4-1990

A Proof for a QuickHull Algorithm

Jonathan Scott Greenfield,
Syracuse University

SU-CIS-90-03

A Proof for a QuickHull Algorithm

Johnathan Greenfield

April 1990

School of Computer and Information Science
Suite 4-116
Center for Science and Technology
Syracuse, New York 13244-4100

(315) 443-2368

A Proof for a QuickHull Algorithm

Jonathan Greenfield

School of Computer and Information Science
Syracuse University
Syracuse, New York 13244

April 1990

Abstract: The planar convex hull problem is fundamental to computational geometry and has many applications, including pattern recognition and image processing. QuickHull is a simple planar convex hull algorithm analogous to Hoare's QuickSort [1]. This paper presents a pedagogical description and analysis of a QuickHull algorithm, along with a formal proof of correctness.

Keywords: complexity analysis, computational geometry, convex hull, correctness proof, divide-and-conquer, prune-and-search, QuickHull.

1. Introduction

Many algorithms have been proposed in order to solve the planar convex hull problem [2]. The algorithms are varied, but generally they have time complexities of either O(n log n) (such as Graham's Scan [3]) or O(mn) (such as Jarvis' March (the package-wrapping method) [4]), where n is the number of points and m is the number of hull vertices [2]. Divide-and-conquer algorithms have been proposed [5,6] including algorithms with linear expected running time [6].

This paper describes a simple divide-and-conquer, prune-and-search algorithm for finding the convex hull of a finite number of planar points. Although our version of this algorithm was developed independently, [7,8,9] proposed similar versions of this algorithm previously (later termed QuickHull algorithms by [10]). In contrast to the QuickHull descriptions of [7,8,9,10], we present a proof of correctness for our algorithm. The proof offers some insight into the difficulty in generalizing this algorithm for non-planar convex hull problems.

The algorithm has a worst-case time complexity of $O(n^2)$, but an expected time complexity of O(n log n), when all points are hull vertices. Furthermore, when the number of hull vertices is small compared to the number of points (in an asymptotic sense, as is the case for many distributions [7,11]), the algorithm has an expected time complexity of O(n).

2. Informal problem definition

The *convex hull problem* is one of the common computational geometry problems and has many applications, including pattern recognition and image processing. We can develop an intuitive understanding of the problem by first considering the planar convex hull problem (for a finite set of points).

Consider a finite set of points in two dimensions (fig. 1a). Suppose that each of the points is a nail that has been hammered into a piece of wood. We stretch a rubber band so that it surrounds all of the nails (fig. 1b), and then release the rubber band, allowing it to contract around the nails (fig. 1c).

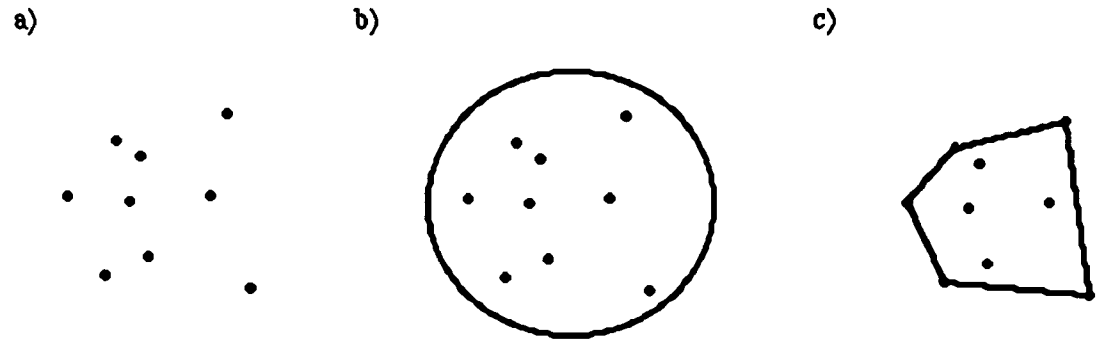


Figure 1: Intuitively constructing the convex hull of points in the plane

Now, we can easily visualize the convex hull of the set. It is precisely the area enclosed by the rubber band (fig. 1c). Note that the rubber band forms the boundary of the convex hull, and that the boundary has the shape of a convex polygon. Further note that some of the nails touch the rubber band and form vertices of the bounding polygon. Others do not, and are inside of the rubber band.

Finding the convex hull is precisely the problem of identifying these vertices (in clockwise or counter-clockwise order). We may note that, in general, the convex hull is an infinite set, which is described by a finite number of points: the vertices of the bounding polygon.

This informal definition of the convex hull may be generalized for points in arbitrary dimensions. It is, however, not as easy to envision as the two-dimensional case.

3. Formal problem definition

A set, **S**, of points in an Euclidean space is said to be *convex* if and only if for all pairs of points, p and q, in **S**, every point on the line segment connecting p and q is contained in **S**.

The convex hull of a set, **S**, is defined to be the smallest convex set that contains **S**. A convex hull is normally described by its boundary. The boundary of the convex hull of a set, **S**, is denoted **CH(S)**.

A common convex hull application requires finding the convex hull of a finite set of points lying in the plane. For this problem, **CH(S)** consists of a convex polygon. We describe **CH(S)** by an ordered set of vertices for the convex polygon which forms the boundary of the hull.

Note that there are both *primary* and *secondary vertices* for the polygon. A primary vertex of a polygon is a vertex that joins two sides of the polygon. Any point on the polygon which is not a primary vertex is a secondary vertex. Clearly, all primary vertices are necessary to define the polygon. Secondary vertices may or may not be included in the definition of the polygon, without changing the polygon.

In general, we are not particularly concerned with the inclusion of some secondary vertices in our definition of **CH(S)** since there is a very quick and simple

method for removing them. Our algorithm, in its simplest form, does not distinguish between primary and secondary vertices. However, we shall present the algorithm in a form which produces only primary vertices, since the algorithm is not significantly changed, but the corresponding proof of the algorithm is considerably simplified.

