Exploration of the Divide and Conquer Convex Hull Algorithm

Katherine Wilsdon

Idaho State University

Source Code

Compute Hull Algorithm

The Compute Hull algorithm first asserts that *points* is a list of *QPointF*. Next, the *points* are sorted according to the x coordinate, from left to right. The *convex_hull* function returns smallest, clockwise set of points in a convex shape. The list of points is converted into a list of *QLineF* lines. Lastly, the function shows the hull and time elapsed on the GUI.

```
definition compute_hull(points)
    points.sort(points) return points in ascending order of point.x()
    convex_hull(points) returns set of points of convex hull in clockwise order
    polygon = create lines from points
    showHull(polygon)
```

Figure 1: Pseudo Code for Compute Hull Algorithm.

```
# 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()
    # Sort points depending on the x direction
    points.sort(key=lambda point: point.x())
    print(points)
    t2 = time.time()

    t3 = time.time()
    # Creates a set of lines from the set of points returned from convex_hull
    points_of_polygon = self.convex_hull(points)
    polygon = [QLineF(points_of_polygon[i], points_of_polygon[(i + 1) % len(points_of_polygon)]) for i in
               range(len(points_of_polygon))]
    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.showHull(polygon, RED)
    self.showText('Time Elapsed (Convex Hull): {:3.3f} sec'.format(t4 - t3))
```

Figure 2: Compute Hull Algorithm.

Convex Hull Algorithm

The Convex Hull algorithm recursively splits the points in half until reaching a base case of 3 points or less. If the size of points is less than or equal to 2, the set is already in clockwise order and the set of points is returned. If the size of points equals 3, return the clockwise orientation of the three points. Once obtaining the Left and Right Hulls, the two hulls are merged into one continuous hull. If the *Show Recursion* box on the GUI is checked, the list of points is converted into a list of *QLineF* lines. Then, the function shows the hull and time elapsed on the GUI. After pausing for 2 seconds, the algorithm continues where it left off.

```
definition convex_hull(points)
    if length of points <= 2
        return points
    elif length of points == 3
        return clockwise(point1, point2, point3)
    mid = floor(length of points / 2)
    left = points from 0 to (mid - 1)
    right = points from mid to (length of points - 1)
    L = convex_hull(left)
    R = convex_hull(right)
    return merge(L, R)
```

Figure 3: Pseudo Code for Convex Hull Algorithm.

```
definition clockwise(point1, point2, point3)
    Create array to hold clockwise points
    value = (point2.y() - point1.y()) * (point3.x() - point2.x()) -
            (point2.x() - point1.x()) * (point3.y() - point2.y())
    if value == clockwise
        return array[point1, point2, point3]
    if value == counterclockwise
        return array[point1, point3, point2]
```

Figure 4: Pseudo Code for Clockwise Algorithm.

```
# Determines the smallest, clockwise set of points in a convex shape
def convex_hull(self, points):
    # Return points in clockwise order
    if len(points) <= 2:
        return points
    elif len(points) == 3:
        return self.clockwise(points[0], points[1], points[2])
    midpoint = (int)(len(points) / 2)
    left = points[:midpoint]
    right = points[midpoint:]
    # Recursively splits the points in half and then merges the Left Hull and the Right Hull
    Left_Hull = self.convex_hull(left)
    Right_Hull = self.convex_hull(right)
    points_of_polygon = self.merge(Left_Hull, Right_Hull)
    # If `Show Recursion` box is checked on the GUI
    if self.pause:
        polygon = [QLineF(points_of_polygon[i], points_of_polygon[(i + 1) % len(points_of_polygon)]) for i in
                   range(len(points_of_polygon))]
        self.showHull(polygon, RED)
    return points_of_polygon
```

Figure 5: Convex Hull Algorithm.

```
# Returns a set of three points in clockwise order
def clockwise(self, p1, p2, p3):
    clockwise_points = []
    value = (p2.y() - p1.y()) * (p3.x() - p2.x()) - (p2.x() - p1.x()) * (p3.y() - p2.y())
    # When value is clockwise, return the points as previously ordered
    if value > 0:
        clockwise_points.append(p1)
        clockwise_points.append(p2)
        clockwise_points.append(p3)
        return clockwise_points
    # When value is counterclockwise, switch the order of the second and third points
    elif value < 0:
        clockwise_points.append(p1)
        clockwise_points.append(p3)
        clockwise_points.append(p2)
        return clockwise_points
```

Figure 6: Clockwise Algorithm.

