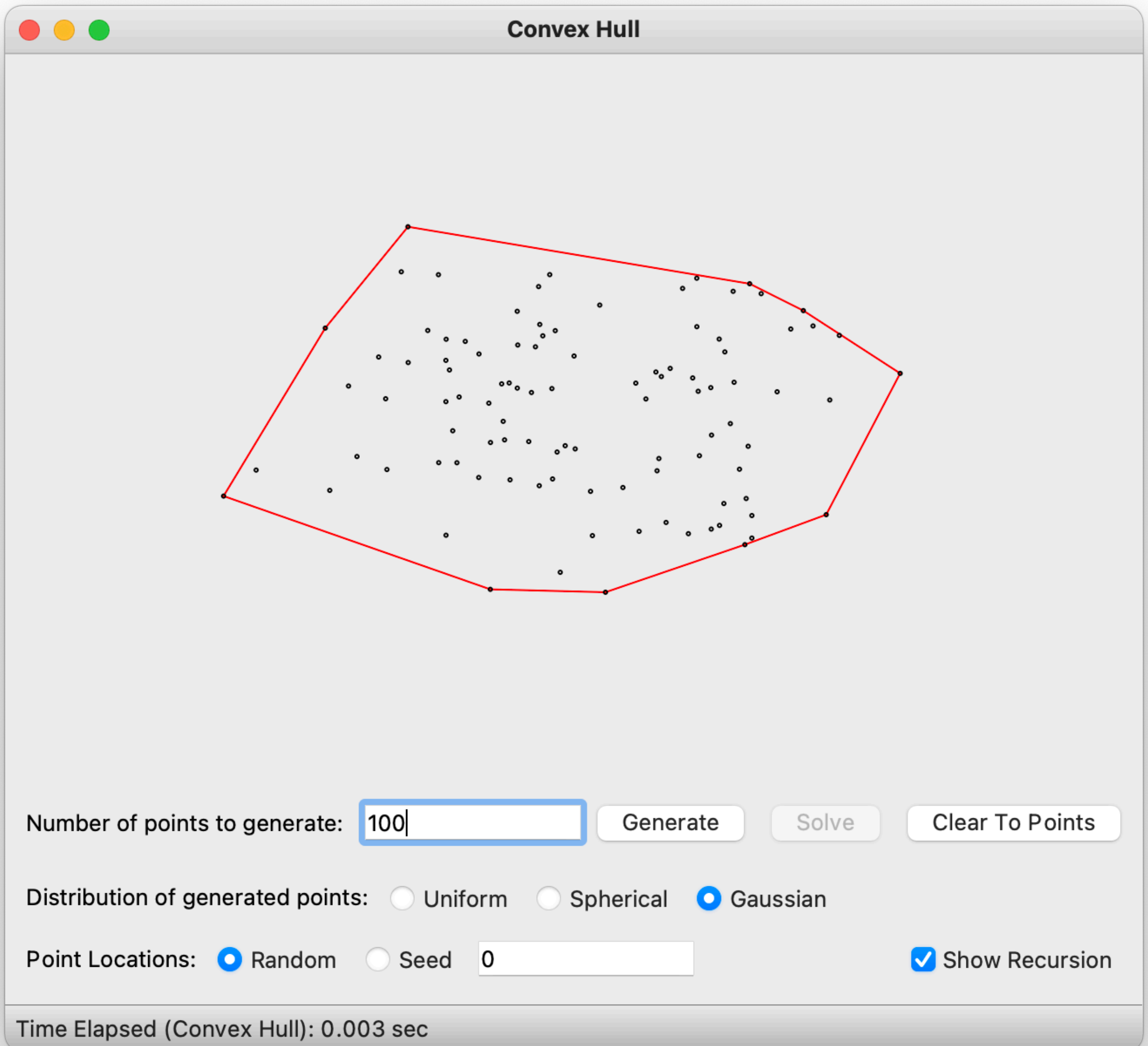