4. The algorithm

The algorithm is based on the idea of triangular expansion. We consider two primary vertices, p and q (fig. 2a). The two vertices define a line segment, **pq** (fig. 2b). This line segment forms the *baseline* of some triangle. We define the right and left sides of a baseline, **pq**, to be relative to baseline **pq** viewed vertically, with point p above point q. For simplicity, we assume that all points under consideration lie to the right of **pq**. We select a third primary vertex, r, to the right of the baseline, forming triangle **pqr** (fig. 2c).

We have now generated two new line segments, **pr** and **rq**. We repeat the above process, recursively, by considering the two segments as baselines (separately). This is the divide step. When we consider baseline **pr**, we select a primary vertex, s, to the right of the baseline. This forms triangle **prs** (fig. 2d). We end the recursive repetition whenever we discover a baseline for which no point lies to the right of the baseline (fig. 2e).

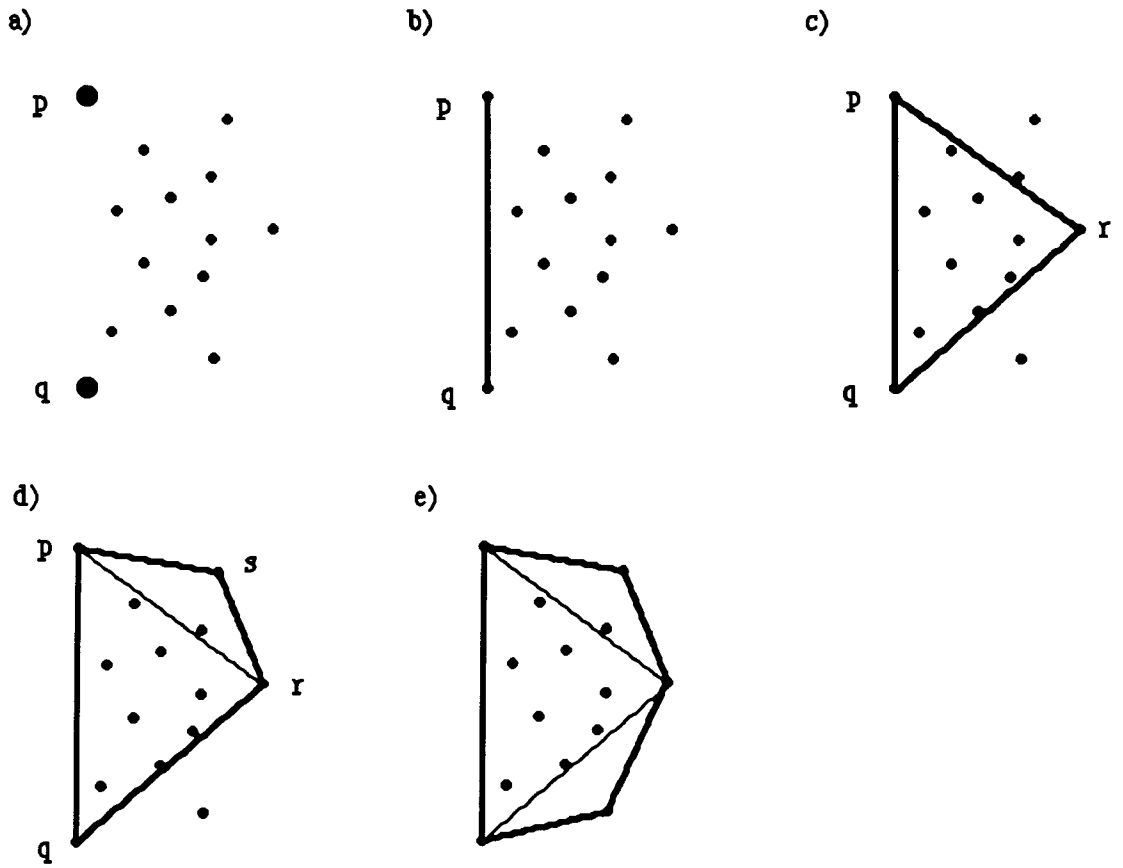


Figure 2: Graphic depiction of triangular expansion

Note that this process may be viewed as the (triangular) expansion of an approximate bounding polygon for the convex hull. Our initial approximation consists of the two-sided polygon defined by the two primary vertices initially selected (fig. 2b). At each step, we include one additional primary vertex into our approximate set of vertices, expanding the approximate bounding polygon (fig. 2c-e). When all primary vertices have been included, the approximate bounding polygon is the actual bounding polygon for the convex hull (fig. 2e).

Since the approximate bounding polygon always defines a subset of the convex hull, points internal to the approximate bounding polygon are internal to **CH(S)**. We may, therefore, ignore any points found to be internal to the approximate bound polygon. Note that we have done this, implicitly, via our process of selecting primary vertices only from the set of points to the right of the baseline under consideration. This is the prune-and-search element of the algorithm.

In general, it is difficult to select our two initial primary vertices so that all of the points under consideration lie to the right of the initial baseline. Rather, it is easier for us to select two convenient primary vertices, such as the vertices with maximum and minimum y-coordinates. As a result, we initially divide the problem into two subproblems: one corresponding to baseline **pq**, along with the points to its right, and one corresponding to baseline **qp**, along with the points to its right (which are to the left of baseline **pq** -- fig. 3).

