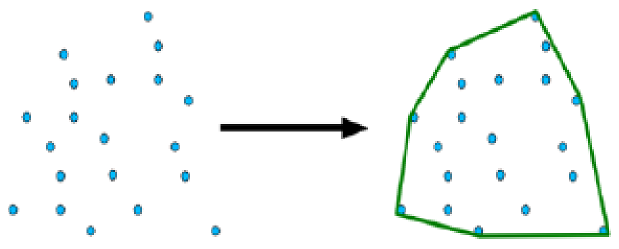


Figure 2: Illustration of points contained within 2D convex hull [3]

The quickhull algorithm takes an input of an array of points specified by the x and y coordinates, depicted on the left side of *Figure 2*. The algorithm determines the convex set that encloses every points within the given set as shown on the right side of *Figure 2*.

Convex hull finding algorithms can be viewed as a data-parallel problem. For each iteration, the operations done per point is the same. Each point goes through some test to see whether or not it should be discarded, and those points which are left are regrouped and tested again until the final set of convex hull points are determined

All divide and conquer algorithms share the similar step of partitioning data into smaller subsets and operating on the subsets via recursive calls. The Quickhull algorithm discards data during each recursive call, cutting down on the workload as the algorithm goes deeper into the recursion by ignoring those points found not to be a part of the convex hull. [4]

3 Pseudo Code

Algorithm 1 2-dimensional Quickhull pseudo code for set of points s

```
1: Find the points with minimum and maximum $x$ coordinates
2: Add extrema points to the convex hull set
3: Remove extrema points from the set $s$
4: Form a line $L$ joining these extrema points        ▷ This line will divide the whole set into two subsets.
5: for points in s do
6:     if point above the line $L$ then
7:         Add point to the set above the line s(Above)
8:     else
9:         Add point to the set below the line s(Below)
10:    end if
11: end for
12: for Each set s(N) do
13:     for Every point in s(N) do
14:         Determine the point with the maximum distance from the hull set.
15:         Include this point in the convex hull set.
16:         Create a new polygon with the updated convex hull set.
17:     end for
18:     for Each point in s(N) do
19:         if point is encapsulated by the polygon then
20:             Discards point from the set.
21:         end if
22:     end for
23: end for
```

4 Time and Space Complexity Analysis

The Quickhull algorithm worst case complexity for 2-dimensional and 3-dimensional space is considered to be $O(n \log(n))$, where n is the number of input points and r is the number of processed points [2]

For the purely 2-dimensional implementation that is considered in this report the Quickhull algorithm has the following complexity:

- Worst case time complexity: $O(n^2)$
- Average case time complexity: $O(n \log(n))$
- Best case time complexity: $O(n \log(n))$
- Space complexity: $O(n)$

5 Limitations and Constraints

The worse performing case of Quickhull can be realised by constructing points on the circumference of a circle. The following rule should be adhered to $P_i = (r, \pi/2i)$ in polar coordinates.

With this set of points the furthest element from the existing convex hull set will always be the one of the points on the circle. So in each step of the recursion you only eliminate one point. This is because the convex

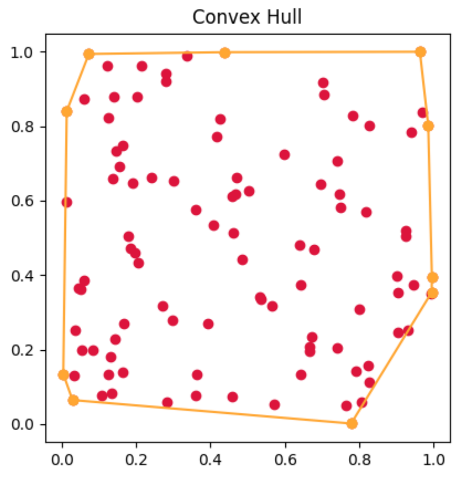


Figure 3: Average Case Complexity

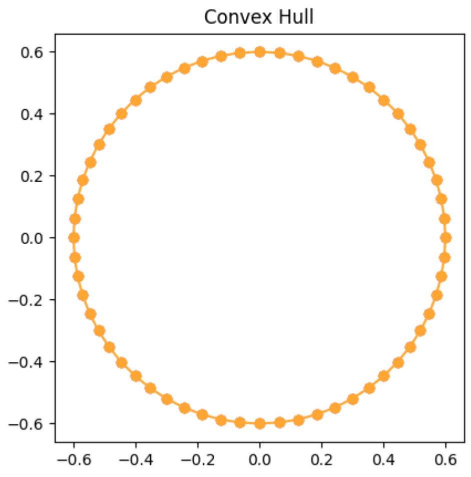


Figure 4: Worst Case Complexity

hull set encloses an empty subset with no points being surrounded by the polygon. The height of the recursion tree is therefore $O(n)$ resulting in an overall efficiency of $O(n^2)$.

Another limitation of the Quickhull algorithms i that it is a static algorithm that requires all points to be known at the start. Suppose that points reach sequentially one after one and the convex hull needs to be maintained at each insertion of a point. A static algorithm would need to reconstruct the new convex hull from scratch to handle this additional point. This would be processed in $O(n \log(n))$ time and is therefore inefficient.

5.1 Advances that could overcome these

[5] proposed an efficient planar incremental convex hull algorithm to find the edges of the boundary polygon of the convex hull of a set of points. The convex hull is maintained as a set of line segments allowing for quicker evaluation of new points added to the set. This alteration of Quickhull aims to remove the need reconsider all previously included points which thereby consumes fewer computational resources and proposes $O(h^2)$ time complexity where h is the number of points that form the convex hull.

For the standard Quickhull implementation N.-D Hoang *et al* proposed using orienting vectors. They further address the redundant Orient operations of the original Quickhull algorithm and the inclusion of data preprocessing and separation step. The time complexity is reduced by introducing eight special extreme points of the input set and removing unnecessary points inside the polygon formed by these extreme points, this step reduces the size of input data and separates the original problem into sub- problems which can be solved faster [6].

Their implementation is as follows:
Given two points, say p and q, a set S of points lying to the left of the line pq, and one of the farthest points among the points of S, say r, from the line pq. Take a vector $\vec{tr}$ to be an orienting vector of (p, q, r) then the following assertions hold true:

(i) *If* $s \epsilon S$ *and* $Orient(p, r, s) > 0$, *then* $Orient(\vec{tr}, s) > 0$.

