Project 2 Report

[40] Correct functioning code to solve the Convex Hull problem using the divide and conquer scheme discussed above. Include your documented source code.

> *My documented source code is found at the bottom of this document in the section labeled Source Code.*

[15] Explain the time and space complexity of your algorithm by showing and summing up the complexity of each subsection of your code. Also, include your theoretical analysis for the entire algorithm including discussion of the recurrence relation.

Below, I discuss the portions of the algorithm that have the largest time and space complexities, since they have the greatest effect on the algorithm's performance overall. Additional time and space complexities can be found in my documented source code.

Recursion

The recursion in compute_hull_helper() has logn recursions and, since orienting the upper and lower tangents are done in O(n) time, the total time complexity for the recursion calls are O(nlogn). The space complexity is O(n) because there will be O(n) instances of compute_hull_helper() at the bottom of the recursion tree.

Sorting Function

Python's native sorting algorithm, with my additional lambda, carries out a typical quicksort in O(nlogn) time. The algorithm also needs access to the n items in the provided list of QPointFs, and therefore has a space complexity of O(n). Both the sorting function's time and space complexities are tied with the recursion calls for being the most costly complexities of my algorithm.

Orienting Tangents

The final portion of the convex hull algorithm worth mentioning is the orient() function that is called from the constructor of my abstract Tangent class. The function orients tangents upon their instantiation. Both its time and space complexities are O(n). The time complexity is for how the function has the potential for visiting every node in the given hulls. The space complexity is for how the function could possibly access every node in the given hulls. Only the space complexity is tied with the sorting function and recursion calls for being the most costly in terms of space complexity.

[15] Include your raw and mean experimental outcomes, plot, and your discussion of the pattern in your plot. Which order of growth fits best? Give an estimate of the constant of proportionality. Include all work and explain your assumptions.

The Master Theorem

a = 2 because there are two branches created at each level of the recursion
b = 2 because the array of points is divided in half by each recursion
d = 1 because it take O(n) time to orient the tangents to make a join

$T(n) = aT(ceil(n/b)) + O(n^d) = 2T(ceil(n/2)) + O(n^1) = 2T(ceil(n/2)) + O(n)$
$\frac{a}{b^d} = \frac{2}{2^1} = \frac{2}{2} = 1$
Since the geometric ratio is 1, the runtime is $O(n^d logn) = O(n^1 logn) = O(nlogn)$.

Empirical Analysis
Below, I use the g(n) from the master theorem, O(nlogn), to estimate the constant of proportionality (k) with two different methods: a table and a plot. Each method shows a slightly different value for k. I therefore label the first k and the second k'. The difference between the two is negligible, being just over two millionths.