Merge Algorithm

In order to merge two hulls together, the rightmost point of the Left Hull and the leftmost point of the Right Hull are determined. These points are parameters in the *upper_tangent* and *lower_tangent* methods that return the indices of the upper and lower tangent points for both the Left Hull and the Right Hull. The combined hull is created starting with the lower tangent of Left Hull … upper tangent of Left Hull → upper tangent of Right Hull … lower tangent of Right Hull. The merged hull is returned.

```
# Combines two Hulls into one Hull
def merge(self, L, R):
    points = []

    Lsize = len(L)
    Rsize = len(R)
    L_rightmost = 0
    R_leftmost = 0

    # Find the rightmost point of L
    for i in range(1, Lsize):
        if L[i].x() > L[L_rightmost].x():
            L_rightmost = i

    # Find the leftmost point of R
    for i in range(1, Rsize):
        if R[i].x() < R[R_leftmost].x():
            R_leftmost = i

    # Finds the upper and lower tangent points
    Ltangent_upper, Rtangent_upper = self.upper_tangent(L, R, L_rightmost, R_leftmost)
    Ltangent_lower, Rtangent_lower = self.lower_tangent(L, R, L_rightmost, R_leftmost)

    # Creates the new Hull from lower tangent to upper tangent of the Left Hull
    # then to the upper tangent to the lower tangent of the Right Hull
    i = Ltangent_lower
    points.append(L[i])
    while i != Ltangent_upper:
        i = (i + 1) % Lsize
        points.append(L[i])
    i = Rtangent_upper
    points.append(R[i])
    while i != Rtangent_lower:
        i = (i + 1) % Rsize
        points.append(R[i])
    return points
```

Figure 7: Merge Algorithm.

```
definition merge(L, R)
    Create array
    Lsize = length of L
    Rsize = length of R
    L_rightmost = 0
    R_leftmost = 0

    for i in range of the size of L:
        find the rightmost point of L
    for i in range of the size of R:
        find the leftmost point of R

    Ltangent_upper, Rtangent_upper = upper_tangent(L, R, L_rightmost, R_leftmost)
    Ltangent_lower, Rtangent_lower = lower_tangent(L, R, L_rightmost, R_leftmost)

    append points to array from Ltangent_lower to Ltangent_upper
    append points to array from Rtangent_upper to Rtangent_lower
    return array
```

Figure 8: Pseudo Code for Merge Algorithm.

Upper and Lower Tangent Algorithms

The upper and lower tangent algorithms contain one outer while loop that proceeds while the tangent is not found and two inner while loops that separately find the tangent on the Left Hull and the tangent on the Right Hull. Each of the inner while loops check the previous and next index of the current tangent to determine whether the tangent should be incremented or decremented. The orientation function determines whether the points are clockwise, counterclockwise, or co-linear.

```
definition orientation(point1, point2, point3)
    Create array to hold clockwise points
    value = (point2.y() - point1.y()) * (point3.x() - point2.x()) -
            (point2.x() - point1.x()) * (point3.y() - point2.y())
    if value == co-linear
        return 0
    if value == clockwise
        return 1
    return -1
```

Figure 9: Pseudo Code for Orientation Algorithm.

```
definition upper_tangent(L, R, L_rightmost, R_leftmost)
    Create array
    Lsize = length of L
    Rsize = length of R
    Ltangent = L_rightmost
    Rtangent = R_leftmost

    isFound = False

    while not isFound
        while the oreintation of either incrementing or decrementing Ltangent and holding Rtangent constant
        is clockwise or co-linear
            increment or decrement Ltangent
        while the orientation of either incrementing or decrementing Rtangent and holding Ltangent constant
        is counterclockwise or co-linear
            increment or decrement Rtangent
            isFound = False

    return Ltangent, Rtangent
```

