Convex Hull Algorithms

Eric Eilberg

Denison University

Abstract

This paper discusses the origins of the convex hull, and the development of algorithms designed to solve them. We start with the most basic brute force method, Graham’s Scan, progressing to the Jarvis March, then to Quick-hull and convex hulls in N-space. The latter part of the paper will briefly touch on a simple approximation algorithm.

Introduction

Convex hulls are one of the most fundamental constructions of computational geometry. They are utilized in many fields for pattern recognition, regression, collision detection, estimation, spectrometry, cartography, topology, and as a preliminary step to solve many seemingly unrelated problems. If you have ever had excel put a regression line into your graph, you have seen a convex hull algorithm in action. To create that regression line, excel first finds the convex hull of your data points. Then, it finds the midpoint distance between the top and bottom walls for every x value, and puts a dot there. These dots form the regression line. The convex hull of a set S of points in Euclidian space is defined to be the smallest convex polygon that contains all the
elements of S. If you were to shrink wrap S, the shape the plastic shrink wrap forms around S is the convex hull. Physically, this is not a very complex situation to understand.

Fig. 2

Similar to sorting, there exist many different convex hull algorithms. Also similar to sorting, some are better than others.

Actually, convex hull algorithms share a lot with sorting algorithms. They can actually be transformed into sorting algorithms in linear time. Assume there is a set S of N numbers, n₁, n₂ … nₙ where all n > 0. To sort this list using a convex hull algorithm, just create the set T, where T contains the point (n, n²) for all n ∈ S, then run a convex hull algorithm on T. Since all the points exist on the curve x², we know the shape of the resulting convex hull (see Fig. 3). Traverse that resultant hull starting with the lowest point and moving counter clockwise along the hull (called the standard form) and the output of the points.

The result is the x coordinates in sorted order. The set of x coordinates is the exact set we were trying to sort. We now have in our possession the knowledge to create a convex hull
sorting algorithm. To use a convex hull sort would not be the optimal solution, but the possibility exists. So why would the fact that it can sort be of any use to a computer scientist?

Preparata and Shamos [1] show that “If a problem A is known to require $T(N)$ time and A is $\tau(N)$-transformable to B ($A \preceq_{\tau(N)} B$), then B requires at least $T(N) - O(\tau(N))$ time.” ([1], pg 29) We know that sorts are linearly transformable to convex hulls, as shown above. Combine this with the knowledge that the time complexity of sorting has a lower bound of $\Omega(n \lg n)$, and we can safely assert that the lower bound of the convex hull problem is also $\Omega(n \lg n)$. Knowing the lower bound of the problem allows us to judge the efficiency of algorithms for convex hulls.

**Graham Scan**

The most basic of these algorithms utilizes the idea that all of the points on the convex hull will be comprised of so called extreme points. A point $p \in S$ is said to be an extreme point if and only if there is no possible hull in $S$ that contains $p$ within it. Combine this and the proven theorem that given any set $S$ of $N$ points, there are at most $N^3$ triangles that can be formed in $S$ ([1], pg 104). These triangles will be the hulls used to determine if points are extreme or not. It takes constant time to determine if a point exists within one of these triangles and there are $N$ points, so it takes $O(N^4)$ to find the set $E$ of extreme points. Next, find a point $Q$ that does not exist in $E$. Sort $E$ by polar angle around $Q$, and the convex hull appears. This is not efficient, but it works.

In 1972, R. L. Graham published an algorithm called the Graham Scan. [2] This scan was the first convex hull algorithm to run in $O(n \lg n)$ time. The algorithm discussed
above used a Cartesian coordinate system (based in x and y), but Graham found a much more efficient way using polar coordinates. (Polar coordinates are defined in terms of \( r \) and \( \theta \), such that a point at \((r, \theta)\) exists a distance \( r \) from the origin at an angle of \( \theta \) from the positive x axis. The Cartesian point \((1,1)\) exists at \((\sqrt{2}, 45^\circ)\) in polar.)