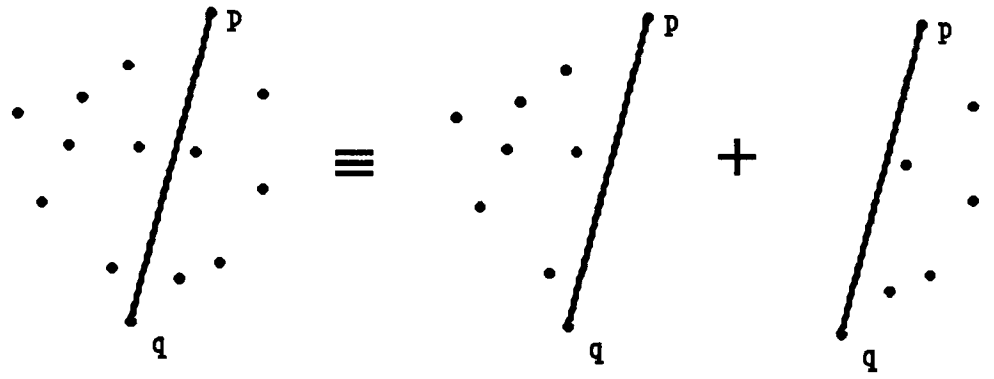


***Figure 3:** Dividing the initial problem into two subproblems*

We now need only consider the primary vertex selection process. For this purpose, we introduce a formal definition of the algorithm.

Consider finding the convex hull of a finite set of points, **S**, where every point in **S** lies in the plane and is described by an x-coordinate and a y-coordinate. We assume that there are at least two distinct points in **S**. We define algorithm `Hull` as follows:

```
InputSet(S);
Select(p,q,S);
Split(p,q,S,R,L);
OutputPoint(p);
QuickHull(p,q,R);
OutputPoint(q);
QuickHull(q,p,L)
```

`Select(p,q,S)` selects two points of **S** which are primary vertices of **CH(S)** (fig. 4). It is convenient to choose the points with the maximum and minimum y-coordinates (using x-coordinate values to break ties).

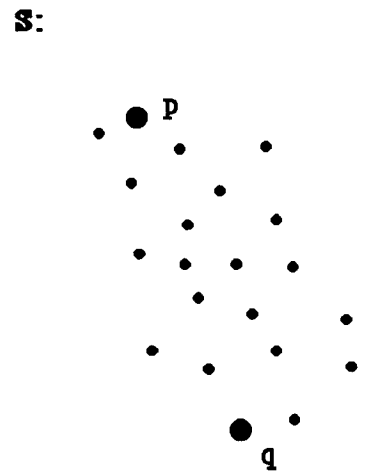


***Figure 4:** Selecting two initial primary vertices: Select(p,q,S)*

`Split(p,q,S,R,L)` splits **S** into three sets, **R**, **L**, and **O** (fig. 5). These sets contain the points of **S** to the right of baseline **pq**, to the left of baseline **pq**, and on baseline **pq**, respectively. (Note that the set **O** is not required by the algorithm. As such, points in the set **O** are discarded.)

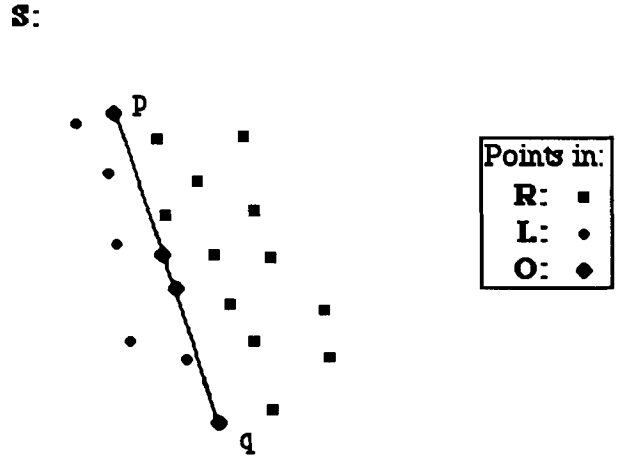


***Figure 5:** Splitting the data points into disjoint sets: Split(p,q,S,R,L)*

`QuickHull(p,q,S)` is defined as follows (note that this algorithm assumes that every point in **S** lies to the right of baseline **pq** -- fig. 7):

```
if not Empty(S) then begin
   r:=FarthestPoint(p,q,S);
   PruneAndSplit(p,q,r,S,U,L);
   QuickHull(p,r,U);
   OutputPoint(r);
   QuickHull(r,q,L)
end
```

The partitioning within `QuickHull` is depicted in figure 6.

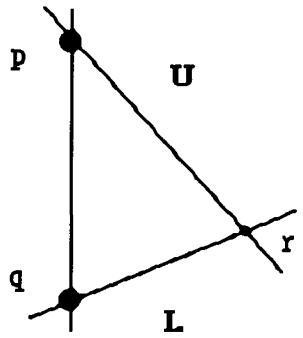


Figure 6: Partitioning within QuickHull

`FarthestPoint(p,q,S)` returns a primary vertex in **S** (fig. 7 -- note that **S** is assumed to lie to the right of baseline **pq**). It is convenient to choose the point with the largest perpendicular distance from baseline **pq** (using the parallel projection of the points on the baseline to break ties).

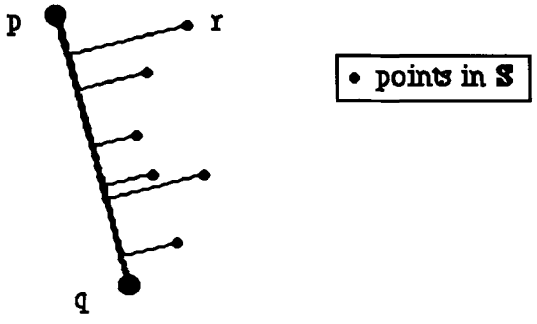


Figure 7: Selecting the farthest point from baseline pq: r:=FarthestPoint(p,q,S)

`PruneAndSplit(p,q,r,S,U,L)` splits **S** into two sets, **U** and **L**, which contain the points to the right of baselines **pr** and **rq**, respectively. Points internal to triangle **pqr** are pruned.

Algorithm `Select` may be implemented using a simple search. Algorithm `Split` and function `FarthestPoint` may be implemented using standard geometric techniques. Algorithm `PruneAndSplit` may be implemented using `Split`.

