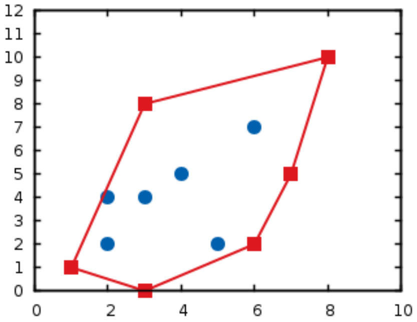
Project Writeup

Dan Crayne
CS 350, Fall 2016

1 Introduction

A convex hull is a subset C of points from set S which bound all other points $S - C$ (see Figure 1). We will talk about three different algorithms to calculate this subset: 1) *Quickhull* which is a divide and conquer algorithm with average-case complexity $\Theta(n\,log\,n)$, but worst-case complexity of $\Theta(n^2)$; 2) *Grahm Scan* which is an iterative approach based on calculating angles from a base point, with complexity $\Theta(n\,log\,n)$; and 3) *Monotone Chain* which is a variation on Grahm Scan which simplifies the algorithm by not requiring angle calculations. The goal of this paper is to compare the characteristics of each algorithm using empirical data obtained by recording running times of a sample implementation in Python.

Figure 1: A Convex Hull



2 Algorithm Descriptions

2.1 Finding the Orientation of a Point

Before describing the specific algorithms, we should develop a geometric tool which is essential to all algorithms given. This tool provides a method to find the orientation of some point p_i relative to a line $P_{i-1}P_{i-2}$. To accomplish this, we use the determinant of the three points $p_i = (x_1, y_1)$, $p_{i-1} = (x_2, y_2)$, and $p_{i-2} = (x_3, y_3)$, the result of which can be used to determine if the current point lies to the *left*, *right*, or *on* the line $p_{i-1}p_{i-2}$.

$$d_i = \begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} = x_1y_2 + x_3y_1 + x_2y_3 - x_3y_2 - x_2y_1 - x_1y_3$$

If $d_i > 0$ then p_i is to the left of p_1p_n, if $d_i < 0$ then p_i is to the right of p_1p_n, and if $d_i = 0$ then p_i is on P_1P_n. Implementation is trivial, and will show up in our code listings as the function $determinant(p_{i-2}, p_{i-1}, p_i)$ *(adapted from Levitin, 197)*.

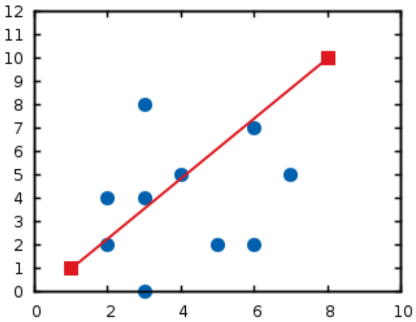
2.2 Quickhull

Overview

Quickhull is a divide and conquer algorithm for finding the convex hull of a set of points. It begins by finding the minimum and maximum x values of the set and then uses them as a dividing line to split the points into a lower hull and upper hull (see Figure 2). Next the convex hulls of these two subsets are recursively found, and finally the results are combined, giving the convex hull of the original set.

Dividing the Points

Figure 2: Dividing the Points



The first step in Quickhull is to find the two points in our set of points (named *points*) with minimum and maximum x values; we will call them p_1 and p_n respectively. A simple linear search is done to obtain the values, taking $\Theta(n)$ time. We know that these points are on the convex hull, and so we later append them to the list of upper convex hull points. First, we use p_1 and p_n to describe a line, which will divide *points* into two subsets of points, *upper_points* and *lower_points*. To make the split, we find the points in *points* that are to the right of the line P_nP_1 for *upper_points* and the points in *points* which are to the right of the line P_1P_n for *lower_points*. The splitting is accomplished with two wrapper functions, *get_lower_points* and *get_upper_points* each taking $\Theta(n)$ time, and having the following form:

Listing 1: Getting Upper and Lower Subsets

```
    for point in points:
        if determinant(p1, pn, point) < 0:
            subset_points.append(point)
```

Recursively Finding the Hull

Now that we have *upper_points* and *lower_points*, we can recursively find the convex hull of each and combine the results:

Listing 2: Preparing for Recursion

```
upper_convex_hull = get_hull(upper_points, p1, pn)
upper_convex_hull.append(p1)
upper_convex_hull.append(pn)
lower_convex_hull = get_hull(lower_points, pn, p1)

return upper_convex_hull + lower_convex_hull
```

The *get_hull* function is the meat-and-potatoes of the algorithm, recursively finding the hull of its first argument, a subset of the original *points*.