(ii) *If* $s \epsilon S$ *and* $Orient(r, q, s) > 0$, *then* $Orient(\vec{tr}, s) < 0$.

The inclusion of this preprocessing step heavily reduces the number of points that need to be considered to determine the convex hull. Figure 2 below depicts in red the points which are removed by the inclusion of the eight special extreme points and the polygon formed by combination of these. The points in green are the remaining points to be evaluated by Quickhull. Obviously, the points that lie inside or on the edges of the polygon connecting these extreme points except the eight extreme points are not vertices of the convex hull and can be ignored. The convex hull of the original set coincides with the convex hull of the set of the remaining points. However, we do not need to check all the points of the initial set in order to find the remaining points.

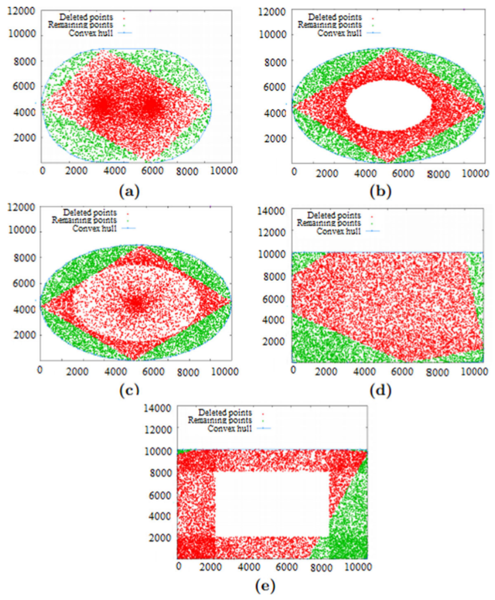


Figure 5: Preprocessing and separation for five types of data [6]

From here we only need to consider the four unresolved smaller sets surrounding the ignored points. This greatly increases the computational efficiency of running Quickhull on the remaining search space.

6 Applications of the Quickhull Algorithm

Convex hulls and by extension the Quickhull algorithm can be applied to collision detection. This can be useful for dynamic games objects which are usually approximated by simple shapes

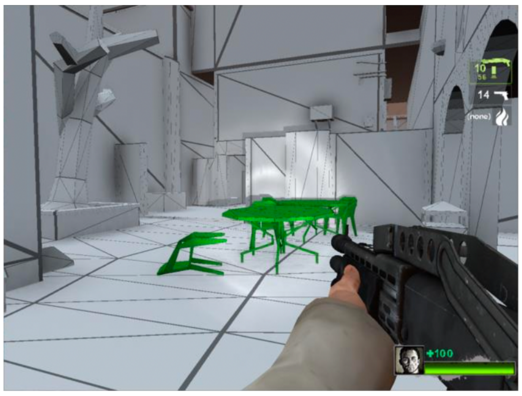


Figure 6: Approximating object shapes for collision detection [7]

Dynamic game objects are usually approximated by simpler shapes for collision detection since using the render geometry would not be efficient. Convex hulls are a good candidate since they can approximate even complex geometry quite well. In 2D we used the rubber band analogy to get some intuition for a convex hull while in 3D you can think of it as shrink wrapping the object. Enveloping all the points that make up the chair and table with a convex hull and doing the same to all the point that constitute a player would allow us to perform simple, rudimentary collision detection. This is useful to circumvent players walking through objects that they should otherwise have to avoid or walk around. This can be extended to incorporate path planning.

Other applications of Quickhull include dynamic vehicle detection. This is important for the safety of autonomous vehicles [8].

The convex hull of an object's point-clouds is first extracted and convex hull of the car along with the location, direction and speed of the two entities is used to avoid the convex hull of the target overlapping that of the car which would result in a crash.

Due to the need for real-time information to feed the decision making of autonomous vehicles it becomes more efficient to approximate the outline of an object using a convex hull rather than mapping the detailed specification of the obstacle [8]. Therefore, Quickhull is useful for inclusion in accurate computer vision technologies for approximating 3D models.

Quickhull has also been explored in the following applications. It has been used for stress analysis, stability of robot grasps, classification of molecules by their biological activity, vibration reduction, support structures for layered manufacturing [2], analysis of spectrometry data [9], constrained control allocation [10], robot navigation [11] and micromagnetic modeling [12].

References

[1] Z. Yahya, R. W. Rahmat, F. Khalid, A. Rizaan, and A. Rizal, "A concave hull based algorithm for object shape reconstruction," *Int. J. Inf. Technol. Comput. Sci*, vol. 9, pp. 1–9, 2017.

[2] C. B. Barber, D. P. Dobkin, and H. Huhdanpaa, "The quickhull algorithm for convex hulls," *ACM Transactions on Mathematical Software (TOMS)*, vol. 22, no. 4, pp. 469–483, 1996.

[3] R. Braga, A. Valdetaro, G. Nunes, A. Raposo, and B. Feijó, "Evaluating the visibility algorithm of point-based graphics for real-time applications," *ResearchGate*, 2011.

[4] S. Tzeng and J. D. Owens, "Finding convex hulls using quickhull on the gpu," *arXiv preprint arXiv:1201.2936*, 2012.

[5] K. Wijeweera and S. Kodituwakku, "An efficient planar incremental convex hull algorithm to find the edges of the boundary polygon of the convex hull of a set of points," *Ceylon Journal of Science*, vol. 50, no. 3, pp. 261–268, 2021.

[6] N.-D. Hoang and N. K. Linh, "Quicker than quickhull," *Vietnam Journal of Mathematics*, vol. 43, no. 1, pp. 57–70, 2015.

[7] "Implementing quickhull," 2014. [Online]. Available: http://media.steampowered.com/apps/valve/2014/DirkGregorius$_ImplementingQuickHull.pdf[Accessed : 03 - Dec - 2021]$

[8] K. Liu and J. Wang, "Fast dynamic vehicle detection in road scenarios based on pose estimation with convex-hull model," *Sensors*, vol. 19, no. 14, p. 3136, 2019.

[9] J. W. Boardman *et al.*, "Automating spectral unmixing of aviris data using convex geometry concepts," in *Proc. Summaries 4th Annu. JPL Airborne Geosci. Workshop*, vol. 1, 1993, pp. 11–14.

[10] K. A. Bordingnon and W. C. Durham, "Closed-form solutions to constrained control allocation problem," *Journal of Guidance, Control, and Dynamics*, vol. 18, no. 5, pp. 1000–1007, 1995.