The basic idea behind a Graham scan is pretty simple. Start at \( P_1 \), the lowest point (leftmost such point in case of tie), and then scan in polar (by increasing \( \theta \)) until a point \( P_2 \) is found. Scan from \( P_2 \) and find \( P_3 \). Compute angle \( P_1 P_2 P_3 \). If the angle is greater than 180\(^\circ\), it means the hull is concave and not convex. This is considered a right turn, and if the angle is less than 180\(^\circ\) it is considered a left turn. Therefore the point \( P_2 \) does not exist in the convex hull. Continue this process moving along the vertices and the points left over are the convex hull’s vertices in standard form. See Figure 4 for a visual example of a Graham scan.

**Fig. 4**

- **Red Lines** = Polar Lines to Points
- **Black Lines** = Currently Considered Edges
- **Blue Lines** = Convex Hull Edges

A simple example of how the Graham Scan works, showing each step in the algorithm.

Notice that in the third step, the addition of that vertex would cause a concave hull, and therefore it is not added to the final solution.
The Graham scan is a great concept but hard to implement on conventional computers. Computers do not understand polar coordinate systems natively, and this adds overhead to an otherwise simple algorithm. To convert the Graham scan to work in the Cartesian plane, the points must be sorted lexicographically by polar angle. If there are three collinear points, disregard the middle point. It cannot be part of the final convex hull so there is no point in even considering it. Once the list is sorted, run Graham’s algorithm on it. The pseudocode for Graham’s algorithm looks like this:

1. Let p₀ be the first point (lowermost, left if tie)
2. Let { p₁, p₂, p₃ . . . pₙ } be the rest of the points in lexicographic polar sorted order
3. Stack.push(p₀)
4. Stack.push(p₁)
5. Stack.push(p₂)
6. for( int i=3; i<=m; i++ )
7.   while angle from pᵢ, stack.top, and stack.second is a non-left turn
8.        Stack.pop( )
9.     Stack.push(pᵢ)
10. return the stack

Now, consider the following loop invariant: at the start of the for loop, the stack contains the convex hull of the subset of points from 0 to i-1, sorted in counterclockwise order. Proof that the Graham Scan works follows:

Initialization:
At the initialization of the loop, the stack contains \( p_0 \), \( p_1 \) and \( p_2 \). According to the loop invariant, the stack should contain the convex hull of \( p_0 \) to \( p_{i-1} \), which in this case is \( p_0 \) to \( p_2 \). The set of points \( p_0 \), \( p_1 \) and \( p_2 \) form a convex hull for points \( p_0 \) to \( p_2 \).

Maintenance:
Before each iteration of the loop, \( p_{i-1} \) is the top element on the stack. Let \( p_x \) be the second from top element (not necessarily \( p_{i-2} \)), and \( p_{x-1} \) be the point below \( p_x \). If we return to the instant before we pushed \( p_i \) during the last iteration of the loop, we know that the stack contains exactly the convex hull for \( p_0 \) to \( p_{i-1} \), given by the loop invariant. It is known that the angle formed by \( p_i \), \( p_x \) and \( p_{x-1} \) takes a left turn due to the while loop. Had the angle been anything but a left turn, \( p_x \) would have been popped and not allowed to remain on the stack. Therefore we have the situation shown in Figure 5. It is known that the point \( p_i \) exists within the shaded area (but not on the boundary) due to the geometry behind a left turn angle. Since the stack contains the hull for \( p_0 \) to \( p_{i-1} \) and then has \( p_i \) added to it, for the loop invariant to hold true the stack + \( p_i \) must be the convex hull for \( p_0 \) to \( p_i \). Using geometry, adding any point in the shaded area to the polygon will result in a convex polygon. Since we know \( p_i \) exists within the shaded area and we know that adding a point in the shaded area to the polygon will result in a convex polygon, we know that the stack is indeed the convex hull for the set of points \( p_0 \) to \( p_i \) at the end of the loop. The variable \( i \) is incremented before the start of the next iteration, and therefore the stack contains the convex hull for points \( p_0 \) to \( p_{i-1} \).

Termination:
When the loop terminates, \( i = m+1 \). The invariant claims the stack will hold the convex hull for \( p_0 \) to \( p_m \). Since there are only \( m \) vertices, this means that the stack contains the convex hull for all \( p \in P \).