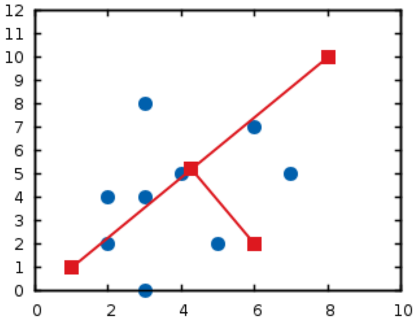
Listing 3: Getting the Hull

```
def get_hull(points, p1, pn):
    convex_hull = []
    if not points:
        return convex_hull

    pmax = find_furthest_point_from_line(points,
                                         p1, pn)
    convex_hull.append(pmax)
    points1 = find_points_to_left_of_line(points,
                                          p1, pmax)
    points2 = find_points_to_left_of_line(points,
                                          pmax, pn)

    return convex_hull + get_hull(points1, p1, pmax)
```

```
                                  + get_hull(points2, pmax, pn)
```

After ensuring that our subset is not empty, it finds the point which is furthest away from P_1P_n (see Figure 3) by using the magnitude of the determinant of the two points making up the line and each point in our subset (in this case we we use the determinant as a distance measure).

Figure 3: Finding the Furthest Point

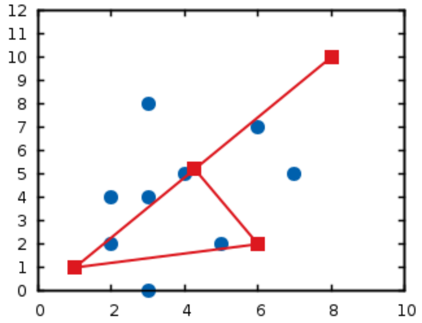


Listing 4: Finding the Furthest Point

```
def find_furthest_point_from_line(points, line_pt1,
                                  line_pt2):
    max_point = points[0]
    max_distance = 0

    for point in points:
        distance_to_point = abs(determinant(
                             line_pt1, line_pt2, point))
        if distance_to_point > max_distance:
            max_distance = distance_to_point
            max_point = point

    return max_point
```

Next, since we know that p_{max} is part of the convex hull, we add it, and create the next two subsets of *points* which will be recused upon to find the next hull values. These subsets, *points*1 and *points*2, are the points beyond the triangle $P_1P_{max}P_n$ (see Figure 4). (The points within this triangle are disregarded.) When we have exhausted the points in a subset, *get_hull* will return the points found up to this time. Once all subsets have been searched,

Figure 4: Triangle $P_1 P_{max} P_n$



the recursion unwinds, and it returns all convex hull points for both the original upper and lower subsets; we then combine them and return the result.

2.3 Grahm Scan

Overview

Grahm scan finds the convex hull of a set of points by first sorting those points by the angle which they are related to some extreme point (such as the point with lowest y-coordinate). Once the sort has been accomplished, we traverse the sorted list of points counter-clockwise, adding the points to the convex hull which do not cause clockwise ”movements”.

Details

There are two main phases to this algorithm, *1) sorting the points*, and *2) searching those sorted points* for convex hull points. The unique aspect of Grahm Scan is that it sorts the points according to its angle relative to a base point. In our implementation, the base point is the point in our set of points which has the lowest y-coordinate. Finding this point is a simple matter of traversing our list of points and performing a comparison on each element, so takes $\Theta(n)$ time. Next, we traverse the list again, and using this lowest point, we calculate the angle for each point, storing the results in a new list which also contains the angle for each element (see Listing 5).

Listing 5: Finding the Angles

```
...
    # adapted from http://www.algomation.com/
    #                 algorithm/graham-scan-convex-hull
    points_with_angles = get_angles(bottom_point,
                                    points)
    points_with_angles.sort(key=lambda point: point[1])
    sorted_points = []
    for point in points_with_angles:
        sorted_points.append(point[0])
...

def get_angles(p0, points):
    angle_list = []

    for point in points:
        angle_list.append((point,
                           get_angle(p0, point)))

    return angle_list

def get_angle(p1, p2):
    return cartesian_angle(float(p2[0]) - float(p1[0]),
                           float(p2[1]) - float(p1[1]))
```

After we have calculated the angles, we sort this new list with an efficient sorting algorithm (*Timsort* in our case), by its angle component, which takes $\Theta(n\, log\, n)$ time in the worst case. Next, we start adding points to the convex hull from our list of points until we find a point which is positioned either clockwise relative to the previous two suspected convex hull points, or on the same line as them, at which time we remove all intermediate points between the current point and the last suspected convex hull point until we once again have a counter-clockwise relationship between them; this step takes $\Theta(n)$ time and is accomplished as follows:

Listing 6: Finding the Hull Points

```
    for point in sorted_points:
        while len(convex_hull) >= 2 and
              determinant(convex_hull[-2],
              convex_hull[-1], point) <= 0:
```

```
            convex_hull.pop()

        convex_hull.append(point)
```

2.4 Monotone Chain

Overview

Monotone Chain is a simple and easy to implement algorithm for finding the convex hull of a set of points. It can be seen as a variation on Grahm Scan, where instead of sorting the points by their angle relative to some base point, they are sorted lexiographically, and then traversed once forward and once backward counter-clockwise.

