Convex Hull Report

Niels Turley

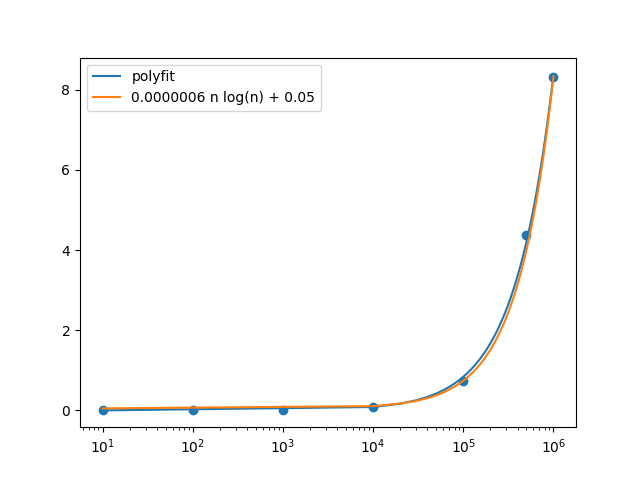
1. **Correct functioning code**
2. from which\_pyqt import PYQT\_VER
3. if PYQT\_VER == 'PYQT5':
4. from PyQt5.QtCore import QLineF, QPointF, QObject
5. elif PYQT\_VER == 'PYQT4':
6. from PyQt4.QtCore import QLineF, QPointF, QObject
7. elif PYQT\_VER == 'PYQT6':
8. from PyQt6.QtCore import QLineF, QPointF, QObject
9. else:
10. raise Exception('Unsupported Version of PyQt: {}'.format(PYQT\_VER))
11. import time
12. import math
13. # Some global color constants that might be useful
14. RED = (255, 0, 0)
15. GREEN = (0, 255, 0)
16. BLUE = (0, 0, 255)
17. # Global variable that controls the speed of the recursion automation, in seconds
18. #
19. PAUSE = 0.1
20. #
21. # This is the class you have to complete.
22. #
23. class ConvexHullSolver(QObject):
24. # Class constructor
25. def \_\_init\_\_(self):
26. super().\_\_init\_\_()
27. self.pause = False
28. # Some helper methods that make calls to the GUI, allowing us to send updates
29. # to be displayed.
30. def showTangent(self, line, color):
31. lines = [line]
32. self.view.addLines(lines, color)
33. if self.pause:
34. time.sleep(PAUSE)
35. def eraseTangent(self, line):
36. self.view.clearLines([line])
37. def blinkTangent(self, line, color):
38. self.showTangent(line, color)
39. self.eraseTangent(line)
40. def showHull(self, polygon, color):
41. self.view.addLines(polygon, color)
42. if self.pause:
43. time.sleep(PAUSE)
44. def eraseHull(self, polygon):
45. self.view.clearLines(polygon)
46. def showText(self, text):
47. self.view.displayStatusText(text)
48. # This is the method that gets called by the GUI and actually executes
49. # the finding of the hull
50. def compute\_hull(self, points, pause, view):
51. self.pause = pause
52. self.view = view
53. assert (type(points) == list and type(points[0]) == QPointF)
54. t1 = time.time()
55. points.sort(key=lambda p: p.x()) # sort function for list in python is O(n log n)
56. t2 = time.time()
57. t3 = time.time()
58. hull\_points = self.create\_hull(points)
59. t4 = time.time()
60. # when passing lines to the display, pass a list of QLineF objects. Each QLineF
61. # object can be created with two QPointF objects corresponding to the endpoints
62. polygon = [QLineF(hull\_points[i], hull\_points[(i + 1) % len(hull\_points)]) for i in range(len(hull\_points))]
63. self.showHull(polygon, RED)
64. self.showText('Time Elapsed (Convex Hull): {:3.3f} sec'.format(t4 - t1))
65. # Time complexity: Divide and conquer, so use master theorem. Dividing into two subproblems (a = 2),
66. # each has half the size (b = 2), merging them back together in linear time (d = 1). 2 / (2^1) = 1, giving us
67. # O(n log n) time
68. # Space complexity: O(n). The recursion tree is iterating down one branch of the tree and you aren’t creating any
69. # new lists at each node. The max you could ever have at once is n memory used.
70. def create\_hull(self, points):
71. # recurse down, splitting the points in half, till you have 2 or 3 point elements
72. if len(points) > 3:
73. middle = math.floor(len(points) / 2)
74. l\_hull = self.create\_hull(points[:middle])
75. r\_hull = self.create\_hull(points[middle:])
76. elif len(points) == 3:
77. # if it's 3 points, sort them in clockwise order
78. return self.sort\_base\_case\_clockwise(points)
79. else:
80. # else it is 2 points, and there is no need to sort because we already sorted L to R
81. return points
82. # polygon\_l = [QLineF(l\_hull[i], l\_hull[(i + 1) % len(l\_hull)]) for i in range(len(l\_hull))]
83. # self.showHull(polygon\_l, RED)
84. # polygon\_r = [QLineF(r\_hull[i], r\_hull[(i + 1) % len(r\_hull)]) for i in range(len(r\_hull))]
85. # self.showHull(polygon\_r, RED)
86. # find the R most index in l\_hull and the L most index in r\_hull (which is just the first one)
87. current\_index\_l = self.find\_right\_most\_index(l\_hull)
88. current\_index\_r = 0
89. # call the helper functions to find the upper bounds and lower bounds, then save them as their point indexes
90. up\_left\_index, up\_right\_index = self.find\_upper\_bound(l\_hull, r\_hull, current\_index\_l, current\_index\_r)