Figure 10: Pseudo Code for Upper Tangent Algorithm.

```
definition lower_tangent(L, R, L_rightmost, R_leftmost)
    Create array
    Lsize = length of L
    Rsize = length of R
    Ltangent = L_rightmost
    Rtangent = R_leftmost

    isFound = False

    while not isFound
        while the oreintation of either incrementing or decrementing Rtangent and holding Ltangent constant
        is clockwise or co-linear
            increment or decrement Rtangent
        while the orientation of either incrementing or decrementing Ltangent and holding Rtangent constant
        is counterclockwise or co-linear
            increment or decrement Ltangent
            isFound = False

    return Ltangent, Rtangent
```

Figure 11: Pseudo Code for Lower Tangent Algorithm.

```
# Determines if the points are clockwise, counterclockwise, or co-linear
def orientation(self, p1, p2, p3):
    value = (p2.y() - p1.y()) * (p3.x() - p2.x()) - (p2.x() - p1.x()) * (p3.y() - p2.y())
    # value is co-linear
    if value == 0:
        return 0
    # value is clockwise
    if value > 0:
        return 1
    # value is counterclockwise
    return -1
```

Figure 12: Orientation Algorithm.

```
# Finds the upper tangent points of the Left Hull and the Right Hull
def upper_tangent(self, L, R, L_rightmost, R_leftmost):
    Lsize = len(L)
    Rsize = len(R)
    Ltangent = L_rightmost
    Rtangent = R_leftmost
    isFound = False
    while not isFound:
        isFound = True
        # Find the upper tangent on the Left Hull
        while (self.orientation(R[Rtangent], L[Ltangent], L[(Lsize + Ltangent - 1) % Lsize]) >= 0 or
               self.orientation(R[Rtangent], L[Ltangent], L[(Lsize + Ltangent + 1) % Lsize]) >= 0):
            # If the previous index on the Left Hull is clockwise or co-linear, decrement the left tangent point
            if self.orientation(R[Rtangent], L[Ltangent], L[(Lsize + Ltangent - 1) % Lsize]) >= 0:
                Ltangent = (Lsize + Ltangent - 1) % Lsize
            # If the next index on the Left Hull is clockwise or co-linear, increment the left tangent point
            if self.orientation(R[Rtangent], L[Ltangent], L[(Lsize + Ltangent + 1) % Lsize]) >= 0:
                Ltangent = (Lsize + Ltangent + 1) % Lsize
        # Find the upper tangent on the Right Hull
        while (self.orientation(L[Ltangent], R[Rtangent], R[(Rtangent + 1) % Rsize]) <= 0 or
               self.orientation(L[Ltangent], R[Rtangent], R[(Rtangent - 1) % Rsize]) <= 0):
            # If the next index on the Right Hull is counterclockwise or co-linear, increment the right tangent point
            if self.orientation(L[Ltangent], R[Rtangent], R[(Rtangent + 1) % Rsize]) <= 0:
                Rtangent = (Rtangent + 1) % Rsize
                isFound = False
            # If the previous index on the Right Hull is counterclockwise or co-linear, decrement the right tangent point
            if self.orientation(L[Ltangent], R[Rtangent], R[(Rtangent - 1) % Rsize]) <= 0:
                Rtangent = (Rtangent - 1) % Rsize
                isFound = False
    return Ltangent, Rtangent
```

Figure 13: Upper Tangent Algorithm.

```
# Finds the lower tangent points of the Left Hull and the Right Hull
def lower_tangent(self, L, R, L_rightmost, R_leftmost):
    Lsize = len(L)
    Rsize = len(R)
    Ltangent = L_rightmost
    Rtangent = R_leftmost
    isFound = False
    while not isFound:
        isFound = True
        # Find the lower tangent on the Right Hull
        while (self.orientation(L[Ltangent], R[Rtangent], R[(Rsize + Rtangent + 1) % Rsize]) >= 0 or
               self.orientation(L[Ltangent], R[Rtangent], R[(Rsize + Rtangent - 1) % Rsize]) >= 0):
            # if the next index on the Right Hull is clockwise or co-linear, increment the right tangent point
            if self.orientation(L[Ltangent], R[Rtangent], R[(Rsize + Rtangent + 1) % Rsize]) >= 0:
                Rtangent = (Rsize + Rtangent + 1) % Rsize
            # if the previous index on the Right Hull is clockwise or co-linear, decrement the right tangent point
            if self.orientation(L[Ltangent], R[Rtangent], R[(Rsize + Rtangent - 1) % Rsize]) >= 0:
                Rtangent = (Rsize + Rtangent - 1) % Rsize
        # Find the lower tangent on the Left Hull
        while (self.orientation(R[Rtangent], L[Ltangent], L[(Ltangent - 1) % Lsize]) <= 0 or
               self.orientation(R[Rtangent], L[Ltangent], L[(Ltangent + 1) % Lsize]) <= 0):
            # if the previous index on the Left Hull is counterclockwise or co-linear, decrement the left tangent point
            if self.orientation(R[Rtangent], L[Ltangent], L[(Ltangent - 1) % Lsize]) <= 0:
                Ltangent = (Ltangent - 1) % Lsize
                isFound = False
            # if the next index on the Left Hull is counterclockwise or co-linear, increment the left tangent point
            if self.orientation(R[Rtangent], L[Ltangent], L[(Ltangent + 1) % Lsize]) <= 0:
                Ltangent = (Ltangent + 1) % Lsize
                isFound = False
    return Ltangent, Rtangent
```