[11] P. Cucka, N. S. Netanyahu, and A. Rosenfeld, "Learning in navigation: Goal finding in graphs," in *Studies In Pattern Recognition: A Memorial to the Late Professor King-Sun Fu.* World Scientific, 1996, pp. 39–56.

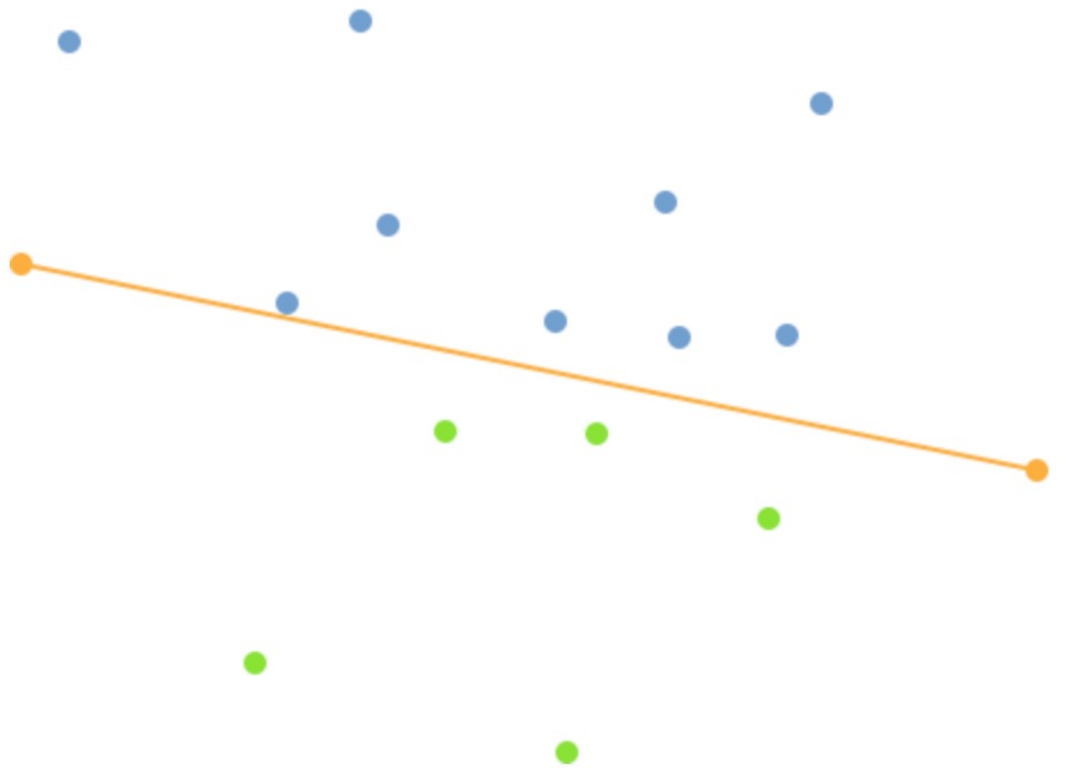
[12] D. Porter, E. Glavinas, P. Dhagat, J. O'Sullivan, R. Indeck, and M. Muller, "Irregular grain structure in micromagnetic simulation," *Journal of Applied Physics*, vol. 79, no. 8, pp. 4695–4697, 1996.