Our algorithm `Hull` is now complete. The primary vertices of **CH(S)** are output in proper order. Listing 1 (at the end of this paper) is a Pascal implementation of the complete `Hull` algorithm.

5. Correctness proof

We shall prove that our algorithm correctly identifies the convex hull in three parts. First, we shall prove that our algorithm outputs only primary vertices of **CH(S)**. Next, we shall prove that all primary vertices of **CH(S)** are output. And finally, we shall prove that the vertices are output in proper order.

Theorem 1: Algorithm `Hull` outputs only primary vertices of **CH(S)**.

Proof:

Define a *supporting line* of a set **S** to be a straight line that has at least one point in common with **S** and all vertices of **S** lying on the same side (inclusive) of the line [2]. Clearly, any point of **S** which lies on a supporting line of **S** is in **CH(S)**.

Consider a supporting line, L, of **S** (fig. 8). Assume that n points in **S** lie on L. Clearly, the two (end) points that form the longest line segment along L are primary vertices [10].

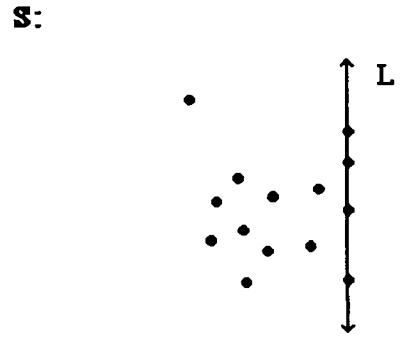


Figure 8: A set, S, and supporting line, L

Algorithm `Hull` outputs points p and q. The points p and q are chosen to have the minimum and maximum y-coordinates. Therefore, each point defines a horizontal line through the point which constitutes a supporting line of **S**. Therefore, both points are in **CH(S)**. Furthermore, we use the x-coordinates to insure that both points are endpoints. Therefore, both points are primary vertices.

Algorithm `QuickHull(p,q,R)` may be shown to output only primary vertices in **CH(S)** by strong mathematical induction on the cardinality of **R**. We shall use the following invariant for algorithm `QuickHull`:

Invariant QH:
1) p and q are primary vertices in **CH(S)**
2) **R** is a subset of **S** and lies to the right of baseline **pq**
3) the set **S** - **R** lies to the left of baseline **pq**

Clearly, algorithm `Hull` establishes this invariant for both of its calls to `QuickHull`.

Basis step: If $|\mathbf{R}| = 0$ then `QuickHull` outputs no vertices. This vacuously establishes the conclusion.

Inductive step: Assume the conclusion for $0 \leq |\mathbf{R}| \leq n$. Consider the case of $|\mathbf{R}| = n+1$. Assume the invariant, QH. `QuickHull` selects and outputs a point, r, in **R**, furthest from baseline **pq** (fig. 9). Let L be a line drawn parallel to the baseline and through point r. Clearly, L is a supporting line of **R**, where **R** lies to the left of L. Furthermore, the baseline lies to the left of L, and **S** - **R** lies to the left of the baseline. Therefore, **S** lies to the left of L, and L is a supporting line of **S**. Therefore, point r is in **CH(S)**. Furthermore, we choose point r to be an endpoint. Therefore, point r is a primary vertex of **CH(S)**.

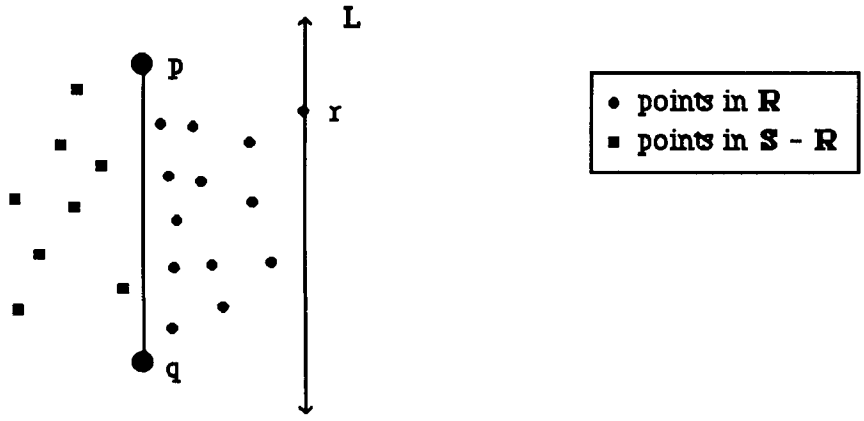


***Figure 9:** Selection of primary vertex r*

$|\mathbf{R} - \{r\}| = n$. Therefore, no matter how we partition the set, our recursive calls to `QuickHull` will consider sets with cardinality $\leq n$. We, therefore, need only prove that our recursive calls to `QuickHull` maintain the invariant QH.

Let us refer to the two recursive calls to `QuickHull`, for the baselines **pr** and **rq**, as case 1 and case 2, respectively (fig. 10).

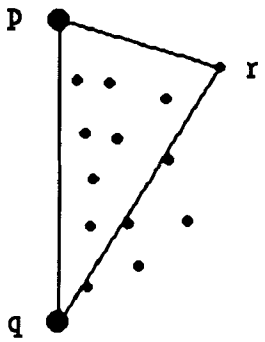


***Figure 10:** Constructing recursive calls to QuickHull*

The maintenance of part 1) of QH is trivial for both cases.

The maintenance of part 2) of QH is trivial by our pruning-and-splitting algorithm. (Note that no point of **R** - {r} lies both to the right of baseline **pr** and to the right of baseline **rq**. This is a result of the fact that L is a supporting line of **R**.)

We shall prove that part 3) of QH is maintained for case 1. The proof for case 2 is analogous and trivially follows from the proof for case 1.