Figure 14: Lower Tangent Algorithm.

Time and Space Complexity of Algorithms

Upper and Lower Tangent Algorithms

The time and space complexity for finding the upper and lower tangents of two hulls is O(1). The runtime is constant time because at most a few points are checked to find the tangent of the Left Hull and the Right Hull.

Merge Algorithm

The time and space complexity for merging the two hulls is O(n). The runtime is linear because algorithm visits almost every index in both hulls to add each point to the merged hull.

Convex Hull Algorithm

The time and space complexity for the convex hull algorithm is O(n log(n)). Since the algorithm recursively halves the size of the list, the runtime for this section is O(log(n). After reaching the base case, the Left and Right Hulls are merge, which has a runtime of O(n) as previously discussed.

Compute Hull Algorithm

The time and space complexity for the compute hull algorithm is O(n log(n)). Sorting the points by the x coordinate has a runtime of O(n log(n)). As previously discussed, the convex hull algorithm has a runtime of O(n log(n)). Creating a list of lines from a list of points returned by the convex hull algorithm has a runtime of O(n).

Recurrence Relation

For the recurrence relation, the number of subtasks to be solved, a, and the size of the sub-instances, b, is 6. This is because the base case is 3 points or less. The polynomial order of work at each node, d, is 1. So the recurrence relation is $T(n) = 6T(n/6) + O(n)$ resulting in a complexity of O(n log(n)).

Experimental Outcomes

The time was recorded for computing the convex hull of 5 sets of n points where n ∈ {10, 100, 1000, 10,000, 100,000, 500,000, 1,000,000} using a seed from 0 to 8 million incrementing by 2 million. The 5 sets were averaged as shown as the empirical data. The theoretical analysis was calculated by kn log n, where the constant of proportionality k = 1/320,000 and was found by trial and error. So, the best fit of the empirical data is an order of growth of kn log n.

Space and Time Complexity For Convex Hull						Empirical	Theoretical
N (units)	Set 1 (seconds)	Set 2 (seconds)	Set 3 (seconds)	Set 4 (seconds)	Set 5 (seconds)	Mean (seconds)	k = 1/320,000, k n log n (seconds)
10	0	0.001	0	0	0	0.000	0.000
100	0.002	0.002	0.001	0.001	0.001	0.001	0.001
1000	0.019	0.018	0.019	0.018	0.018	0.018	0.009
10000	0.162	0.162	0.167	0.159	0.182	0.166	0.125
100000	1.506	1.47	1.498	1.475	1.458	1.481	1.563
500000	10.954	8.564	10.618	8.872	9.75	9.752	8.905
1000000	17.547	17.741	18.108	17.353	17.795	17.709	18.750