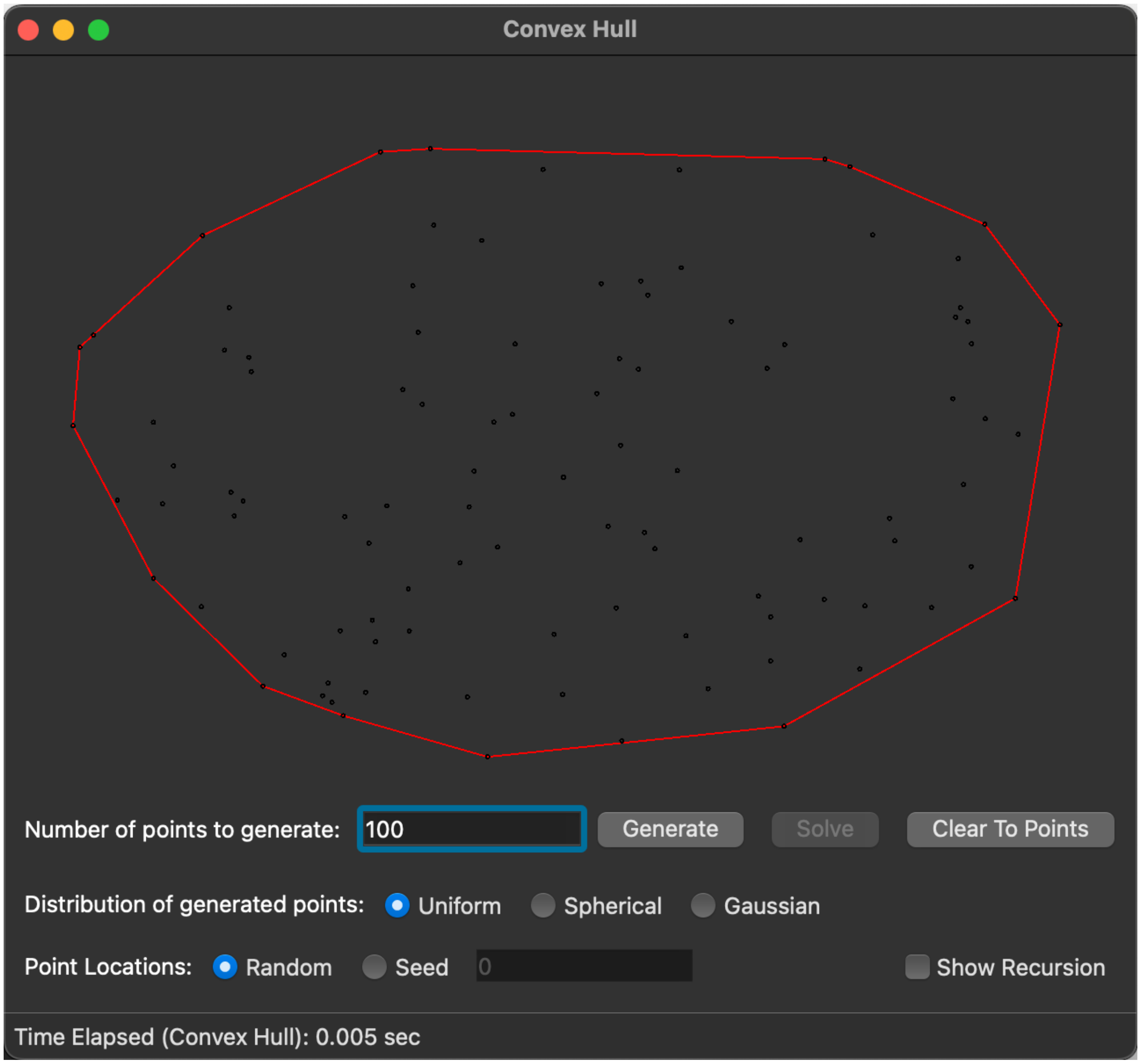
The Table
The table in Figure 2.1 shows my raw and mean experimental outcomes. It also shows what the outcomes would be if the algorithm were perfectly nlogn. It divides the mean empirical values with the nlogn values to find k values for each point count. The average of the k values is shown at the bottom right and is labeled "Mean k".

The Plot
*Figure 2.2 shows a plot of g(n), k*g(n), k'*g(n), and my empirical results, where k is the constant of proportionality found via the table and k' is the constant of proportionality found by estimating the value visually. I estimated k' to be 0.00002 or $\frac{1}{200000}$.*

[10] Discuss and explain your observations with your theoretical and empirical analyses, including any differences seen.
My theoretical analysis concluded that my algorithm would run in O(nlogn) time. It did not, however, account for the processing speed of my computer, the current load on my CPU, or any of the other circumstantial factors that would affect performance. Those factors were accounted for by my empirical analysis though.
Like my theoretical analysis, my empirical analysis shows a runtime of O(nlogn) time. Specifically, the plot shows a curve that resembles the practically parallel O(nlogn) next to it. My empirical analysis also allowed me to discover a surprisingly low constant of proportionality (k), which was about 0.00002.

[10] Include a correct screenshot of an example with 100 points and a screenshot of an example with 1000 points. You can capture the image of the window using the ctrl-alt-shift-PrtSc facility for capturing an image of the window in focus.
See Figures 1.1 and 1.2