Consider case 1 (fig. 11). We call `QuickHull(p,r,U)`. That **R** - **U** lies to the left of baseline **pr** is trivial by our splitting algorithm. We, therefore, need only show that **S** - **R** lies to the left of the baseline. We show this by contradiction.

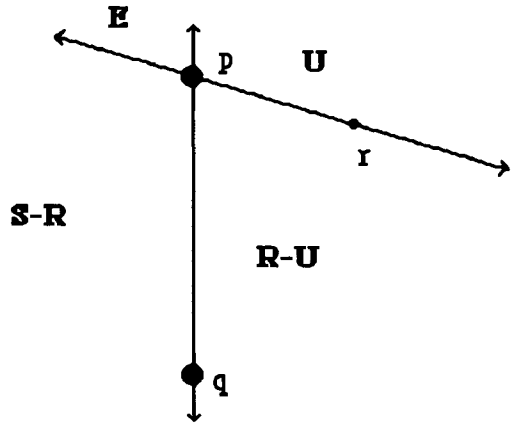


Figure 11: Partitioning of sets for case 1

Assume there is some point, e, in **S** - **R** that is not to the left of the baseline. This point is in set **E** of fig. 11. However, if e is in **E** then, clearly, point p is not a primary vertex (fig. 12). This is a contradiction of part 1) of QH. Therefore, set **E** is empty, and **S** - **R** lies to the left of baseline **pr**.

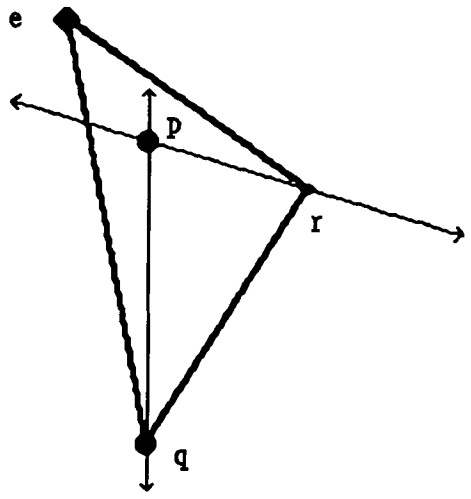


Figure 12: Considering a point, e, in E

Therefore, invariant QH is maintained, and all points output are primary vertices of **CH(S)**.

Theorem 2: Algorithm `Hull` outputs all primary vertices of **CH(S)**.

Proof:

The proof is trivial using strong mathematical induction. We provide an outline for the proof.

Clearly, algorithm `Hull` and every call to `QuickHull` results in the removal of a finite number of points (≥0) from the original set, **S**. Furthermore, there are only two ways of removing points: recognizing that a point is a primary vertex, in which case the point is output, and recognizing that a point is internal to the polygon formed by the currently known primary vertices, in which case the point is not a primary vertex. The recursive calls to `QuickHull` end only when all points of **S** have been removed. Since all primary vertices are output when removed, the algorithm can not terminate until all primary vertices have been output.

Furthermore, it is clear that the algorithm terminates since we may bound the number of recursive calls to `QuickHull` from `QuickHull(p,q,R)` by 2*|**R**|. Therefore, algorithm `Hull` outputs all primary vertices of **CH(S)**.

Theorem 3: Algorithm `Hull` outputs the primary vertices of **CH(S)** in proper order.

Proof:

The proof is trivial by strong mathematical induction.

Consider a call `QuickHull(p,q,R)`. We use strong mathematical induction on the cardinality of **R** to show that `QuickHull` outputs the primary vertices in **R** in proper order.

Basis step: If |**R**| = 0 then no points are output and the conclusion is vacuously established.

Inductive step: Assume the conclusion for $0 \leq |\mathbf{R}| \leq n$. Consider the case of |**R**| = n+1. As shown previously, a primary vertex, r, is selected, and both recursive calls to `QuickHull` involve sets with cardinality ≤ n. Therefore, both the upper and lower calls (cases 1 and 2, respectively) result in outputting the two subsets of primary vertices in proper order. Clearly, primary vertex r must be output between the two calls in order to maintain proper order. Therefore, algorithm `QuickHull` outputs the primary vertices in proper order.

Furthermore, given this proper ordering of output vertices by algorithm `QuickHull`, it is clear that algorithm `Hull` will result in a proper ordering of output vertices.

We, thereby, complete our proof of algorithm `Hull`.

6. Time complexity

In general, the time complexity of geometric algorithms is difficult to analyze as characterization of the data is not simple [11]. We offer the following simple analysis of the time complexity of this algorithm. We consider data consisting of n points, m of which are actually primary vertices. Note that the searching and splitting operations of algorithm `Hull` will always have a time complexity of O(n).

Case 1: (almost) every point a primary vertex:

This is the case where m is O(n). Clearly, this is the worst case for any convex hull algorithm, since all primary vertices must be output. We consider two subcases: the worst subcase, in which the points are distributed such that our partitioning of points (almost) always generates one empty call to `QuickHull` (an empty call is one in which no points lie to the right of the given baseline), and the best (expected) subcase, in which the partitioning of points is (reasonably) even.

The worst subcase:

The worst subcase for case 1 is clearly the worst possible case for the algorithm. Clearly, we require O(n) calls to `QuickHull`, each of which requires an O(n) computation, on average. We, therefore, have a worst case time complexity of $O(n^2)$. A sample point distribution which achieves the worst-case time complexity is shown in fig. 13.