Details

We first sort the points lexiographically, first by x-, and then by y-coordinates (if two points have a common x-coordinate, we want to examine the lowest position first). As mentioned previously, adding points to the hull is a two phase procedure, we traverse the points in an increasing (”rightward”) direction, adding points to the lower hull, and then in decreasing (”leftward”) direction, adding points to the upper hull. So, starting from the lowest x-coordinate, we work our way right, and add points to the hull using the same 3-coordinate hull analysis in the Grahm Scan algorithm. We then either sort the list in reverse order or start decrementing our index and work toward the left, and use the same procedure to add points to a separate upper hull. Once this traversal is finished, we combine the two lists, first removing the first and last points of the upper hull, as they are already contained in the lower.

3 Experimental Procedure

To gather emperical data, four different random point generators were used:

1. Points with x and y coordinates between 1 and 10,000.

2. Points with x and y coordinates skewed between 9,000 and 10,000.

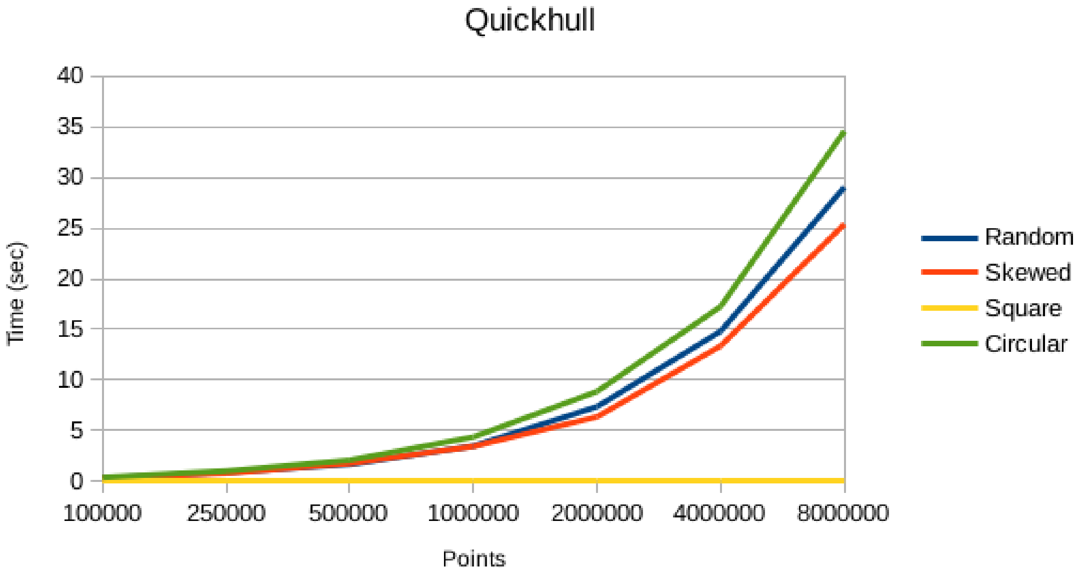
3. Points with x and y coordinates between 1 and 10,000, and points in the four extreme corners (creates a square).

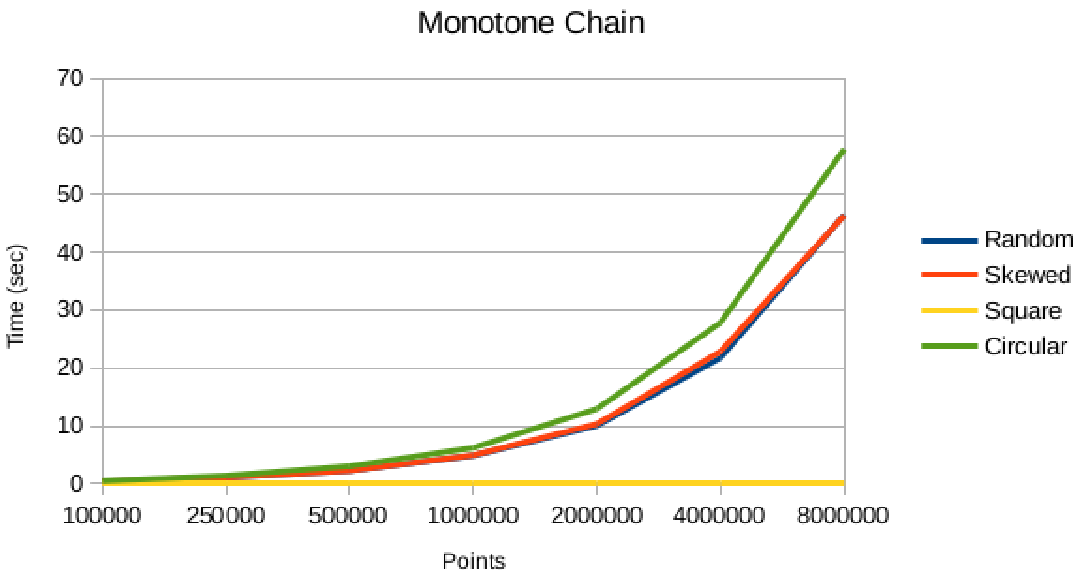
4. Points with x and y coordinates between 1 and 10,000, and a circular distribution shape (maximizes hull points).