Table 1: Space and Time Complexity for Theoretical vs. Empirical Convex Hull

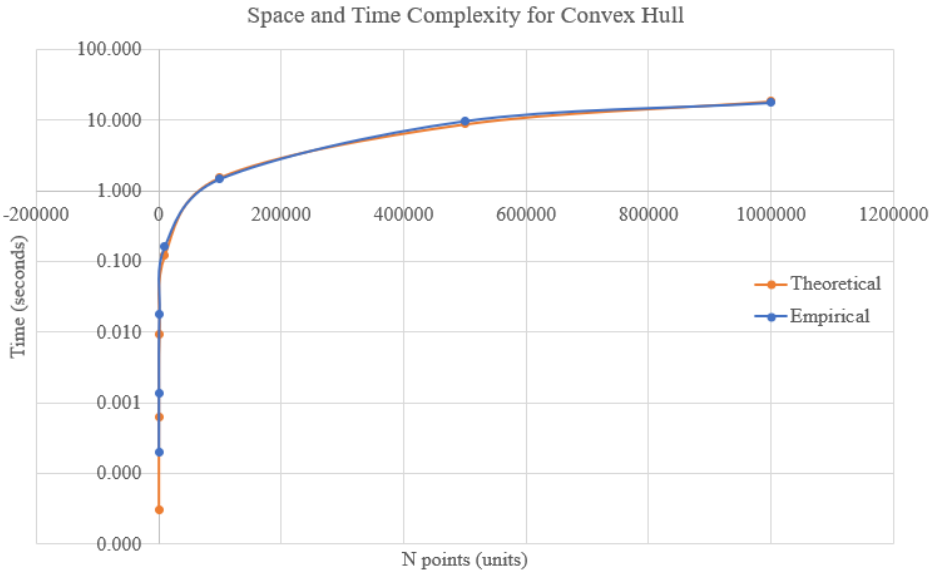

Figure 15: Space and Time Complexity for Theoretical vs. Empirical Convex Hull

Analysis of Experimental Outcomes

Overall, the theoretical and empirical analyses are equivalent and contain only minute differences. The theoretical analysis over-estimated N = 100,000 by 0.08 of a second and N = 1,000,000 by 1 second. The theoretical analysis under-estimated N = 10,000 by 0.04 of a second and N = 500,000 by 0.85 of a second. So, the space and time complexity is asserted to be kn log n because the differences are minimal.

Experimentation of Inputs

Figure 16 and 17 depict examples of a convex hull for 100 points and 1,000 points. The exterior points are connected by lines in a convex shape.