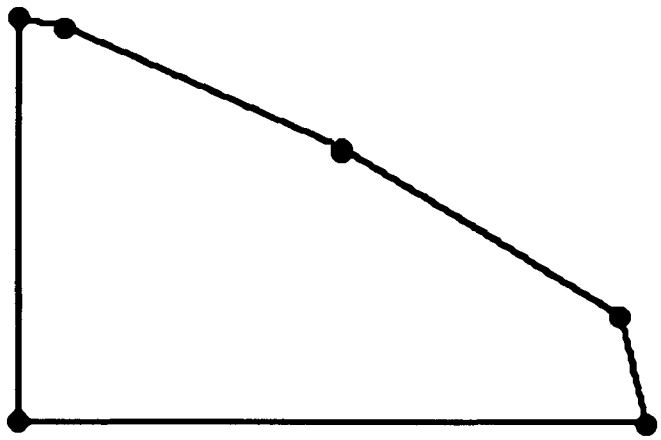


***Figure 13:** A worst-case point distribution*

The best (expected) subcase:

Here, we assume that our partitioning of the data within `QuickHull` is (reasonably) even. Clearly, by the divide-and-conquer nature of the algorithm, we have a time complexity of O(n log n) for this case.

Case 2: (reasonably) uniformly distributed points:

This is the (expected) m << n case. For this case we also have two subcases: a worst subcase, in which the primary vertices are distributed such that our partitioning of

points (almost) always generates one empty call, and the best (expected) subcase, in which the partitioning of points is (reasonably) even.

The worst subcase:

We clearly have O(m) calls to `QuickHull`, each of which requires (no more than) an O(n) computation. We, therefore, have a time complexity of O(mn) for this case.

The best (expected) subcase:

For our analysis, we assume that most of the points internal to the boundary of the convex hull are removed from the computation in the very early stages algorithm. This seems reasonable for problems involving reasonably uniform distributions of points since the early stages of the algorithm account for the vast majority of the area from which internal points are pruned. We, therefore, assume that after the early stages, O(m) points remain to be considered.

After the early stages, we clearly have a time complexity of O(m log m) by the divide-and-conquer nature of the algorithm. The early stages have a time complexity of O(n). The overall time complexity is, therefore, O(n + m log m). If we assume that m is $O(n^{1/3})$ (which is a reasonable upper bound for circular distributions, rectangular distributions, and several other distributions [11] as well as for a number of non-uniform distributions [7]), then we have a time complexity of O(n) for the algorithm.

Performance measurements:

In order to provide some verification of our expected-case time complexity results, we performed some simple performance experiments. We applied the algorithm (using the implementation of listing 1) to randomly generated sets of points, uniformly distributed in both rectangular and circular configurations. For each data set size, we randomly generated five rectangular distributions and five circular distributions. Tables 1 and 2 show the average number of operations performed by the algorithm for the rectangular and circular distributions, respectively. Graph 1 shows the average number of operations versus the total number of points, plotted for both distributions.

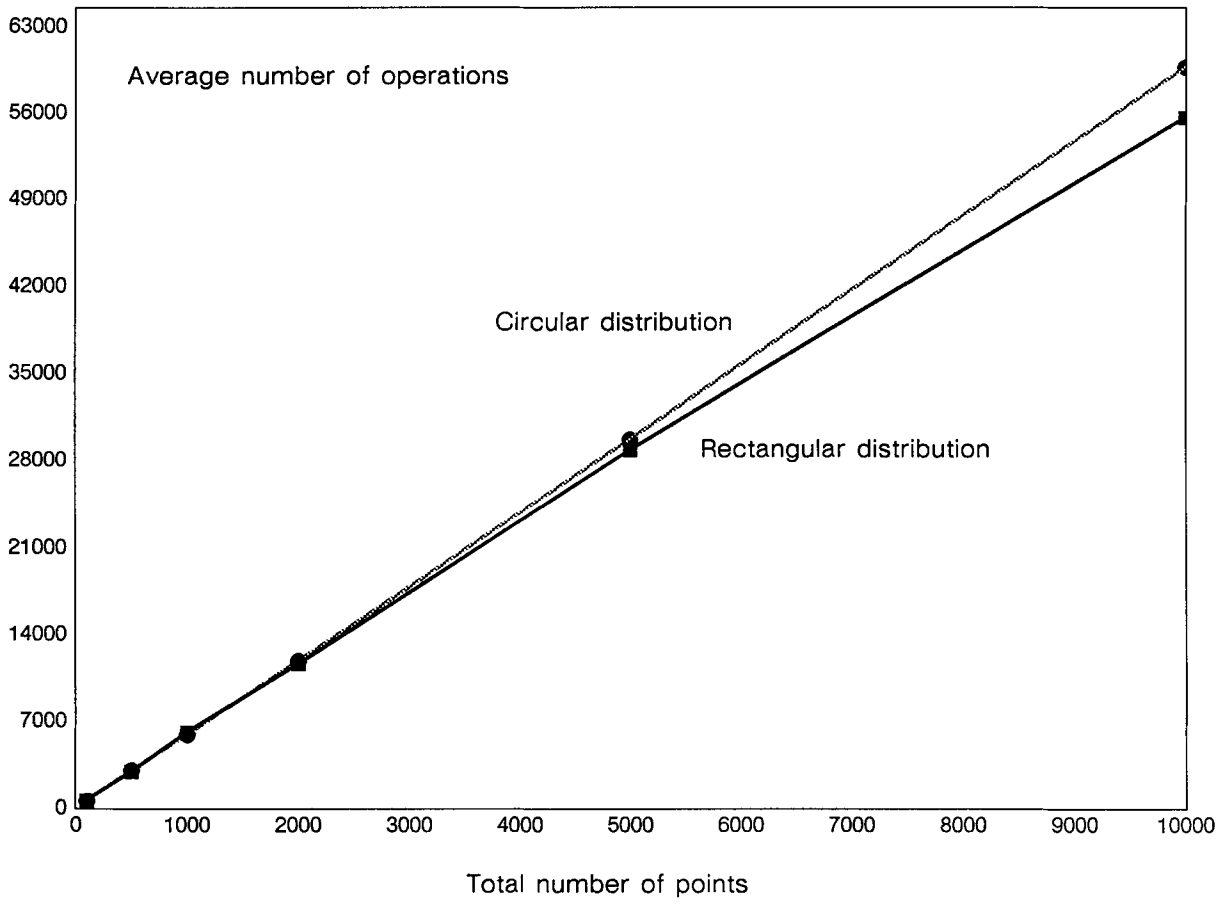
Points	10,000	5000	2000	1000	500	100
Operations	55,598.2	28,820.4	11,597.6	6099.0	2918.4	616.4
Vertices	22.4	20.6	20.6	16.4	16.0	11.4