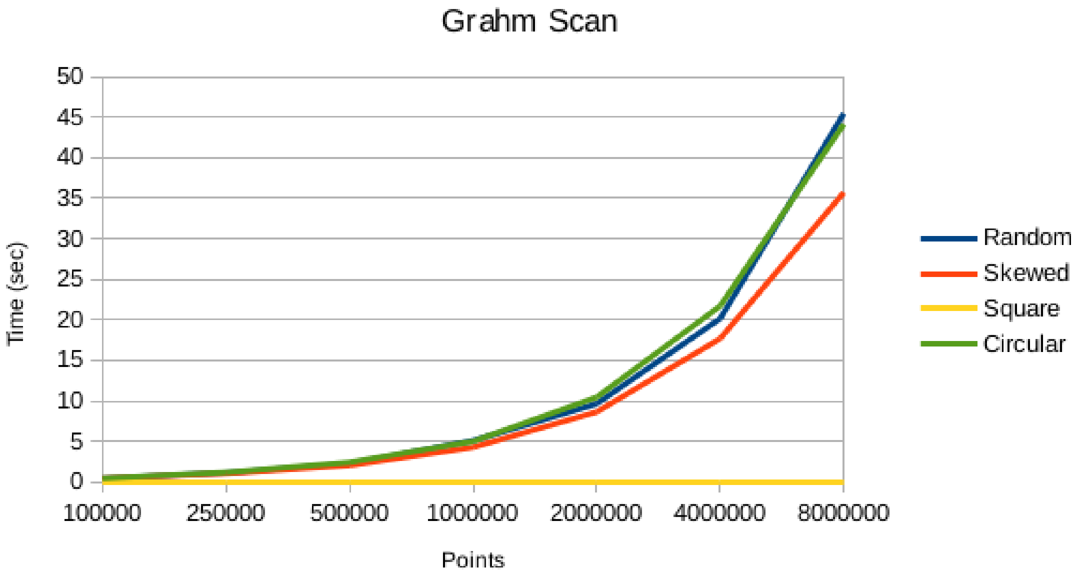
The algorithms were run on the random inputs five times for each point count and for each generator type to ensure that the results were representative and to help eliminate any ”noise” caused by the operating system sharing resources. The number of points were increased incrementally from 100,000 up to 8,000,000; with the input size approximately doubling at each increment. Time to complete was calculated by recording the difference in time before and after each algorithm (input generation was not included in the execution time). The specifications of the computer performing the tests were as follows: OS: Debian GNU/Linux 8, CPU: Intel(R) Core(TM) i5-2540M @ 2.60GHz, RAM: 4096 MB DDR3 1333 MHz (2.3GB free during tests).

4 Experimental Results

Graphs of the results follow, tables of data were omitted for brevity.




Monotone Chain
70
60
50
40
30
20
10
0
Time (sec)
100000
250000
500000
1000000
2000000
4000000
8000000
Points
Random
Skewed
Square
Circular


Grahm Scan
50
45
40
35
30
25
20
15
10
5
0
Time (sec)
100000
250000
500000
1000000
2000000
4000000
8000000
Points
Random
Skewed
Square
Circular

5 Conclusions

The average time complexity for each algorithm is $O(n\, log\, n)$, and the results are consistent with that efficiency class. The results show us that all algorithms process a square distribution in constant time, but the algorithms have different characteristics when the distributions follow a particular shape. Quickhull at its worst, performs better than the iterative algorithms, even at their best. It performs best under a skewed dataset, but worst with a circular one. Quickhull was noticeably more difficult to implement, but compared to the other two iterative algorithms, divide and conquer has a clear speed advantage. Our iterative algorithms perform nearly the same with an unshaped random distribution, but Monotone Chain, the easier to implement, has a clear disadvantage with shaped datasets. As mentioned, Grahm Scan out performs Monotone Chain for shaped data, but falls short of Quickhull; interestingly, it performs slightly better on a circular dataset than it does on an unshaped random dataset; sorting points by angle appears to provide an advantage here.