Quickhull Algorithm

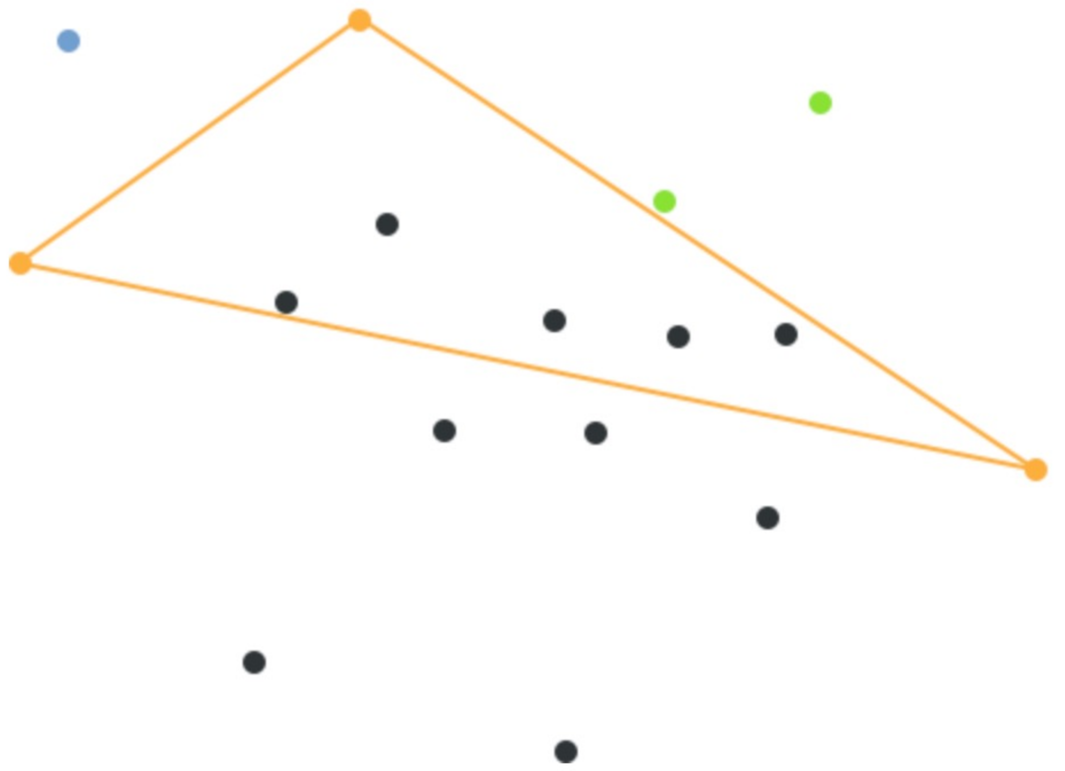
Algorithms that Changed the World

680033128

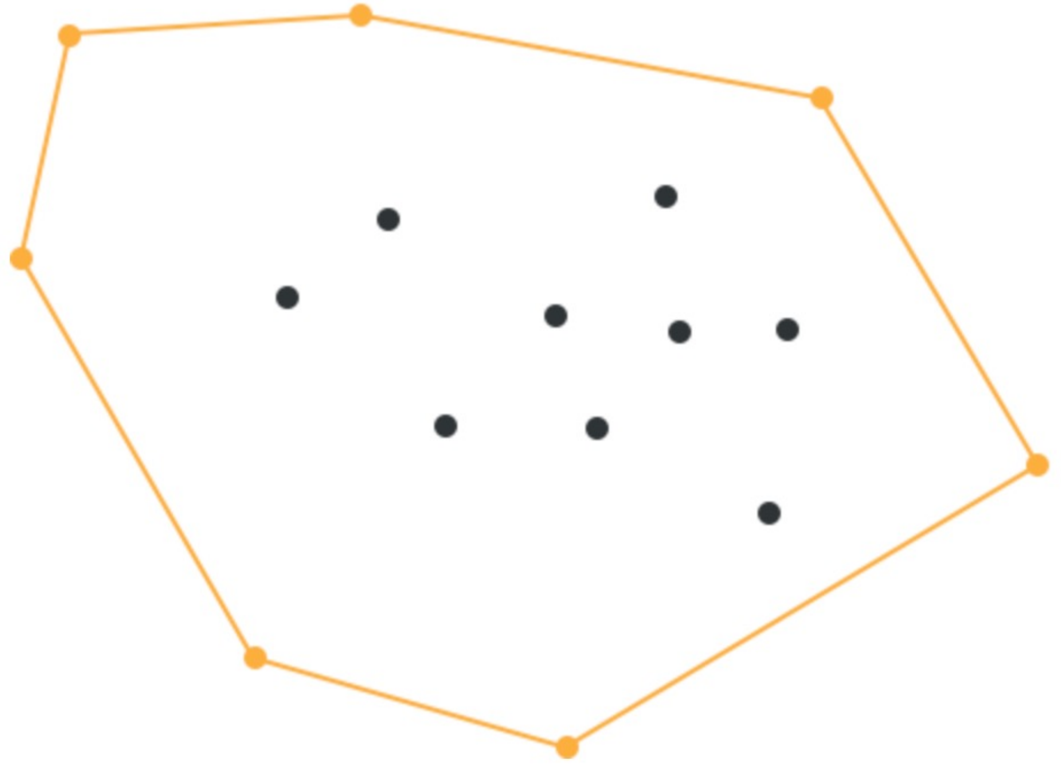
Divide the points into two subsets



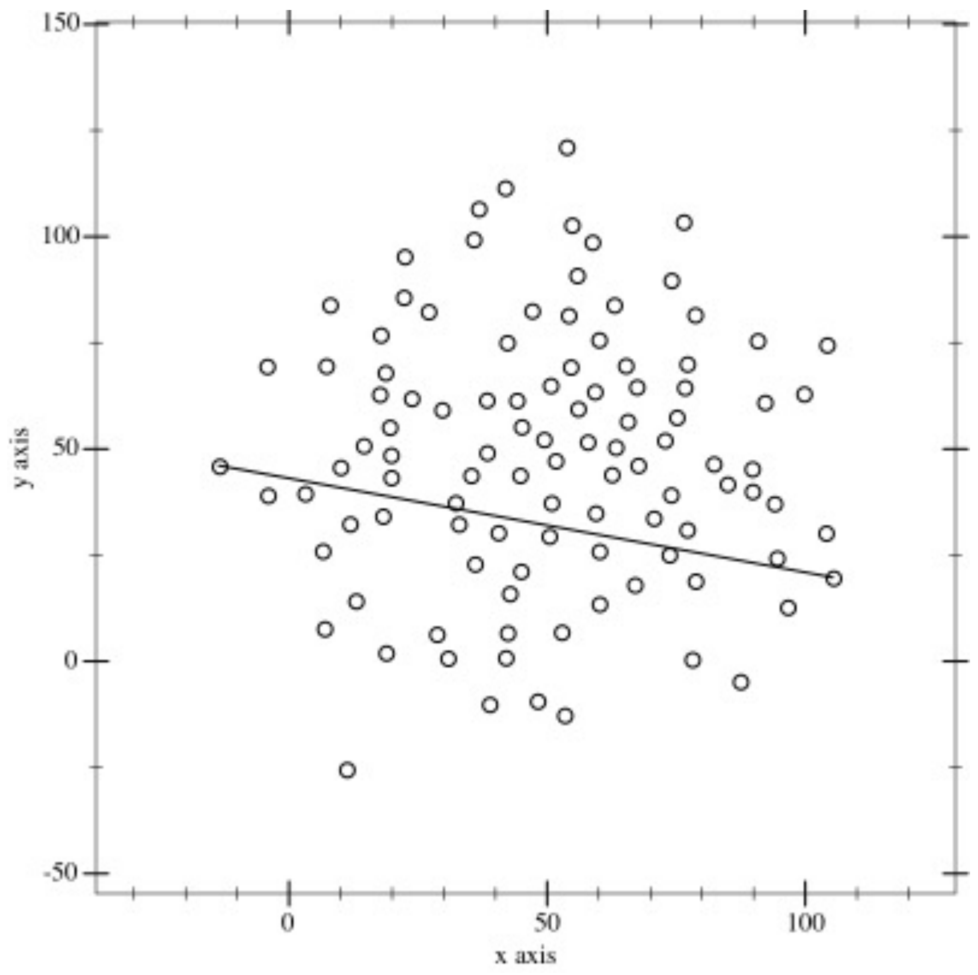
Find maximal distance point, ignore points inside triangle and repeat