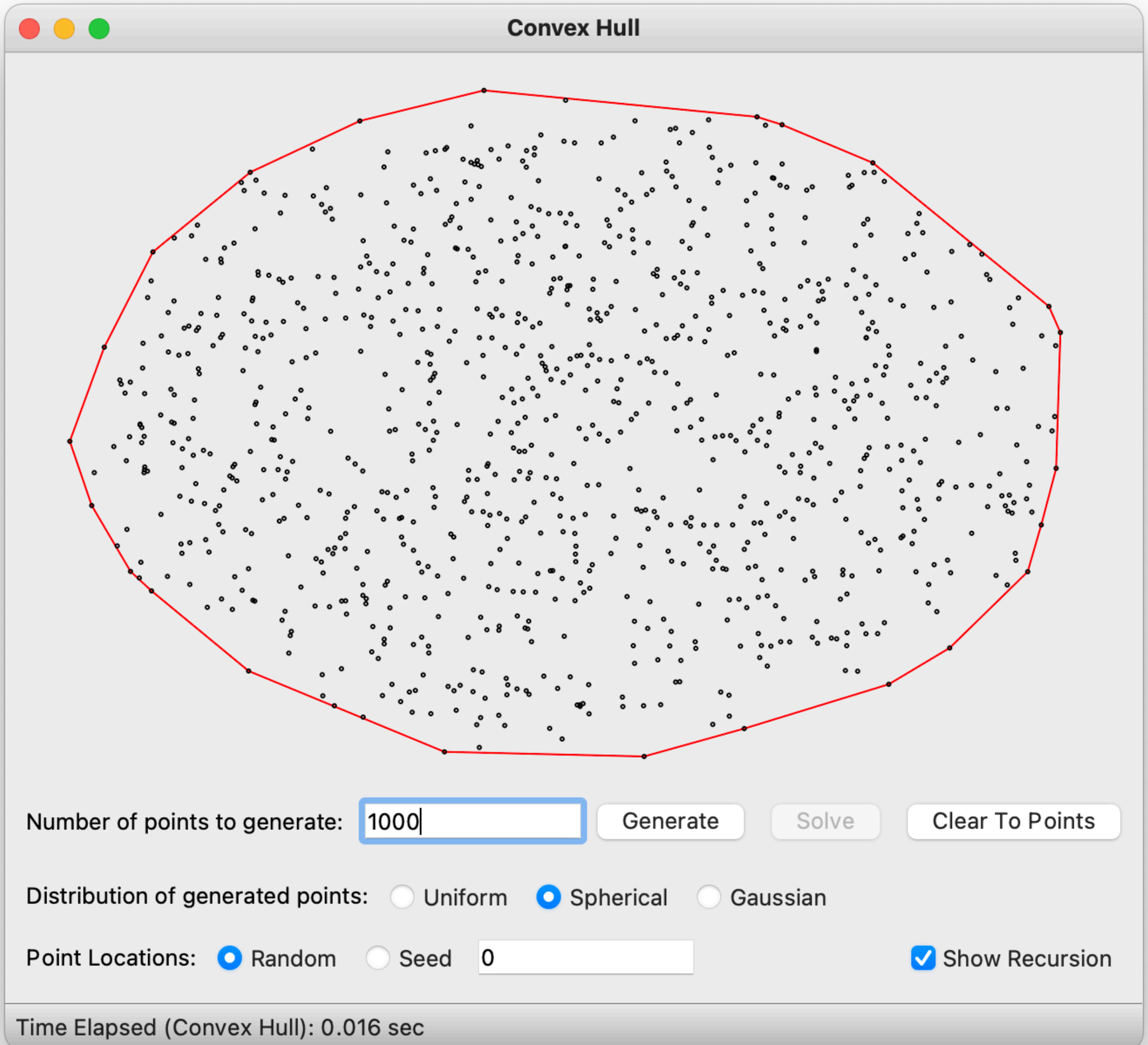
Project 2: Convex Hull

