CS312 Project 2 Convex Hull Report

Cannon Farr

1. Code:
    a. (See Appendix)

2. Time and Space Complexity

    a. Time Complexity: O(n log(n))

    The time n log(n) is the worst case scenario for this algorithm and that is due to the significance of the divide_and_conquer method. The method takes a group of points and splits them up into two groups and runs the tangent and merge algorithms on them. It does this recursively effectively halving the amount of points for each iteration.

    Secondly, the O(n) time comes from iteratively calculating the upper and lower tangents of all of the sub-hulls which ends up being up to all of the points in the plot.

    Finally, the merging of the sub-hulls is also O(n) time which is also reconstructing the order of up to n points in the list.

    b. Space Complexity: O(n)

    The space complexity is mostly comprised of just the storage of the points themselves which is essentially just n. There are also a couple of temporary lists and values that are stored within the tangent algorithms and the merge algorithm, but these only happen once per sub hull calculation and are deleted afterwards.

c. Recurrance Relation and Master Theorem:

Recurrance relation: $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$

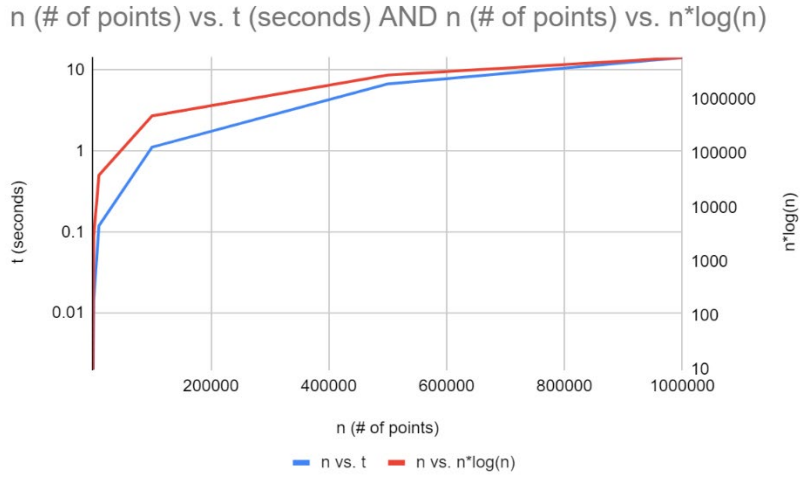
If we use the master theorem to find the complexity we get the following values:

$a = 2$
$b = 2$
$d = 1$

$$\frac{a}{b^d} \rightarrow \frac{2}{2^1} = 1 \rightarrow O\left(n^d \log(n)\right) \rightarrow O\left(nlog(n)\right)$$