Graham’s scan was the first algorithm proposed that could run in \( O(n \lg n) \) time, and was therefore optimal. To see this, note that line 1 traverses the whole list to determine the lowest point, taking \( O(n) \) time. Line two is a sort, running in optimal \( O(n \lg n) \) time. Lines 3, 4 and 5 take constant time each. The for loop executes \( n-3 \) times. The push command is constant, so the time complexity within the for loop comes solely from the while loop. Each point is pushed onto the list exactly once, and therefore can be popped exactly once. We also know that the stack will always contain at least 3 points since it always contains a convex hull. Therefore the while loop can execute a maximum of \( n-3 \) times. Combine all this, and the algorithm runs in \( O(n \lg n) \).

Graham’s scan is quite effective in 2 dimensions, but the transition to three dimensions is extremely difficult and moving past three into \( n \) dimensions is impossible. When moving to three dimensions, spherical coordinates can be utilized in place of polar. (Spherical coordinates define a point in terms of \( r \), \( \theta \) and \( \phi \). The \( r \) and \( \theta \) work the same as polar, and the \( \phi \) dimension is the angle from the origin’s xy plane to the point). The added dimension of \( \phi \) makes the algorithm messy and quite difficult to understand; the scan loses its elegance.

**Jarvis March**

Another quite efficient algorithm for dealing with convex hulls was developed by Jarvis in 1973 [2]. Rather than creating the convex hull of all points up to the current one
and ignoring all points beyond, the points touched during the Jarvis March will all be a part of the final convex hull. Recall wrapping Christmas presents around the holidays. Once you have the wrapping paper cut to the right size, you begin to wrap by taping one edge of the paper to the bottom of the gift. Then, you wrap the paper around the gift by holding the paper tight and rotating the gift. When you reach the start point again you are done wrapping in that one dimension, and you deal with the other dimension, in this case the two remaining edges of the box. Jarvis uses the same idea in his algorithm.

The Jarvis march starts at the lowermost point \( p_0 \) (leftmost if a tie) and finds the point with the least polar angle greater than zero compared to \( p_0 \). This continues until you reach the \( p_h \), apex of the hull, at which point you must change your focus to be the greatest polar angle less than 0 (smallest absolute value). These angles are represented in figure 6 as the red arrows. Notice that starting at \( p_0 \), the red
arrow between the x axis and the line $p_0p_1$ will be smaller than the one between the x axis and the line $p_0p_4$. The chain of points from $p_0$ to $p_h$ found in the first step of the algorithm is the first chain, and the ones found in the second part are called the second chain as shown in figure 7. Combine these two chains and you have the convex hull for the entire set, and it is even sorted in lexicographic order due to the nature of the algorithm. The pseudocode is below:

1. Find $p_0$, the lowest (and leftmost if a tie) point
2. $p_x = p_0$
3. do {
   4. find polar angle of all points relative to $p_x$
   5. if in first string
   6. find $p_y$, the point with smallest angle $\geq 0$
   7. if no such point exists, set flag to second string
   8. if in second string
   9. find $p_y$, the point with largest angle $\leq 0$
   10. add $p_x$ to ConvexPointSet
   11. set $p_x = p_y$
5. } while ( $p_x$ not equal to $p_0$ )
13. return ConvexPointSet

Line 1 takes $O(n)$ time to find the lowest point. Line 2 is $O(1)$. Line 4 takes $O(n)$ time to determine the angle between all n points. The rest of the interior do loop take $O(1)$ time when combined with a smartly written “find polar angle.” The do loop itself execute $h$ times, where $h$ is the number of vertices in the convex hull of the set. Therefore
the running time of the do loop is \( O(hn) \). Combine all of this, and we find that the Jarvis March executes in a time of \( O(hn) \). In the worst case this can be \( O(N^2) \), if all the vertices are in the convex hull. When \( h \) is known beforehand to be small in comparison to the number of points, this algorithm runs quite efficiently though.