91. down\_left\_index, down\_right\_index = self.find\_lower\_bound(l\_hull, r\_hull, current\_index\_l, current\_index\_r)
92. # create the hull that will be returned. this is done by circling clockwise around the edges of the hull,
93. # dropping the points that are in the middle using the indexes
94. # 0 --> up\_left\_index --> up\_right\_index --> down\_right\_index --> down\_left\_index --> back to 0
95. hull\_points = []
96. hull\_points.extend(l\_hull[:up\_left\_index + 1])
97. if down\_right\_index != 0:
98. hull\_points.extend(r\_hull[up\_right\_index:down\_right\_index + 1])
99. else:
100. hull\_points.extend(r\_hull[up\_right\_index:])
101. hull\_points.append(r\_hull[0])
102. if down\_left\_index != 0:
103. hull\_points.extend(l\_hull[down\_left\_index:])
104. # self.eraseHull(polygon\_l)
105. # self.eraseHull(polygon\_r)
106. # polygon = [QLineF(hull\_points[i], hull\_points[(i + 1) % len(hull\_points)]) for i in range(len(hull\_points))]
107. # self.showHull(polygon, GREEN)
108. return hull\_points
109. @staticmethod
110. def slope(x1, y1, x2, y2):
111. # returns the slope given the x / y values of two points
112. return (y2 - y1) / (x2 - x1)
113. def sort\_base\_case\_clockwise(self, points):
114. # 3 points need to be sorted in clockwise order
115. sorted\_points = [points[0]]
116. slope\_1 = self.slope(QPointF.x(points[0]),
117. QPointF.y(points[0]),
118. QPointF.x(points[1]),
119. QPointF.y(points[1]))
120. slope\_2 = self.slope(QPointF.x(points[0]),
121. QPointF.y(points[0]),
122. QPointF.x(points[2]),
123. QPointF.y(points[2]))
124. # compare the slopes between the first and second/first and third
125. if slope\_1 > slope\_2:
126. sorted\_points.extend([points[1], points[2]])
127. else:
128. sorted\_points.extend([points[2], points[1]])
129. return sorted\_points
130. # Time complexity: worst case scenario would be iterating through the whole hull, so O(n)
131. # Space complexity: only storing the list of points, so O(n)
132. @staticmethod
133. def find\_right\_most\_index(l\_hull):
134. right\_most\_index = 0
135. # iterate through the points in the l\_hull--if the pt's x is bigger, keep going. else return the index.
136. for index, pt in enumerate(l\_hull):
137. if index == 0:
138. continue
139. if QPointF.x(pt) > QPointF.x(l\_hull[right\_most\_index]):
140. right\_most\_index = index
141. else:
142. return right\_most\_index
143. return right\_most\_index
144. # Time complexity: the nested while loops are the part that will take the longest; however, the inner while loops
145. # will maximally terminate after looping through all the points (the slope must fail the check if it loops back
146. # to the start of the hull). So the inner loops are both O(n), and the outer loop terminates if neither inner
147. # loop changes, meaning that this is O(n + n) --> O(n).
148. # Space complexity: the only thing stored is both lists of hulls, so O(n)
149. def find\_upper\_bound(self, l\_hull, r\_hull, current\_index\_l, current\_index\_r):
150. upper\_bound\_slope = self.slope(QPointF.x(l\_hull[current\_index\_l]),
151. QPointF.y(l\_hull[current\_index\_l]),
152. QPointF.x(r\_hull[current\_index\_r]),
153. QPointF.y(r\_hull[current\_index\_r]))
154. # line = QLineF(l\_hull[current\_index\_l], r\_hull[current\_index\_r])
155. # self.blinkTangent(line, BLUE)
156. # nested while loop--outer loop continues if there has been a change in the upper\_bound\_slope, inner loops move
157. # the indexes of the l\_hull and r\_hull around to find the best slope, respectively.
158. change = True
159. while change:
160. change = False
161. while True:
162. check\_index\_l = (current\_index\_l - 1) % len(l\_hull)
163. check\_slope = self.slope(QPointF.x(l\_hull[check\_index\_l]),
164. QPointF.y(l\_hull[check\_index\_l]),
165. QPointF.x(r\_hull[current\_index\_r]),
166. QPointF.y(r\_hull[current\_index\_r]))
167. # line = QLineF(l\_hull[check\_index\_l], r\_hull[current\_index\_r])
168. # self.blinkTangent(line, BLUE)
169. if check\_slope > upper\_bound\_slope:
170. # end the loop--we've found the best slope for the current\_index\_r
171. break
172. else:
173. # keep going--save the check variables into the current variables and try again