***Table 1:** Average number of operations for uniform rectangular distributions*

Points	10,000	5000	2000	1000	500	100
Operations	59,675.2	29,713.2	11,870.2	5865.4	3024.8	600.6
Vertices	43.4	37.2	28.8	24.4	19.6	14.2

***Table 2:** Average number of operations for uniform circular distributions*

These performance figures show an essentially linear increase in the number of operations performed as the number of data points is increased. The figures, therefore, tend to support our expected-case time complexity analysis.



***Graph 1:** Average number of operations for the two distributions*

7. Space complexity

Consider a convex hull problem consisting of n data points, m of which are actually hull vertices. Every non-empty call to `QuickHull(p,q,R)` results in a partitioning of **R** into three sets (one for each of the two recursive calls to `QuickHull`, as well as one for the internal points), without duplication. Furthermore, since O(m) calls are required for the complete algorithm, no more than O(m) activation records will exist at any one time. Therefore, the activation records collectively contain no more than O(n) pieces of information, and an efficient implementation of the algorithm will require only O(n) memory.

Note that the implementation of listing 1 depends upon the automatic reclamation and re-use of storage space (released by the removal of points from a set) in order to achieve the above space complexity. This is a result of the abstraction of the set data type. If necessary, the data type abstraction may be removed in order to allow manual reclamation and re-use of storage space.

8. Special cases

There are three special cases for the convex hull problem. They correspond to finding the convex hull of an empty set, a non-empty zero-dimensional set, and an one-dimensional set. We have already seen that our algorithm works for the special case of an one-dimensional set. It is trivial to include provisions for the other two special cases in our algorithm. Note also that our algorithm does not depend on distinctness of points in the data set, S (as long as there are at least two distinct points).

9. Conclusion

We have presented a proof of correctness for a simple QuickHull algorithm. The algorithm has a worst case time complexity of $O(n^2)$, but an expected time complexity of O(n). The algorithm requires only O(n) memory.

We had originally hoped that this algorithm would generalize to higher-dimensional convex hull problems. However, it has become clear that it does not do so easily. The main problem in generalizing appears to be the inability to maintain an invariant clause similar to part 3) of the invariant QH, used for the proof of the planar algorithm.

The QuickHull algorithm works by recursively breaking a planar convex hull problem into two independent subproblems. In general, however, we cannot easily divide a non-planar convex hull problem into independent subproblems. The key to finding a generalization of this algorithm to higher-dimensional problems is, therefore, finding a more general technique for dividing convex hull problems into independent subproblems.

Acknowledgements

I would like to acknowledge the contributions of Per Brinch Hansen. His considerable patience and plentiful suggestions have greatly improved this paper.

References

1. C. A. R. Hoare, "Quicksort," *Computer Journal*, **5**, 10-15 (1962).

2. D. T. Lee and F. P. Preparata, "Computational Geometry--A Survey," *IEEE Transactions on Computers*, **C-33**, 1072-1101 (1984).

3. R. L. Graham, "An efficient algorithm for determining the convex hull of a finite planar set," *Information Processing Letters*, **1**, 132-133 (1972).

4. R. A. Jarvis, "On the identification of the convex hull of a finite set of points in the plane," *Information Processing Letters*, **2**, 18-21 (1973).

5. F. P. Preparata and S. J. Hong, "Convex hulls of finite sets of points in two and three dimensions," *Communications of the ACM*, **20**, 87-93 (1977).

6. J. L. Bentley and M. I. Shamos, "Divide and conquer for linear expected time," *Information Processing Letters*, **7**, 87-91 (1978).

7. W. F. Eddy, "A new convex hull algorithm for planar sets," *ACM Transactions on Mathematical Software*, **3**, 398-403 (1977).

8. A. Bykat, "Convex hull of a finite set of points in two dimensions," *Information Processing Letters*, **7**, 296-298 (1978).

9. P. J. Green and B. W. Silverman, "Constructing the convex hull of a set of points in the plane," *Computer Journal*, **22**, 262-266 (1979).