Recurse until no more points are left and recombine the two subsets

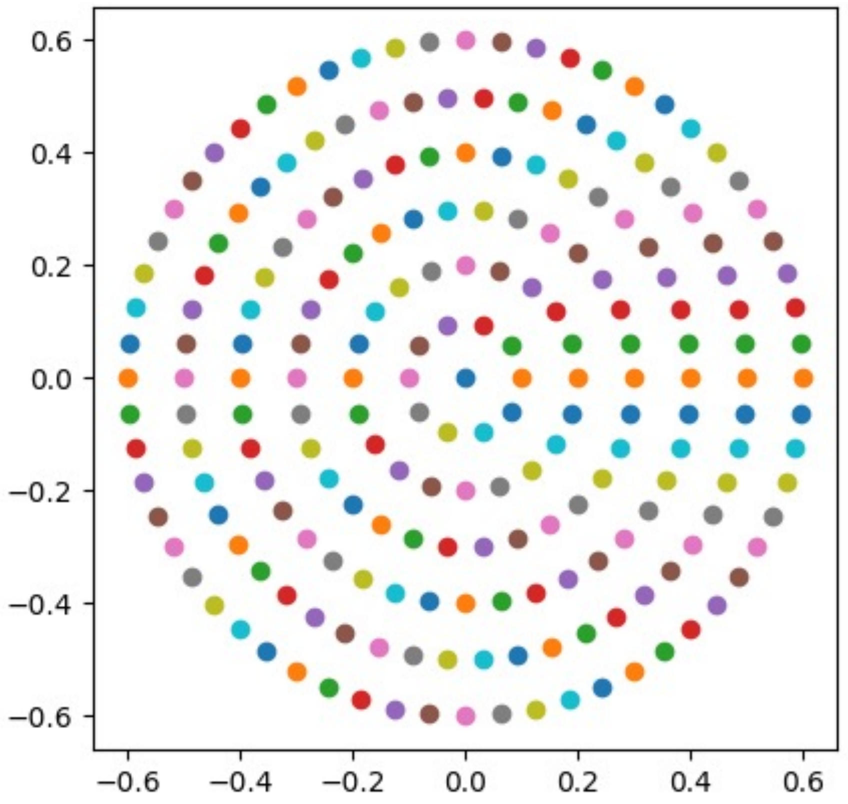


This animation depicts the Quickhull algorithm.