Similar to the Graham scan, the Jarvis March also runs into problems when dealing with multidimensional spaces. Spherical coordinates enable the algorithm to deal with another dimension, but the also complicate the process and erase the efficiency of the algorithm.

**Quick-hull**

Earlier in this paper, it was proven that a convex hull algorithm could be a sorting algorithm as well. Does this relationship hold in the opposite direction also? Can a sorting algorithm find a convex hull? The answer, as shown by Clarkson and Shor in 1989 ([4] pg 471), is a most definite yes. They introduced the first Quick-Hull algorithm, named as such for similarities with Hoare’s Quick-Sort algorithm.

Quick-Hull starts from a rather basic assumption: Given \( L \) and \( R \), two extreme points of a set, the third point \( H \) that forms the triangle of largest area is part of the convex hull of the set. This assumption seems counter-intuitive at first but as one thinks about it the validity becomes apparent. The area of a triangle is defined to be one half the base times height. We know that any possible triangle

![Fig. 8](image)

\( h_3 \) is the largest height, meaning \( P_3 \) is the point that forms the triangle of largest volume. As you can easily see from the picture, \( P_3 \) is a part of the convex hull of the set. Since \( P_3 \) will always be an extreme point, it will always be part of the convex hull of the set.
formed will be an acute triangle since two of the three points are on the extreme of the set. This means the base will be constant for all the different triangles, and the deciding factor of the area will be the perpendicular from the line RL to the point H. The larger the area the farther away it is, and therefore the farthest away point must be a part of the convex hull. See Fig. 8 for the proof of this. Since the points R, L and H form a triangle, we know that all points contained within the triangle cannot be on the convex hull since they are already contained within a hull, and can be ignored. However, the points not contained in triangle R H L can be divided into two sets, S₁ and S₂ where S₁ is the set of points outside L H and S₂ is the set of points outside H R. (see fig 9 below for a pictorial representation)

Fig. 9

Triangle L R H (Blue Lines) has the greatest area; therefore H is a part of the convex hull. Ignore the points contained within the triangle and recursively call Quick-hull on the left and right set of points not contained within L R H, called S₁ and S₂ respectively. This process will return the convex hull of the entire set (Red Line).

By running the Quick-Hull algorithm recursively on S₁ and S₂, we will eventually come up with the convex hull of the entire set. This leads us to a more formal definition of the algorithm, below.
Quickhull( PointSet S, Point L, Point R )

1. if ( S = {L, R} ) // if the only points in the set are L and R
2. return {L, R}
3. else
4. H = farthest( S, L, R )
5. S1 = points of S on or left of LH
6. S2 = points of S on or right of HR
7. return ( Quickhull(S1, L, H) ∪ Quickhull( S2, H, R ) )

where farthest( PointSet S, Point L, Point R) is a function that determines which point yields the triangle of largest area.

Given the above Quick-Hull algorithm, a simple parent routine that begins the Quick-Hull process with the smallest X-Coordinate point for L and sets R equal to an imaginary point at the same X-Coordinate but with the Y-Coordinate lowered by ε. This imaginary point is removed afterwards, making ε equal to zero.

Lines 4, 5 and 6 (above) run in an average O(n) time. However, in the worst case line 4 alone will take O(n) time and the overall algorithm will run slower. If each of the subsets S1 and S2 have a cardinality of at most a constant fraction of S, the algorithm will run in O(n lg n) [1]. Unfortunately, this is just the average case. The algorithm can have the same pitfalls as quick-sort, running the worst case time to O(n^2).

N-Space Convex Hulls

So far, most of the algorithms we have spoken of have worked well in two dimensions, but have had serious shortcomings when trying to apply the same process to
higher dimensional spaces. There are some algorithms that are effective in N-space, but to understand them we need some new vocabulary. We represent an N-space convex hull as a set of facets and lists giving the neighbors of each facet. The boundaries of a facet are called ridges. In 3 dimensions, facets are triangles and ridges are the edges of the triangles.

One of the algorithms we have already seen can be modified to work in N-space, but not very easily. If we pick any facet formed by the set, and look at all the vertices above it, we come up with many more possible facets. The same holds true for all the points below the facet. Using a Quick-Hull-esque algorithm, we can divide the set of points into an upper and lower, find the convex half-hulls, and combine to find the total convex hull. There is a proof of this in [4], but is outside the scope of this paper.