Noah Schill Fall 2021 CS312





Pseudocode

Write the full, unambiguous pseudo-code for your divide-and-conquer algorithm for finding the convex hull of a set of points Q. Be sure to label the parts of your algorithm. Also, label each part with its worst-case time efficiency.

```
function convex_hull(Q):
  hull_points = self.hull_helper(Q)
  lines = []
  for index in range(len(hull_points)-1):  // takes O(points) time in worst case
    line = new line(point[index], point[index+1])
    lines.append(line)
  lines.append(new line(hull_points[-1], hull_points[0]))
```

```
function hull_helper(points):  // T(n) = 2(n/2)+O(n^1) -> O(n*log(n)). Space complexity:
O(n)
  if len(points) <= 2:  // Base case, 2 points only
    return points sorted by x-value

  //Divide points into l, r and run the helper on them
  l_points, r_points = points[0...k//2], points[k//2...k]
  l_hull = hull_helper(l_points)
  r_hull = self.hull_helper(r_points)

  return merge(l_hull, r_hull)

function merge(l_hull, r_hull):  // Time: O(n), Space: O(n)
  upper_tangent = upper_tangent(l_hull, r_hull)
  lower_tangent = lower_tangent(l_hull, r_hull)

  final_hull = []

  P = first element in l_hull
  while P is not the left upper tangent:
    add P to final_hull
    increment P to next counter-clockwise element in l_hull
  add left upper tangent to final_hull

  P = right upper tangent in r_hull
  while P is not right lower tangent:
    add P to final_hull
    increment P to next clockwise element in r_hull
  add lower left tangent to final_hull

  P = lower left tangent in l_hull
  while P is not the 0th element of l_hull:
    if P isn't already in final_hull:
      add P to final_hull
    increment P to next clockwise point in l_points

  return final_hull

function upper_tangent(l_hull, r_hull):  // Time: O(n), Space: O(1)
  p = rightmost point in l_hull
  q = leftmost point in r_hull
  found_tangent = false
  current_slope = slope(p, q)

  while not found_tangent:
```

```
    found_tangent = true
    next_p = next counter-clockwise point in l_hull
    while current_slope > slope(next_p, q):
      p = next_p
      current_slope = slope (p, q)
      found_tangent = false

    next_q = next clockwise point in r_hull:
    while current_slope < slope(p, next_q):
      q = next_q
      current_sloppe = slope(p, q)
      found = false

  return p, q

function lower_tangent(l_hull, r_hull):  // Time: O(n), Space: O(1)
  p = rightmost point in l_hull
  q = leftmost point in r_hull
  found_tangent = false
  current_slope = slope(p, q)

  while not found_tangent:
    found_tangent = true
    next_p = next clockwise point in l_hull
    while current_slope < slope(next_p, q):
      p = next_p
      current_slope = slope (p, q)
      found_tangent = false

    next_q = next counter-clockwise point in r_hull:
    while current_slope > slope(p, next_q):
      q = next_q
      current_sloppe = slope(p, q)
      found = false

  return p, q

function slope(p1, p2):  // O(1)
  return p2.y - p1.y / p2.x - p1.x
```

Time-complexity Analysis

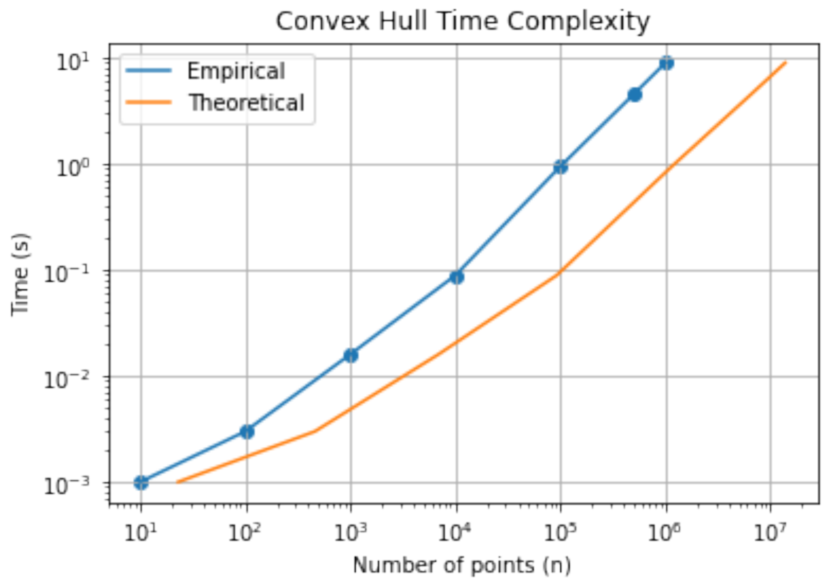
Analyze the whole algorithm for its worst-case time efficiency. State the Big-O asymptotic bound. Discuss how this relates to the Master Theorem estimate for runtime.

The Big-O asymptotic bound is $O(n \log n)$ for n points. This is because in the helper function, there is a recursive call with a branching factor of $a = 2$ cut into $n/2$ pieces for the left and right sub-hulls. At each stack frame, it takes $O(n)$ to find find the tangents and merge the sub-hulls together ($O(n)$ at each step). By master theorem:

$$\begin{aligned} T(n) &= 2T(\frac{n}{2}) + O(n^1) \\ &= O(n \log n) \end{aligned}$$

Empirical analysis

	n	time
0	10	0.001
1	100	0.003
2	1000	0.016
3	10000	0.089
4	100000	0.950
5	500000	4.564
6	1000000	9.019



Given the graph, the empirical function and the theoretical functions mirror each other closely in shape. This graph has a logarithmic x and y axis. I calculated the constant of proportionality to be:

$$k = 7.91428439802643\mathrm{e}{-06}$$

via the following code:

```
sum = 0
for i in range(len(time)):
    sum += (time[i]/logfunc[i])
k = sum/len(time)
```

where logfunc is given by `logfunc = np.log(df["n"].to_numpy())*df["n"].to_numpy()`.