3. Outcomes:

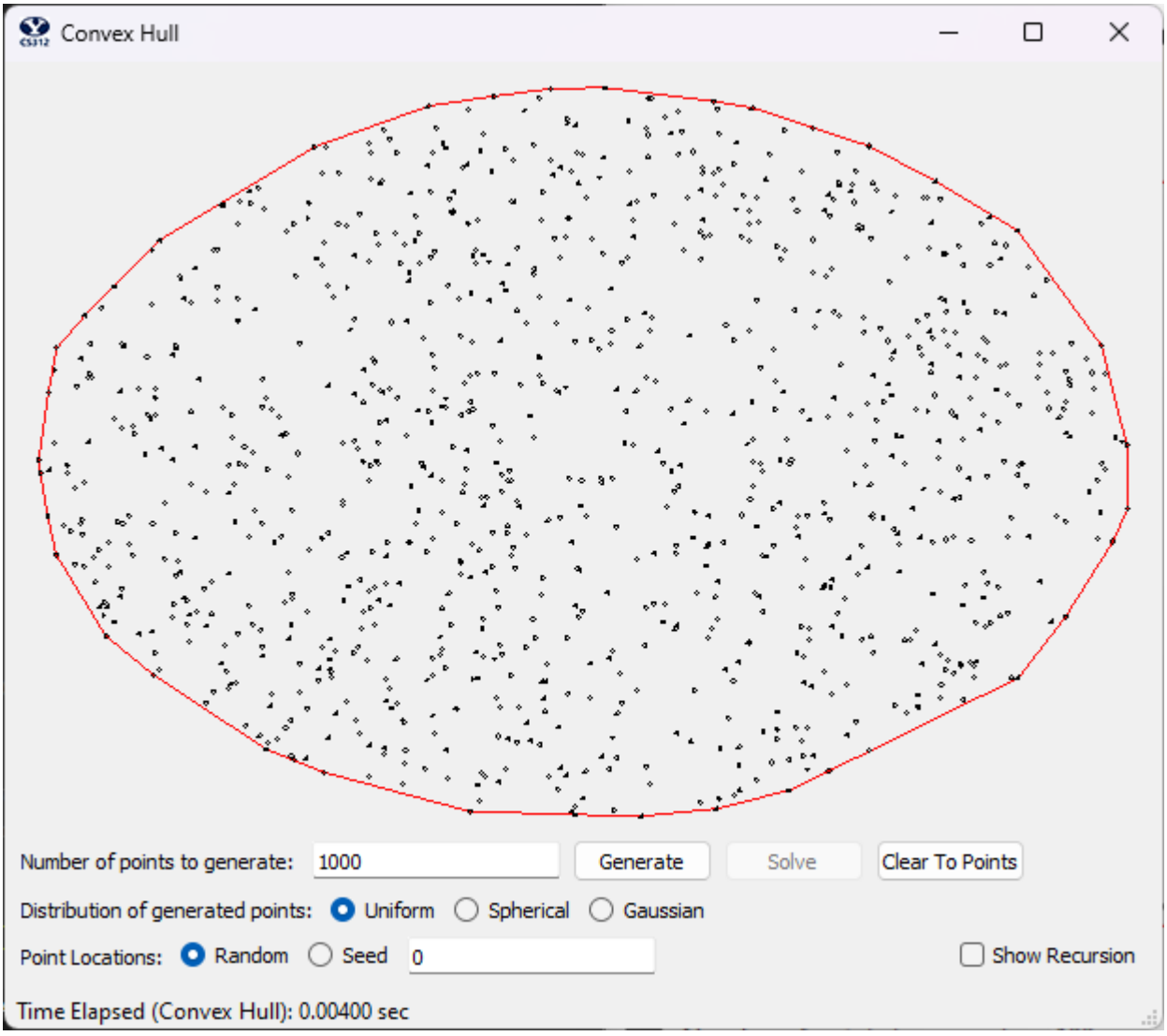
n	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average	n*log(n)
10	0	0	0	0	0	0	**10**
100	0.002981	0.000997	0.002033	0.001993	0.001993	**0.002000**	**200**
1000	0.015001	0.012965	0.018950	0.011968	0.013963	**0.014569**	**3000**
10000	0.119273	0.123263	0.120311	0.118347	0.120192	**0.120277**	**40000**
100000	1.111253	1.140201	1.127599	1.108008	1.114733	**1.120359**	**500000**
500000	6.704785	6.812290	6.754357	6.780400	6.785395	**6.767445**	**2849485.002**
1000000	14.292636	13.961631	13.706400	14.978358	14.534557	**14.294716**	**6000000**