174. change = True
175. upper\_bound\_slope = check\_slope
176. current\_index\_l = check\_index\_l
177. while True:
178. check\_index\_r = (current\_index\_r + 1) % len(r\_hull)
179. check\_slope = self.slope(QPointF.x(l\_hull[current\_index\_l]),
180. QPointF.y(l\_hull[current\_index\_l]),
181. QPointF.x(r\_hull[check\_index\_r]),
182. QPointF.y(r\_hull[check\_index\_r]))
183. # line = QLineF(l\_hull[current\_index\_l], r\_hull[check\_index\_r])
184. # self.blinkTangent(line, BLUE)
185. if check\_slope < upper\_bound\_slope:
186. # end the loop--we've found the best slope for the current\_index\_l
187. break
188. else:
189. # keep going--save the check variables into the current variables and try again
190. change = True
191. upper\_bound\_slope = check\_slope
192. current\_index\_r = check\_index\_r
193. # line = QLineF(l\_hull[current\_index\_l], r\_hull[current\_index\_r])
194. # self.blinkTangent(line, GREEN)
195. # self.blinkTangent(line, GREEN)
196. return current\_index\_l, current\_index\_r
197. # Time complexity: the nested while loops are the part that will take the longest; however, the inner while loops
198. # will maximally terminate after looping through all the points (the slope must fail the check if it loops back
199. # to the start of the hull). So the inner loops are both O(n), and the outer loop terminates if neither inner
200. # loop changes, meaning that this is O(n + n) --> O(n).
201. # Space complexity: the only thing stored is both lists of hulls, so O(n)
202. def find\_lower\_bound(self, l\_hull, r\_hull, current\_index\_l, current\_index\_r):
203. lower\_bound\_slope = self.slope(QPointF.x(l\_hull[current\_index\_l]),
204. QPointF.y(l\_hull[current\_index\_l]),
205. QPointF.x(r\_hull[current\_index\_r]),
206. QPointF.y(r\_hull[current\_index\_r]))
207. # line = QLineF(l\_hull[current\_index\_l], r\_hull[current\_index\_r])
208. # self.blinkTangent(line, BLUE)
209. # double while loop--outer loop continues if there has been a change in the upper\_bound\_slope, inner loops move
210. # the indexes of the l\_hull and r\_hull around to find the best slope, respectively.
211. change = True
212. while change:
213. change = False
214. while True:
215. check\_index\_l = (current\_index\_l + 1) % len(l\_hull)
216. check\_slope = self.slope(QPointF.x(l\_hull[check\_index\_l]),
217. QPointF.y(l\_hull[check\_index\_l]),
218. QPointF.x(r\_hull[current\_index\_r]),
219. QPointF.y(r\_hull[current\_index\_r]))
220. # line = QLineF(l\_hull[check\_index\_l], r\_hull[current\_index\_r])
221. # self.blinkTangent(line, BLUE)
222. if check\_slope < lower\_bound\_slope:
223. # end the loop--we've found the best slope for the current\_index\_r
224. break
225. else:
226. # keep going--save the check variables into the current variables and try again
227. change = True
228. lower\_bound\_slope = check\_slope
229. current\_index\_l = check\_index\_l
230. while True:
231. check\_index\_r = (current\_index\_r - 1) % len(r\_hull)
232. check\_slope = self.slope(QPointF.x(l\_hull[current\_index\_l]),
233. QPointF.y(l\_hull[current\_index\_l]),
234. QPointF.x(r\_hull[check\_index\_r]),
235. QPointF.y(r\_hull[check\_index\_r]))
236. # line = QLineF(l\_hull[current\_index\_l], r\_hull[check\_index\_r])
237. # self.blinkTangent(line, BLUE)
238. if check\_slope > lower\_bound\_slope:
239. # end the loop--we've found the best slope for the current\_index\_l
240. break
241. else:
242. # keep going--save the check variables into the current variables and try again
243. change = True
244. lower\_bound\_slope = check\_slope
245. current\_index\_r = check\_index\_r
246. # line = QLineF(l\_hull[current\_index\_l], r\_hull[current\_index\_r])
247. # self.blinkTangent(line, GREEN)
248. # self.blinkTangent(line, GREEN)
249. return current\_index\_l, current\_index\_r
250. **Time and space complexity**
     1. Create\_hull()
        1. Time complexity: O(n log n). Divide and conquer, so use master theorem. The recursion is dividing into two subproblems with a left hull and a right hull (a = 2), each with half the size (b = 2), and merging them back together in linear time (d = 1).