Code:

```
# This is the method that gets called by the GUI and actually executes
    # the finding of the hull
    def compute_hull(self, points, pause, view):
        self.pause = pause
        self.view = view
        assert (type(points) == list and type(points[0]) == QPointF)

        t1 = time.time()
        # TODO: SORT THE POINTS BY INCREASING X-VALUE
        sorted_points = sort_points(points)
        t2 = time.time()

        t3 = time.time()
        # this is a dummy polygon of the first 3 unsorted points
```

```
        # polygon = [QLineF(points[i],points[(i+1)%3]) for i in range(3)]
        lines = self.convex_hull(points)
        # TODO: REPLACE THE LINE ABOVE WITH A CALL TO YOUR DIVIDE-AND-CONQUER CONVEX HULL
SOLVER
        t4 = time.time()
        # self.showHull(lines, RED)
        # when passing lines to the display, pass a list of QLineF objects.  Each QLineF
        # object can be created with two QPointF objects corresponding to the endpoints
        self.showHull(lines, RED)
        self.showText('Time Elapsed (Convex Hull): {:3.3f} sec'.format(t4 - t3))

    def convex_hull(self, points):
        hull_points = self.help_a_hull_out(points)
        lines = []
        for i in range(len(hull_points) - 1):
            line = QLineF(hull_points[i], hull_points[i + 1])
            lines.append(line)
        lines.append(QLineF(hull_points[-1], hull_points[0]))

        return lines

    def help_a_hull_out(self, points):
        # Base case: 2 points
        if len(points) <= 2:
            return sorted(points, key=lambda x:x.x())

        # Divide points into L and R
        l_points, r_points = points[0:len(points) // 2], points[len(points) // 2:]
        l_hull = self.help_a_hull_out(l_points)
        r_hull = self.help_a_hull_out(r_points)

        return self.merge(l_hull, r_hull)

    def merge(self, l_hull: [], r_hull: []):  # Accepts two hulls as lists of points
        upper_tangent_left, upper_tangent_right = self.upper_tangent(l_hull, r_hull)
        lower_tangent_left, lower_tangent_right = self.lower_tangent(l_hull, r_hull)

        return self.one_with_everything(l_hull, r_hull, upper_tangent_left,
upper_tangent_right, lower_tangent_left,
                                        lower_tangent_right)

    def one_with_everything(self, l_hull, r_hull, upper_tangent_left, upper_tangent_right,
lower_tangent_left,
                            lower_tangent_right):
        the_whole_hull_nothing_but_the_hull = []
```

```
        # The left upper half
        ant_place = 0
        while l_hull[ant_place] != upper_tangent_left:
            the_whole_hull_nothing_but_the_hull.append(l_hull[ant_place])
            ant_place += 1
        the_whole_hull_nothing_but_the_hull.append(upper_tangent_left)

        # the right upper half!
        ant_place = r_hull.index(upper_tangent_right)
        while r_hull[ant_place] != lower_tangent_right:
            the_whole_hull_nothing_but_the_hull.append(r_hull[ant_place])
            ant_place = (ant_place + 1) % len(r_hull)
        the_whole_hull_nothing_but_the_hull.append(lower_tangent_right)

        # the left lower half
        ant_place = l_hull.index(lower_tangent_left)
        while ant_place != 0:
            if l_hull[ant_place] not in the_whole_hull_nothing_but_the_hull:
                the_whole_hull_nothing_but_the_hull.append(l_hull[ant_place])
            ant_place = (ant_place + 1) % len(l_hull)

        return the_whole_hull_nothing_but_the_hull

    def upper_tangent(self, l, r):
        p = max(l, key=lambda x: x.x())  # Rightmost point in l
        q = r[0]  # Leftmost point in r
        found = False
        current_slope = slope(p, q)

        while not found:
            found = True
            next_p = l[(l.index(p) - 1) % len(l)]
            while current_slope > slope(next_p, q):  # Go CCW around l
                p = next_p
                current_slope = slope(p, q)
                found = False

            next_q = r[(r.index(q) + 1) % len(r)]
            while current_slope < slope(p, next_q):
                q = next_q
                current_slope = slope(p, q)
                found = False

        return p, q

    def lower_tangent(self, l, r):
        p = max(l, key=lambda x: x.x())  # Rightmost point in l
```

```python
    p = max(l, key=lambda x: x.x())  # Rightmost point in l
    q = r[0]  # Leftmost point in r
    found = False
    current_slope = slope(p, q)

    while not found:
        found = True
        next_p = l[(l.index(p) + 1) % len(l)]
        while current_slope < slope(next_p, q):  # Go CCW around l
            p = next_p
            current_slope = slope(p, q)
            found = False

        next_q = r[(r.index(q) - 1) % len(r)]
        while current_slope > slope(p, next_q):
            q = next_q
            current_slope = slope(p, q)
            found = False

    return p, q
```