Reference Figures



Number of points to generate: 100

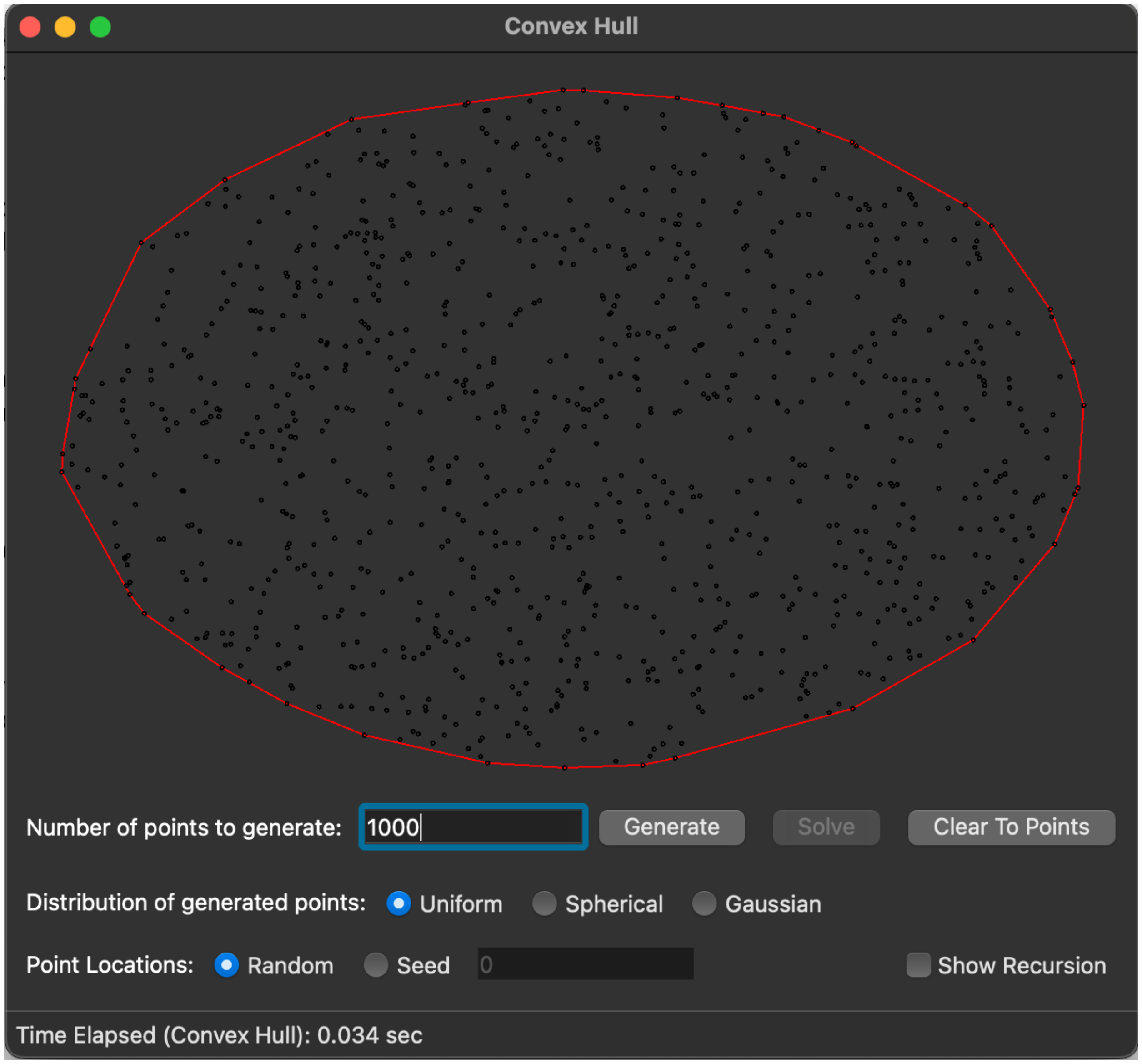
Distribution of generated points: [✓ Uniform] Spherical Gaussian

Point Locations: [✓ Random] Seed 0

Show Recursion

Time Elapsed (Convex Hull): 0.005 sec

Figure 1.1: Empirical test with 100 points.



Number of points to generate: 1000

Distribution of generated points: [✓ Uniform] Spherical Gaussian

Point Locations: [✓ Random] Seed 0 Show Recursion

Time Elapsed (Convex Hull): 0.034 sec

Figure 1.2: Empirical test with 1000 points.

Point Count	**Test 1**	**Test 2**	**Test 3**	**Test 4**	**Test 5**	**Mean**	**nlogn**	**k**
10	0.002	0.001	0.001	0.001	0.000	0.001	10	0.0001
100	0.005	0.005	0.005	0.005	0.004	0.005	200	0.000024
1000	0.034	0.032	0.033	0.034	0.033	0.033	3000	0.00001106666667
10000	0.277	0.282	0.281	0.281	0.283	0.281	40000	0.00000702
100000	2.910	2.917	2.923	2.917	2.914	2.916	500000	0.0000058324
500000	14.900	14.958	14.652	15.088	14.968	14.913	2849485.002	0.00000523364748
1000000	29.715	29.952	29.679	29.746	29.975	29.813	6000000	0.0000049689
							Mean k:	0.00002258880202

Figure 2.1: My empirical results and an estimation for the constant of proportionality (k).