So by the master theorem we get O (nd log n) or O(n log n). My empirical data also supports this answer.

* + 1. Space complexity: O(n). We are iterating down one branch of the recursion tree at a time. Since we aren’t creating any new lists or matrices in each recursion, only passing back and combining the points in the hull, the max space we’d ever use is just n, or the total number of points we already have. The other functions create\_hull calls are also dominated by O(n).
  1. find\_right\_most\_index()
     1. Time complexity: O(n). The worst case scenario would be iterating through the entire hull to find the last point, so the time complexity is O(n).
     2. Space complexity: O(n). The only thing stored is the list of points we have to iterate though, which is O(n). We aren’t creating any new large variables.
  2. Find\_upper\_bound() and found\_lower\_bound()
     1. Time complexity: O(n). The nested while loops are the part that will take the longest; however, the inner while loops will maximally terminate after looping through all the points (the check\_slope must necessarily fail the check if it loops back to the start of the hull). So the inner loops are both O(n), and the outer loop terminates if neither inner loop changes, meaning that this is O(n + n) --> O(n).
     2. Space complexity: O(n): O(n). The only thing stored is the list of points we have to iterate though, which is O(n). We aren’t creating any new large variables.

1. **Experimental outcomes**

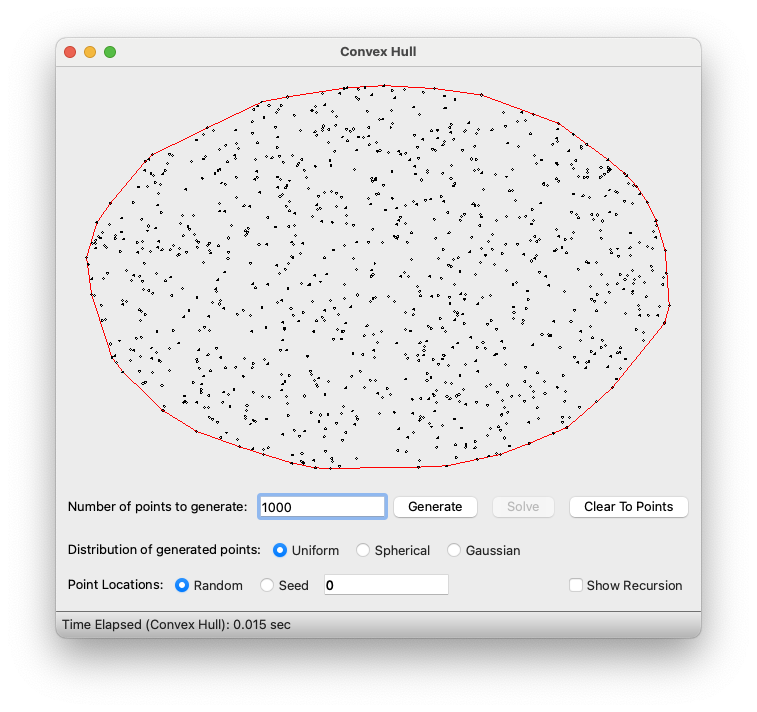
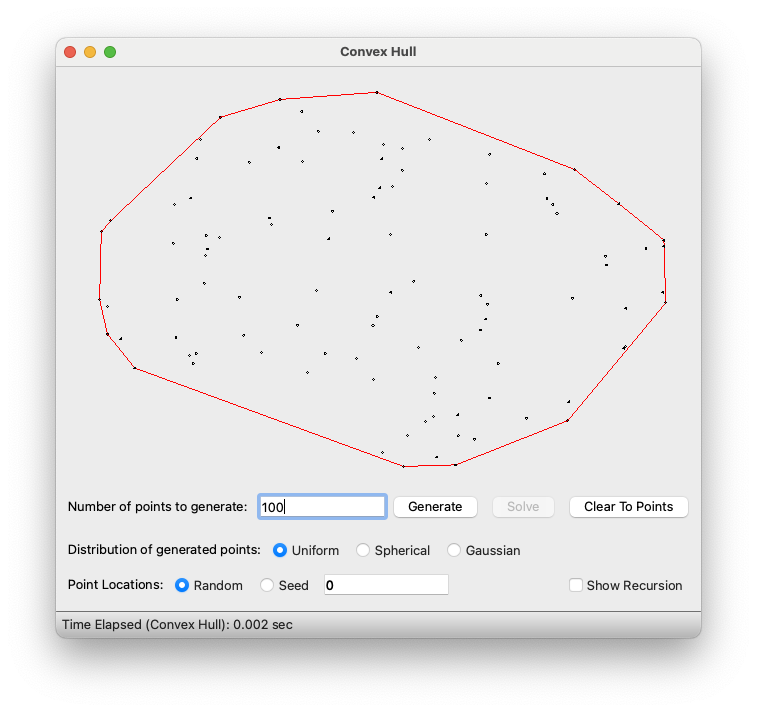
Using uniform distribution  
  
n = 10, mean = 0.000  