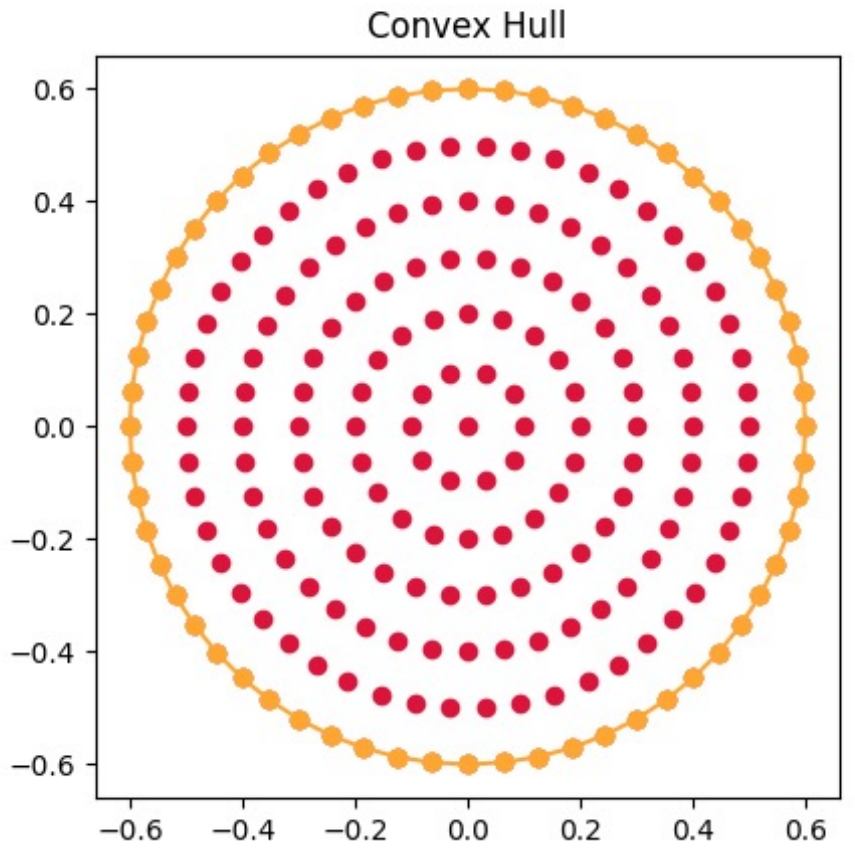
Weird Cases

Concentric circle distribution



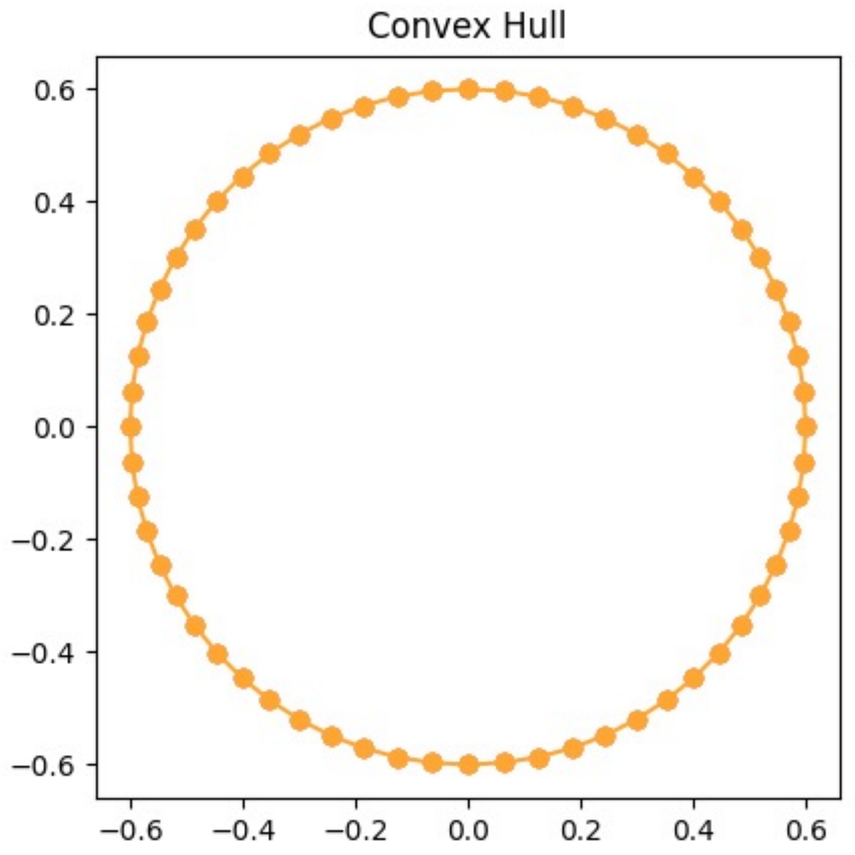
Average Case

Convex Hull envelopes points within the outer circle perimeter



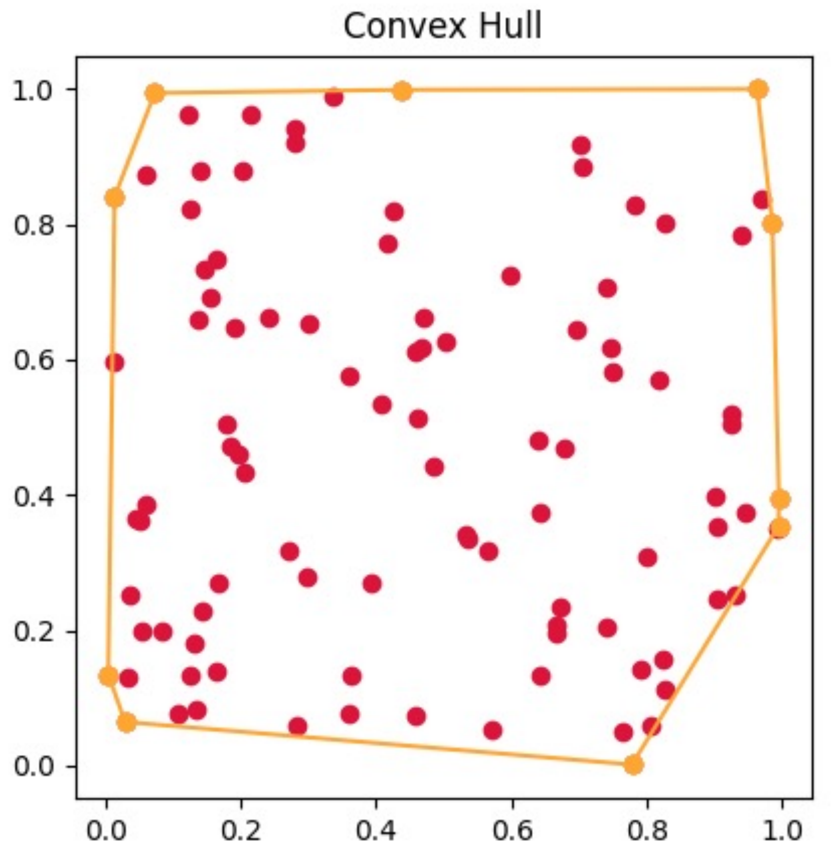
Worst Case

Removing the internal points leads to only a single point being added to the convex hull each iteration and zero internal points are eliminated from the evaluation set [1].