10. F. P. Preparata and M. I. Shamos, *Computational Geometry*, Springer-Verlag (1985).

11. R. Sedgewick, *Algorithms*, second edition, Addison-Wesley (1988).

***Listing 1:** A Pascal implementation of the Hull algorithm*

```
PROGRAM Hull(Input, Output);

{Implements a divide-and-conquer, prune-and-search, triangular expansion
 algorithm to find the convex hull of points in the plane.}

TYPE Point=RECORD
             x,y:REAL
           END;
     SetElement=RECORD
                  info:Point;
                  next:^SetElement
                END;
     PointSet=RECORD
                front,rear,current:^SetElement
              END;

VAR S,R,L:PointSet;
    p,q:Point;

{PointSet operations}

  PROCEDURE NewSet(VAR S:PointSet);
  {Creates a new PointSet S}
  BEGIN
    New(S.front);
    S.rear:=S.front;
    S.current:=S.front
  END;

  PROCEDURE Reset(VAR S:PointSet);
  {Moves the current position within PointSet S to the beginning}
  BEGIN
    S.current:=S.front^.next
  END;
```

```
PROCEDURE Include(p:Point;VAR S:PointSet);
{Includes a point p in PointSet S}
VAR element:^SetElement;
BEGIN
  New(element);
  element^.info:=p;
  element^.next:=nil;
  S.rear^.next:=element;
  S.rear:=element
END;

FUNCTION CurrentPoint(S:PointSet):Point;
{Returns the value of the current point in PointSet S}
BEGIN
  CurrentPoint:=S.current^.info
END;

PROCEDURE Advance(VAR S:PointSet);
{Advances the current pointer of PointSet S}
BEGIN
  S.current:=S.current^.next
END;

FUNCTION Empty(S:PointSet):BOOLEAN;
{Determines if PointSet S is empty}
BEGIN
  Empty:=(S.front^.next=nil)
END;

FUNCTION MorePoints(S:PointSet):BOOLEAN;
{Determines if the current pointer of PointSet S is at the end}
BEGIN
  MorePoints:=(S.current<>nil)
END;

PROCEDURE Remove(VAR p:Point;VAR S:PointSet);
{Removes and returns a point p from (non-empty) PointSet S}
VAR element:^SetElement;
BEGIN
  element:=S.front^.next;
  p:=element^.info;
  s.front^.next:=element^.next;
  Dispose(element)
END;

PROCEDURE Eliminate(VAR S:PointSet);
{Eliminates PointSet S}
VAR pt:Point;
BEGIN
  WHILE NOT Empty(S) DO BEGIN
    Remove(pt,S)
  END
END;
```

```
{QuickHull IO routines}

  PROCEDURE InputSet(VAR data:PointSet);
  {Inputs the original set of data points}
  VAR n,i:INTEGER;
      p:Point;
  BEGIN
    ReadLn(n);
    NewSet(data);
    FOR i:=1 TO n DO BEGIN
      ReadLn(p.x,p.y);
      Include(p,data)
    END
  END;

  PROCEDURE OutputPoint(p:Point);
  {Outputs point p}
  BEGIN
    WriteLn(p.x,p.y)
  END;

{Algorithm Tools}

  FUNCTION ScaledDistance(u,v,p:Point):REAL;
  {Finds the distance between vector uv and point p, scaled by
   the length of vector uv}
  BEGIN
    ScaledDistance:=(v.x-u.x)*(p.y-u.y)-(v.y-u.y)*(p.x-u.x)
  END;

  FUNCTION Direction(u,v,p:Point):REAL;
  {Finds the (scaled) signed distance between vector uv and point p}
  BEGIN
    Direction:=(v.x-u.x)*(p.y-u.y)-(v.y-u.y)*(p.x-u.x)
  END;

  FUNCTION Projection(u,v,p:Point):REAL;
  {Finds the parallel projection of point p onto vector uv, scaled
   by the length of vector uv}
  BEGIN
    Projection:=(v.x-u.x)*(p.x-u.x)+(v.y-u.y)*(p.y-u.y)
  END;
```

```
PROCEDURE Select(VAR p,q:Point;S:PointSet);
{Selects two primary vertices p and q from a (non-empty) set S}
VAR pt:Point;
BEGIN
  Reset(S);
  q:=CurrentPoint(S);
  p:=q;
  Advance(S);
  WHILE MorePoints(S) DO BEGIN
    pt:=CurrentPoint(S);
    IF (pt.y<q.y) OR ((pt.y=q.y) AND (pt.x<q.x)) THEN BEGIN
      q:=pt
    END
    ELSE IF (pt.y>p.y) OR ((pt.y=p.y) AND (pt.x>p.x)) THEN BEGIN
      p:=pt
    END;
    Advance(S)
  END
END;

PROCEDURE Split(p,q:Point;VAR S,R,L:PointSet);
{Splits S into sets R and L, to the right and left of baseline pq,
 respectively}
VAR dir:REAL;
    pt:Point;
BEGIN
  NewSet(R);
  NewSet(L);
  WHILE NOT Empty(S) DO BEGIN
    Remove(pt,S);
    dir:=Direction(p,q,pt);
    IF dir>0.0 THEN BEGIN {point is on the right}
      Include(pt,R)
    END
    ELSE IF dir<0.0 THEN BEGIN {point is on the left}
      Include(pt,L)
    END
  END
END;

PROCEDURE PruneAndSplit(p,q,r:Point;VAR S,U,L:PointSet);
{Selects the upper and lower outside sets, U and L, for baseline pq
 and farthest-point r}
VAR internal:PointSet;
BEGIN
  Split(p,r,S,U,internal);
  Split(r,q,internal,L,internal);
  Eliminate(internal)
END;
```

```
  FUNCTION FarthestPoint(p,q:Point;S:PointSet):Point;
  {Finds a point in (non-empty) S farthest from baseline pq}
  VAR maxdist,dist:REAL;
      u,r,pt:Point;
  BEGIN
    maxdist:=0.0;
    Reset(S);
    WHILE MorePoints(S) DO BEGIN
      pt:=CurrentPoint(S);
      dist:=ScaledDistance(p,q,pt);
      IF dist>maxdist THEN BEGIN
        maxdist:=dist;
        r:=pt
      END
      ELSE IF dist=maxdist THEN BEGIN
        {compare parallel projections along the baseline}
        IF Projection(p,q,pt)>Projection(p,q,r) THEN BEGIN
          maxdist:=dist;
          r:=pt
        END
      END;
      Advance(S)
    END;
    FarthestPoint:=r
  END;

{Main Algorithm}

  PROCEDURE QuickHull(p,q:Point;VAR S:PointSet);
  {Finds the convex hull of S given that p and q are primary vertices
   and S lies to the right of baseline pq}
  VAR r:Point;
      U,L:PointSet;
  BEGIN
    IF NOT Empty(S) THEN BEGIN
      r:=FarthestPoint(p,q,S);
      PruneAndSplit(p,q,r,S,U,L);
      QuickHull(p,r,U);
      OutputPoint(r);
      QuickHull(r,q,L)
    END
  END;

BEGIN {Hull}
  InputSet(S);
  Select(p,q,S);
  Split(p,q,S,R,L);
  OutputPoint(p);
  QuickHull(p,q,R);
  OutputPoint(q);
  QuickHull(q,p,L)
END.
```