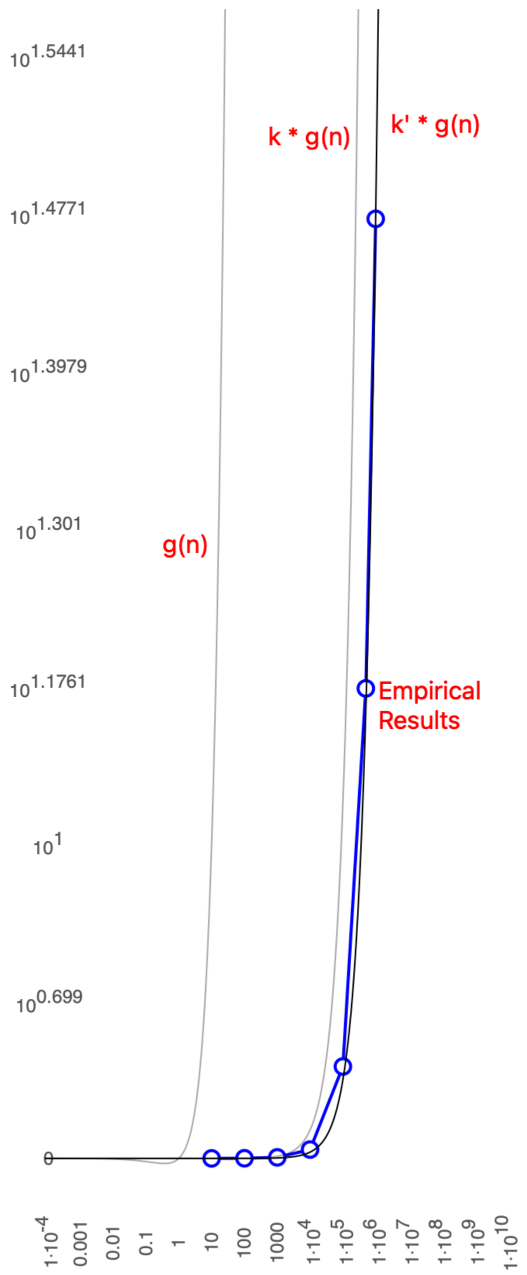


*Figure 2.2: Plot of my empirical results (in blue). The x-axis shows the number of points used and the y-axis shows the runtime in seconds. g(n), k*g(n), and k’*g(n) are also shown.*

Source Code

convex_hull.py

```python
# Time: O(1) Space: O(1)
def join_hulls(self, left_hull, right_hull, upper_tangent, lower_tangent):
   new_hull = Hull(left_hull.get_left_most(), right_hull.get_right_most())

   upper_tangent.stitch()   # Time: O(1) Space: O(1)
   lower_tangent.stitch()   # Time: O(1) Space: O(1)

   return new_hull

# Time: O(nlogn) Space: O(n)
def compute_hull_helper(self, points):
   # Base Case
   if len(points) == 1:
       single_point = Point(points[0])
       new_hull = Hull(single_point, single_point)

       return new_hull

   middle_index = len(points) // 2

   # Recursive calls
   left_hull = self.compute_hull_helper(points[:middle_index])                            # Time: O(nlogn) Space: O(n)
   right_hull = self.compute_hull_helper(points[middle_index:])                           # Time: O(nlogn) Space: O(n)

   # Find tangents
   upper_tangent = UpperTangent(left_hull.get_right_most(), right_hull.get_left_most())    # Time: O(n) Space: O(n)
   lower_tangent = LowerTangent(left_hull.get_right_most(), right_hull.get_left_most())    # Time: O(n) Space: O(n)

   return self.join_hulls(left_hull, right_hull, upper_tangent, lower_tangent)             # Time: O(1) Space: O(1)

# Time: O(nlogn) Space: O(n)
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()

   # Sorts the points
   points.sort(key=lambda point: point.x())                # Time: O(nlogn) Space: O(n)

   t2 = time.time()

   t3 = time.time()

   # Returns lines
   polygon = self.compute_hull_helper(points).to_lines()   # Time: O(nlogn) Space: O(n)

   t4 = time.time()

   # 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.show_hull(polygon, RED)
   self.show_text('Time Elapsed (Convex Hull): {:3.3f} sec'.format(t4 - t3))
```

Point.py

```python
# All functions in Point run with constant time and space complexity
class Point:
   def __init__(self, point):
       self.point = point

       self.clock_point = self
       self.counter_point = self
```

```
    def get_clock(self):
        return self.clock_point

    def get_counter(self):
        return self.counter_point

    def set_clock(self, clock):
        self.clock_point = clock

    def set_counter(self, counter):
        self.counter_point = counter

    def x(self):
        return self.point.x()

    def y(self):
        return self.point.y()

    def to_qpointf(self):
        return self.point
```