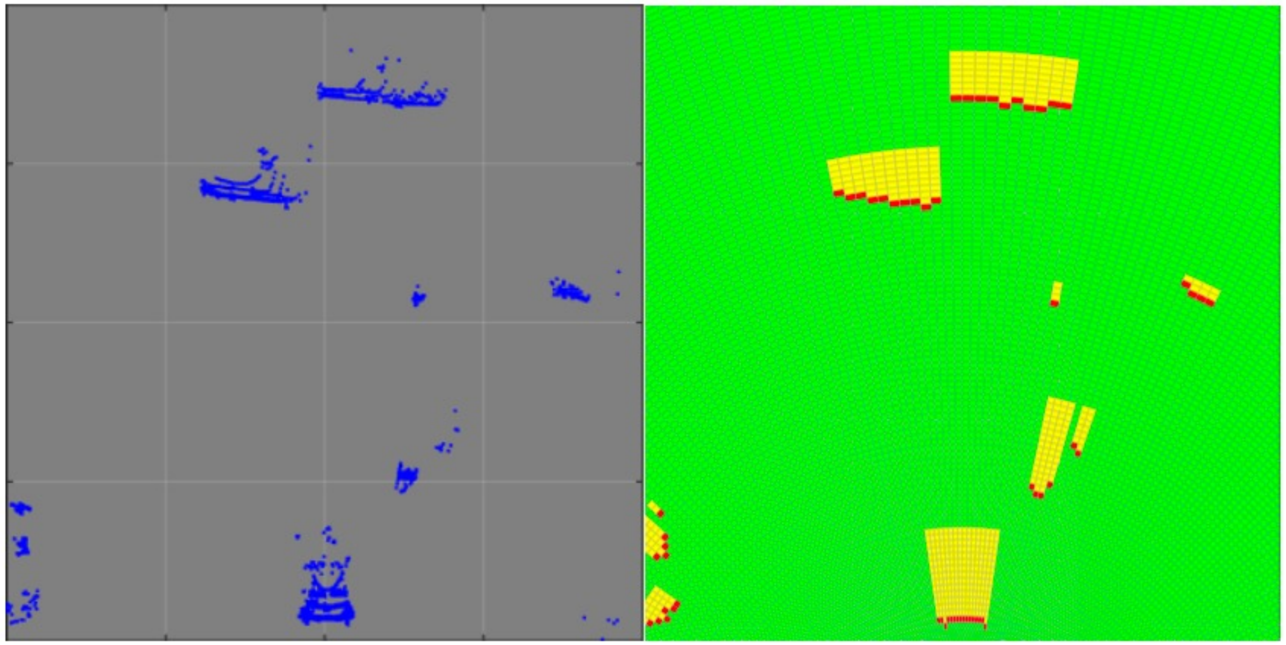
Average Case

Random uniform distribution



Applications

Virtual-Scan Mapping

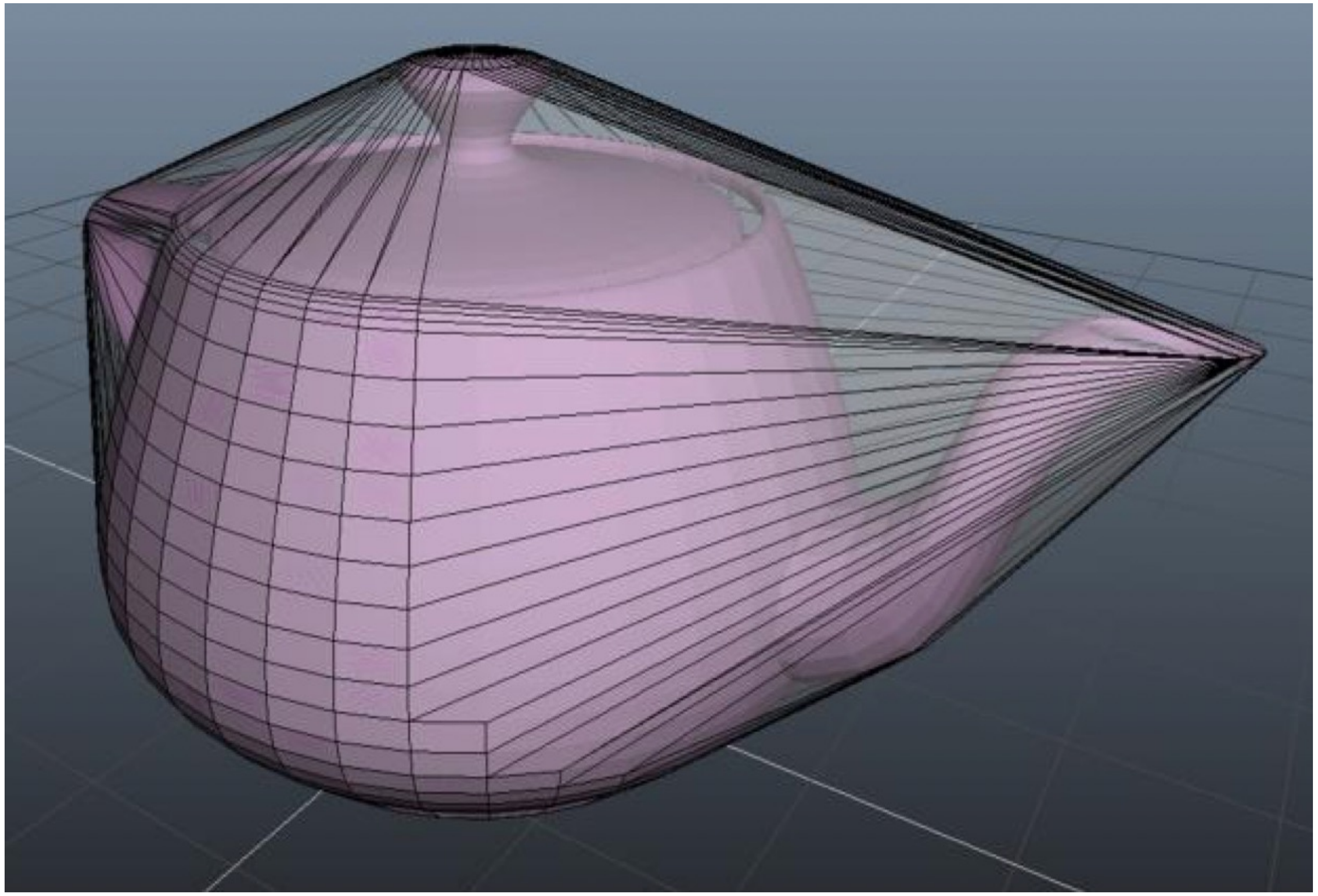


Virtual-scan mapping. (**left**) Point-clouds; (**right**), corresponding virtual scan. States of grids in green, red, and yellow are free, occupied, and occluded, respectively.

[2]

Applications Continued

Collision Detection



[3]

References

- [1] C. B. Barber, D. P. Dobkin, and H. Huhdanpaa, “The quickhull algorithm for convex hulls,”ACM Trans-actions on Mathematical Software (TOMS), vol. 22, no. 4, pp. 469–483, 1996
- [2] K. Liu and J. Wang, “Fast dynamic vehicle detection in road scenarios based on pose estimation withconvex-hull model,”Sensors, vol. 19, no. 14, p. 3136, 2019.
- [3] “Implementing quickhull,” 2014. [Online]. Available: http://media.steampowered.com/apps/valve/2014/DirkGregoriusImplementingQuickHull.pdf[Accessed: 03−Dec−2021]

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt


def generate_points():
    """
    Generate random points (x,y)
    :return: List of randomized point
    :rtype: DataFrame
    """
    try:
        dist = input("Enter 'R' for random points distribution or 'C' for
circular distribution: ").lower()
    except:
        dist = 'r'

    if dist == 'c':
        return circle_points()
    else:
        try:
            count = int(input("Enter number of points: "))
        except:
            count = 200
        return random_distribution(count)


def random_distribution(count):
    x_min = 0
    x_max = 1
    y_min = 0
    y_max = 1

    x_list = np.random.uniform(x_min, x_max, size=count)
    y_list = np.random.uniform(y_min, y_max, size=count)

    point_list = pd.DataFrame({'x': x_list, 'y': y_list})
    print(point_list)
    return point_list


def circ_fig():
    n = [1, 10, 20, 30, 40, 50, 60]
    r = [0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6]
    concentric_circles = []

    for r, n in zip(r, n):
        t = np.linspace(0, 2 * np.pi, n, endpoint=False)
        x = r * np.cos(t)
        y = r * np.sin(t)
        concentric_circles.append(np.c_[x, y])

    fig, ax = plt.subplots()
    for circle in concentric_circles:
        ax.scatter(circle[:, 0], circle[:, 1])
    ax.set_aspect('equal')
    plt.show()


def circle_points():
    T = [1, 10, 20, 30, 40, 50, 60]
    R = [0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6]
    x_list = []
    y_list = []

    fig, ax = plt.subplots()
    for r, t in rtpairs(R, T):
        x_list.append(r * np.cos(t))
        y_list.append(r * np.sin(t))
        ax.plot(r * np.cos(t), r * np.sin(t), 'o')
    ax.set_aspect('equal')
    plt.show()

    point_list = pd.DataFrame({'x': x_list, 'y': y_list})
    print(point_list)
    return point_list


def rtpairs(r, n):
    for i in range(len(r)):
        for j in range(n[i]):
            yield r[i], j*(2 * np.pi / n[i])


def check_cond(point_1, point_2, check_point):
    """
    Check a point position based on a line created by 2 points (above /
below / on)
    :param point_1: line first point
    :type point_1: DataFrame element
    :param point_2: line second point
    :type point_2: DataFrame element
    :param check_point: point which is going to be checked
    :type check_point: DataFrame element
    :return: point position
    :rtype: string
    """

    try:
        p1_x, p1_y, p1_c = point_1
    except:
        p1_x, p1_y = point_1

    try:
        p2_x, p2_y, p2_c = point_2
    except:
        p2_x, p2_y = point_2

    try:
        pc_x, pc_y, pc_c = check_point
    except:
        pc_x, pc_y = check_point

    # Use determinant to determine position
    det = (p2_x - p1_x) * (pc_y - p1_y) - (p2_y - p1_y) * (pc_x - p1_x)

    if det > 0:
        return "above"
    elif det < 0:
        return "below"
    else:
        return "on"
```

```python
def filter_above_below(left_most, right_most, point_list):
    """
    Get all above/below condition
    :param left_most: line left most point (end)
    :type left_most: DataFrame element
    :param right_most: line right most point (end)
    :type right_most: DataFrame element
    :param point_list: list of points
    :type point_list: DataFrame
    :return: list of points on line, above and below
    :rtype: DataFrame
    """