0.001  
0.000  
0.000  
0.000  
0.000  
  
n = 100, mean = 0.002  
0.002  
0.002  
0.002  
0.002  
0.002  
  
n = 1,000, mean = 0.014  
0.015  
0.014  
0.014  
0.013  
0.014  
  
n = 10,000, mean = 0.0828  
0.080  
0.083  
0.081  
0.081  
0.089  
  
n = 100,000, mean = 0.7324  
0.721  
0.685  
0.820  
0.714  
0.722  
  
n = 500,000, mean = 4.3648  
4.639  
4.059  
4.754  
4.073  
4.299  
  
n = 1,000,000, mean = 8.308  
8.332  
8.333  
8.254  
8.276  
8.345



My empirical analysis matches the predicted O(n log n). Using a logarithmic scale on the x axis (which allowed me to visualize all the points by gradually increasing the x-axis step size), I was guessed and checked different constants of proportionality to match the n log n line to my points. I ended up using the constant of proportionality of 0.0000006 (and adding 0.05, but that was trivial).

1. **Discussion**

It’s gratifying that my empirical analysis matched the theoretical analysis predicted using the Master Theorem—I was slightly terrified it wouldn’t. Several of my friends in this class found that their runtimes were much longer than mine (one had 20 seconds on n=1000000, the other had 100 seconds on n=1000000). The graphs all retained the same “shape,” however, seeming to indicate that we each were utilizing an O(n log n) algorithm. The differences could be due to differing processor speeds, or perhaps unoptimized code (not breaking out of unnecessary loops, unnecessary function calls, etc.). But in the end, our runtimes were in the same complexity class, which is the most important for scalability.

1. **Correct screenshots**