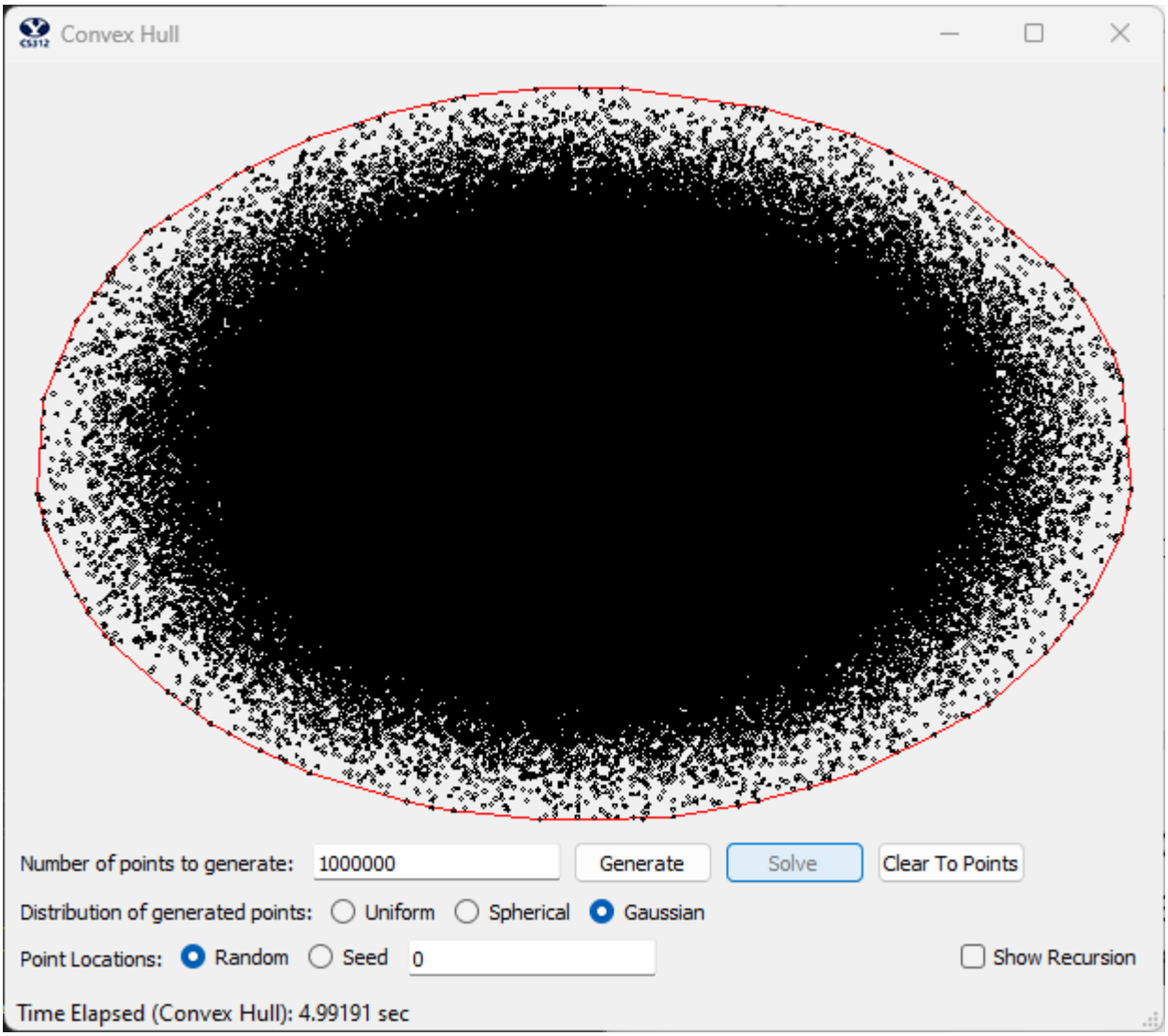


4. Empirical Analysis

We see that the time follows the nice pattern along with the graph n log n

5. Screenshots:





Appendix:

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

        t1 = time.time()
        # SORT THE POINTS BY INCREASING X-VALUE
        points.sort(key=lambda p: p.x())
        t2 = time.time()

        t3 = time.time()
```

```
        A = 2
        polygon = self.divide_and_conquer(points, A)
        # REPLACE THE LINE ABOVE WITH A CALL TO YOUR DIVIDE-AND-CONQUER CONVEX HULL
SOLVER
        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
        hull = []
        for i in range(len(polygon)):
            hull.append(QLineF(polygon[i], polygon[(i + 1) % len(polygon)]))
        self.showHull(hull,RED)
        self.showText('Time Elapsed (Convex Hull): {:3.5f} sec'.format(t4-t3))


    def divide_and_conquer(self, points: list, a: int):
        # print('divide')
        if len(points) <= 3:
            if len(points) == 3:
                slope1 = (points[1].y() - points[0].y()) / (points[1].x() -
points[0].x())
                slope2 = (points[2].y() - points[0].y()) / (points[2].x() -
points[0].x())
                if slope1 > slope2:
                    return [points[0], points[1], points[2]]
                else:
                    return [points[0], points[2], points[1]]
            return points
        parts, left = divmod(len(points), a)
        # split = [points[i * parts + min(i, left):(i + 1) * parts + min(i + 1,
left)] for i in range(a)]
        split = []
        for i in range(a):
            split.append(points[i * parts + min(i, left):(i + 1) * parts + min(i + 1,
left)])

        y1 = []
        for i in range(a):
            y1.append(self.divide_and_conquer(split[i], a))
            # print('divide return')
        y = self.merge(y1, a)
        # print('merge return')
        return y
```

```python
def merge(self, y: list, a: int):
    # print('merge')
    L = y[0]
    R = y[1]
    for i in range(a - 1):
        # #DEBUG
        # shapeR = []
        # shapeL = []
        # for i in range(len(L)):
        #     shapeL.append(QLineF(L[i], L[((i + 1) % len(L))]))
        # self.showHull(shapeL, BLUE)
        # for i in range(len(R)):
        #     shapeR.append(QLineF(R[i], R[((i + 1) % len(R))]))
        # self.showHull(shapeR, BLUE)

        upper = self.findUpperTangent(L,R)
        lower = self.findLowerTangent(L,R)
        temp = []

        i = lower[0]
        while True:
            temp.append(L[i])
            if i == upper[0]:
                break
            i = (i + 1) % len(L)