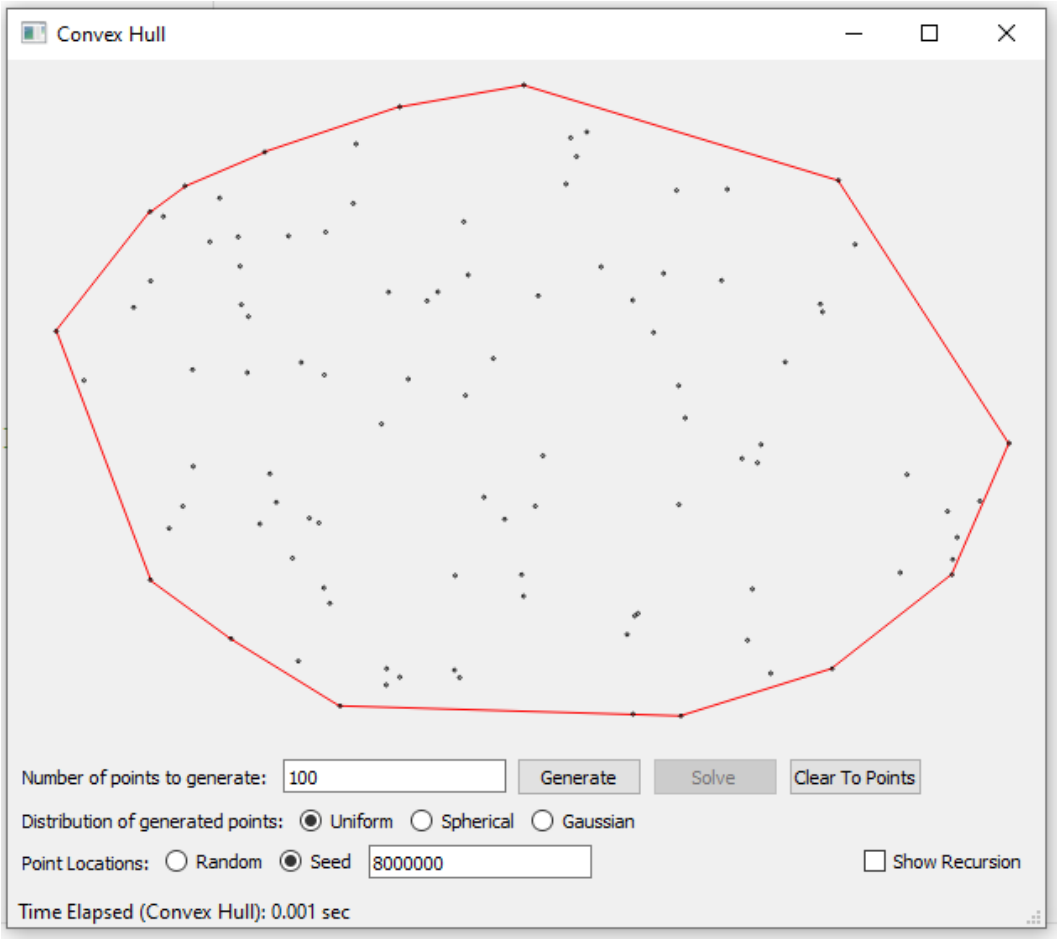

Figure 16: An example of the convex hull of 100 points.

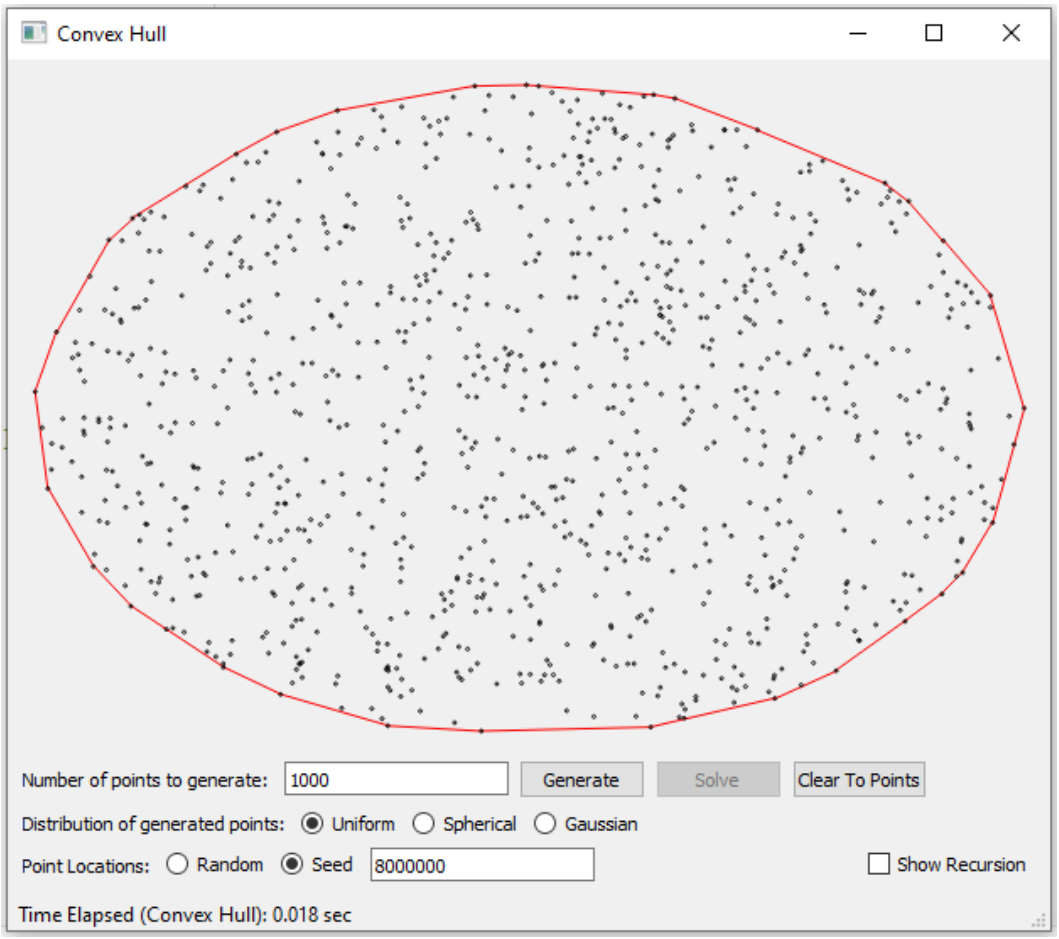

Figure 17: An example of the convex hull of 1000 points.