Hull.py

```
from PyQt6.QtCore import QLineF


class Hull:
    def __init__(self, left_most, right_most):
        self.left_most = left_most
        self.right_most = right_most

    def get_left_most(self):
        return self.left_most

    def get_right_most(self):
        return self.right_most

    # Time: O(n) Space: O(n)
    def to_lines(self):
        lines = []
        current_point = self.left_most.get_clock()

        # Adds first line
        lines.append(QLineF(self.left_most.to_qpointf(), current_point.to_qpointf()))

        # Adds other lines
        while current_point != self.left_most:
            lines.append(QLineF(current_point.to_qpointf(),
current_point.get_clock().to_qpointf()))
```

```
            current_point = current_point.get_clock()

        return lines
```

Tangent.py

```
from PyQt6.QtCore import QLineF
from abc import ABC, abstractmethod


# An abstract parent class for UpperTangent and LowerTangent
class Tangent(ABC):
   def __init__(self, left, right):
       self.left = left
       self.right = right

       self.orient()

   @abstractmethod
   def cycle_left(self):
       pass

   @abstractmethod
   def cycle_right(self):
       pass

   # Time: O(n) Space: O(n)
   def orient(self):
       while True:
           left_cycle_count = self.cycle_left()            # Time: O(n) Space: O(n)
           right_cycle_count = self.cycle_right()          # Time: O(n) Space: O(n)

           if left_cycle_count == 0 and right_cycle_count == 0:
               break

   def get_left(self):
       return self.left

   def get_right(self):
       return self.right

   @abstractmethod
   def stitch(self):
       pass

   def get_slope(self):
       return (self.left.y() - self.right.y()) / \
              (self.left.x() - self.right.x())

   def to_line(self):
       return QLineF(self.left.to_qpointf(), self.right.to_qpointf())
```

UpperTangent.py

```
from Tangent import Tangent


class UpperTangent(Tangent):

    # Time: O(n) Space: O(n)
    def cycle_left(self):
        least_slope = self.get_slope()
        cycle_count = 0

        self.left = self.left.get_counter()

        while self.get_slope() < least_slope:       # Time: O(n) Space: O(n)
            least_slope = self.get_slope()

            self.left = self.left.get_counter()
            cycle_count += 1

        self.left = self.left.get_clock()

        return cycle_count

    # Time: O(n) Space: O(n)
    def cycle_right(self):
        greatest_slope = self.get_slope()
        cycle_count = 0

        self.right = self.right.get_clock()

        while self.get_slope() > greatest_slope:    # Time: O(n) Space: O(n)
            greatest_slope = self.get_slope()

            self.right = self.right.get_clock()
            cycle_count += 1

        self.right = self.right.get_counter()

        return cycle_count

    # Time: O(1) Space: O(1)
    def stitch(self):
        self.left.set_clock(self.right)
        self.right.set_counter(self.left)

        if self.left.get_counter == self.left:
            self.left.set_counter(self.right)

        if self.right.get_clock == self.right:
            self.right.set_clock(self.left)
```

LowerTangent.py

```
from Tangent import Tangent


# Time: O(n) Space: O(n)
class LowerTangent(Tangent):
    def cycle_left(self):
        greatest_slope = self.get_slope()
        cycle_count = 0

        self.left = self.left.get_clock()

        while self.get_slope() > greatest_slope:    # Time: O(n) Space: O(n)
            greatest_slope = self.get_slope()

            self.left = self.left.get_clock()
            cycle_count += 1

        self.left = self.left.get_counter()

        return cycle_count

    # Time: O(n) Space: O(n)
    def cycle_right(self):
        least_slope = self.get_slope()
        cycle_count = 0

        self.right = self.right.get_counter()

        while self.get_slope() < least_slope:       # Time: O(n) Space: O(n)
            least_slope = self.get_slope()

            self.right = self.right.get_counter()
            cycle_count += 1

        self.right = self.right.get_clock()
        return cycle_count

    # Time: O(1) Space: O(1)
    def stitch(self):
        self.left.set_counter(self.right)
        self.right.set_clock(self.left)

        if self.left.get_clock == self.left:
            self.left.set_clock(self.right)

        if self.right.get_counter == self.right:
            self.right.set_counter(self.left)
```