        # Traverse R from upper[1] to lower[1] in a circular manner
        i = upper[1]
        while True:
            temp.append(R[i])
            if i == lower[1]:
                break
            i = (i + 1) % len(R)

        # self.eraseHull(shapeL)
        # self.eraseHull(shapeR)

        # tempLines = []
        # for i in range(len(temp)):
        #     tempLines.append(QLineF(temp[i], temp[((i + 1) % len(temp))]))
        # self.showHull(tempLines, (0, 255, 255))
        # self.eraseHull(tempLines)
```

```
            L = temp


        return L



    def findUpperTangent(self, L: list[QPointF], R: list[QPointF]):
        rightMostL = L.index(max(L, key=lambda p: p.x()))
        leftMostR = R.index(min(R, key=lambda p: p.x()))



        # line = QLineF(L[rightMostL], R[leftMostR])
        # self.blinkTangent(line, GREEN)
        done = False
        temp = (L[rightMostL].y() - R[leftMostR].y()) / (L[rightMostL].x() -
R[leftMostR].x())
        done = False
        while not done:
            done = True
            while temp > ((L[(len(L) - 1 if rightMostL == 0 else rightMostL - 1)].y()
- R[leftMostR].y()) / (L[(len(L) - 1 if rightMostL == 0 else rightMostL - 1)].x() -
R[leftMostR].x())):
                r = (len(L) - 1 if rightMostL == 0 else rightMostL - 1)
                temp = (L[r].y() - R[leftMostR].y()) / (L[r].x() - R[leftMostR].x())
                rightMostL = r
                done = False
                # #DEBUG
                # line = QLineF(L[rightMostL], R[leftMostR])
                # self.blinkTangent(line, GREEN)
            while temp < ((R[(0 if leftMostR == len(R) - 1 else leftMostR + 1)].y() -
L[rightMostL].y()) / (R[(0 if leftMostR == len(R) - 1 else leftMostR + 1)].x() -
L[rightMostL].x())):
                r = (0 if leftMostR == len(R) - 1 else leftMostR + 1)
                temp = (R[r].y() - L[rightMostL].y()) / (R[r].x() -
L[rightMostL].x())
                leftMostR = r
                done = False

                # #DEBUG
                # line = QLineF(L[rightMostL], R[leftMostR])
                # self.blinkTangent(line, GREEN)
        return [rightMostL, leftMostR]
```

```
    def findLowerTangent(self, L: list[QPointF], R: list[QPointF]):
        # print('lower')
        rightMostL = L.index(max(L, key=lambda p: p.x()))
        leftMostR = R.index(min(R, key=lambda p: p.x()))


        done = False
        temp = (L[rightMostL].y() - R[leftMostR].y()) / (L[rightMostL].x() -
R[leftMostR].x())
        done = False
        while not done:
            done = True
            while temp < ((L[(0 if rightMostL == len(L) - 1 else rightMostL + 1)].y()
- R[leftMostR].y()) / (L[(0 if rightMostL == len(L) - 1 else rightMostL + 1)].x() -
R[leftMostR].x())):
                r = (0 if rightMostL == len(L) - 1 else rightMostL + 1)
                temp = (L[r].y() - R[leftMostR].y()) / (L[r].x() - R[leftMostR].x())
                done = False
                rightMostL = r

                # #  DEBUG
                # line = QLineF(L[rightMostL], R[leftMostR])
                # self.blinkTangent(line, RED)
            while temp > ((R[(len(R) - 1 if leftMostR == 0 else leftMostR - 1)].y() -
L[rightMostL].y()) / (R[(len(R) - 1 if leftMostR == 0 else leftMostR - 1)].x() -
L[rightMostL].x())):
                r = (len(R) - 1 if leftMostR == 0 else leftMostR - 1)
                temp = (R[r].y() - L[rightMostL].y()) / (R[r].x() -
L[rightMostL].x())
                leftMostR = r
                done = False

                # #DEBUG
                # line = QLineF(L[rightMostL], R[leftMostR])
                # self.blinkTangent(line, RED)
        return [rightMostL, leftMostR]
```