    # Create an empty list for handle point position
    temp = np.empty(point_list.index[-1] + 1, dtype='U8')

    # Check every point position
    for index, row in point_list.iterrows():
        temp[index] = check_cond(left_most, right_most, row)

    # Filter out None in list
    temp = temp[temp != ''].copy()

    # Add a column of condition (above / below)
    point_list['cond'] = temp

    # Return copy of masked data frame
    line_above = point_list.loc[point_list['cond'] == 'above'].copy()
    line_below = point_list.loc[point_list['cond'] == 'below'].copy()

    return line_above, line_below


def left_most_point(point_list):
    """
    Get the left most point
    :param point_list: list of point
    :type point_list: DataFrame
    :return: left most point
    :rtype: DataFrame element
    """
    return point_list.loc[point_list['x'].idxmin()]


def right_most_point(point_list):
    """
    Get the right most point
    :param point_list: list of point
    :type point_list: DataFrame
    :return: right most point
    :rtype: DataFrame element
    """
    return point_list.loc[point_list['x'].idxmax()]


def get_furthest_point(line_above, line_below, point_list):
    """
    Get furthest point
    :param line_above: line above point
    :type line_above: DataFrame element
    :param line_below: line below point
    :type line_below: DataFrame element
    :param point_list: list of point to check
    :type point_list: DataFrame
    :return: furthest point
    :rtype: DataFrame element
    """

    def distance(line_above, line_below, point):
        """
        Get distance from a point to the line created by 2 points
        :param line_above: line above point
        :type line_above: DataFrame element
        :param line_below: line below point
        :type line_below: DataFrame element
        :param point: point to check
        :type point: DataFrame element
        :return: distance
        :rtype: float
        """

        try:
            x0, y0, c0 = point
        except:
            x0, y0 = point

        try:
            x1, y1, c1 = line_above
        except:
            x1, y1 = line_above

        try:
            x2, y2, c2 = line_below
        except:
            x2, y2 = line_below

        # Calculate distance
        top_eq = abs((y2 - y1) * x0 - (x2 - x1) * y0 + x2 * y1 - y2 * x1)
        bot_eq = ((y2 - y1) ** 2 + (x2 - x1) ** 2) ** 0.5

        return top_eq / bot_eq

    furthest_point = None
    max_distance = 0

    # Find the furthest
    for index, point in point_list.iterrows():
        point_distance = distance(line_above, line_below, point)
        if point_distance > max_distance:
            max_distance = point_distance
            furthest_point = point

    return furthest_point


def quick_hull(line_above, line_below, point_list):
    """
    Algorithm to find the most outer line of a clustered points
    :param line_above: line above edge
    :type line_above: DataFrame element
    :param line_below: line below edge
    :type line_below: DataFrame element
    :param point_list: list of point to check
    :type point_list: DataFrame
    :return: ines connecting the outer points
    :rtype: list
    """

    # If there's no more point to check (basis)
    if len(point_list['x']) == 0:
        return [(line_above, line_below)]

    # Get furthest point from line
    furthest_point = get_furthest_point(line_above, line_below, point_list)

    # Divide point into ABOVE-line's above / below
    point_on_line_above, point_on_line_below =
filter_above_below(line_above, furthest_point, point_list)

    # Delete side that's useless
    delete_side = check_cond(line_above, furthest_point, line_below)

    # Choose one side that's outer
    rec_point_list = None
    if delete_side == "below":
        rec_point_list = point_on_line_above
    elif delete_side == "above":
        rec_point_list = point_on_line_below

    # Recurse
    outer_line_above = quick_hull(line_above, furthest_point,
rec_point_list)

    # Divide point into BELOW-line's above / below
    point_on_line_above, point_on_line_below =
filter_above_below(furthest_point, line_below, point_list)

    # Delete side that's useless
    delete_side = check_cond(furthest_point, line_below, line_above)

    # Choose one side that's outer
    rec_point_list = None
    if delete_side == "below":
        rec_point_list = point_on_line_above
    elif delete_side == "above":
        rec_point_list = point_on_line_below

    # Recurse
    outer_line_below = quick_hull(furthest_point, line_below,
rec_point_list)

    return outer_line_above + outer_line_below


def show_output(tuple_list):
    """
    Format output so it's easier to read
    :param tuple_list: list of tuple indicate line
    :type tuple_list: list
    """
```

```python
    final = []
    for point_1, point_2 in tuple_list:
        line = ((point_1['x'], point_1['y']), (point_2['x'], point_2['y']))
        final.append(line)

    print("Convex Hull")
    for item in final:
        print(item)


def draw(point_list, tuple_list):
    """
    Display visualization
    :param point_list: list of point at start
    :type point_list: DataFrame
    :param tuple_list: list of tuples indicating line
    :type tuple_list: List
    """

    # Setup
    fig = plt.figure(1)
    canvas = fig.add_subplot(111, facecolor='#FFFFFF')
    fig.canvas.draw()

    # Create init scatter
    x = point_list['x']
    y = point_list['y']
    canvas.scatter(x, y, color="#DC143C")

    # Parse line tuple
    final_x = []
    final_y = []
    for point_1, point_2 in tuple_list:
        # Append line edge to list
        final_x.append(point_1['x'])
        final_x.append(point_2['x'])
        final_y.append(point_1['y'])
        final_y.append(point_2['y'])

        # Create line
        list_x = [point_1['x'], point_2['x']]
        list_y = [point_1['y'], point_2['y']]
        canvas.plot(list_x, list_y, color="#ffa632")

    # Create line edge scatter
    canvas.scatter(final_x, final_y, color="#ffa632")
    # Maintains fixed aspect ratio
    canvas.set_aspect('equal')

    plt.title('Convex Hull')
    plt.show()


if __name__ == '__main__':
    # Circular figure
    # circ_fig()

    # Initialize point
    point_list = generate_points()

    # Find the left and right most
    left_most = left_most_point(point_list)
    right_most = right_most_point(point_list)

    # Separate the rest of point into two
    point_on_line_above, point_on_line_below =
filter_above_below(left_most, right_most, point_list)

    # Apply quick hull algorithm
    outer_line_above = quick_hull(left_most, right_most,
point_on_line_above)
    outer_line_below = quick_hull(left_most, right_most,
point_on_line_below)
    outer_final = outer_line_above + outer_line_below

    # Display output
    show_output(outer_final)
    draw(point_list, outer_final)
```