Neither Graham’s Scan nor the Jarvis March can be easily transferred to N-space as they are, but the Jarvis March can be seen as a specific case of a more general algorithm. If you recall, the Jarvis March mimicked the wrapping of a gift. This obviously works in real life, as evidence by the gifts you have received throughout your lifetime. Has there ever been one with a concave wrapper (unless it was specifically taped to be such)? The natural shape of a wrapped gift is the convex hull of that object.

Fig. 10

The figure to the left is a 3 Dimensional convex hull. A, B, C, D and P are points in the convex hull, while DP, PC, DC, PA, PB, AB, DA, and CB are ridges of the convex hull. Triangles DPC, DPA, ABP, APB and the rectangle ABCD are considered the facets of the convex hull.
Start by finding a facet F in the convex hull of the object. In three dimensions F is usually obtained by finding three points in one dimension’s extreme, i.e. the three points with lowest Y coordinate. In higher dimensions, it becomes much more complex to find, involving hyperplanes and several iterations of vector normalization, taking $O(n^d)$ where $d$ is the number of dimensions. (We will deal with a three dimensional example since three dimensions are much easier to visualize and understand.) Then, imagine the plane containing this facet. If F is composed of the three points, $f_1$, $f_2$ and $f_3$, it exists in plane $f_1f_2f_3$. If you were to find all the planes that can be formed by $f_1$, $f_2$ and any other point in the set, you would have a series of so-called half planes. This particular half-plane set is called $HP(F, f_1)$. $HP(F, f_1)$ would resemble a book with pages sticking into the air, where the cover of the book is the plane containing F and each page represents the half-plane containing a facet formed by $f_1$, $f_2$ and another point in S. Of course this would only be in one dimension of the possible three. The other half-plane sets would be $HP(F, f_x)$ where $x$ is the subscript of the point not coplanar with the half-planes.

For each of the three half-plane sets, find the half-plane with the greatest angle less than $\pi$ (180 degrees) between their half plane and the original plane. This is analogous to picking the last page sticking up in the book. You know that the hull formed by the front
cover and last page will bound all other pages in the book. Extend this logic and it follows that the facets contained within these planes (pages) must be part of the convex hull of the set. If a point E existed outside that bound, it would be possible to create a hull using E that would contain F. F would not be in the convex hull if that were the case, and therefore would not have been chosen to start the process. Using this simple contradiction proof we can intuitively see that the logic behind the gift wrapping algorithm is correct.

Pseudocode:

1. create a queue Q of facets and a file T of subfacets. (subfacets are facets sharing an edge with the main facet)
2. F ← an initial convex hull facet
3. Ť ← all subfacets of F
4. Q.enqueue F
5. while( Q != ∅ )
6. F ← Q.dequeue
7. T ← all subfacets of F
8. for each edge e existing in both T and Ť
9. F’ = facet sharing e with F
10. insert into Ť all subfacets of F’ not already there, remove those that are already there
11. Q.enqueue F’
12. output F

The first line is simply a “set up” line allocating space for the needed structures and data types. It runs in O(1). Line 2 finds a convex hull facet to start the wrapping off
with. As previously discussed, this step runs in $O(n^*d^2)$ where $d$ is the number of dimensions. Step three generates the subfacets of a facet. Since there are a constant number of points, determining the subfacets can be done straightforwardly in $O(d)$. Line 4 is a simple enqueue to an empty queue, done in $O(1)$. Line 8 is a search of $\mathcal{S}$, taking $O(d \lg M)$ where $M$ is the size of $\mathcal{S}$. Introduce some notation and the overall time complexity of the algorithm emerges. Allow $\varphi$ to be the total number of facets, and $\varphi'$ to be the total number of subfacets. Steps 6, 11 and 12 can be combined (enqueues and dequeues, plus the output) to be $O(d) \ast \varphi$, since it will execute $\varphi$ times. Step 7 finds the subfacets of a facet is $O(d)$ time (similar to line 3) but will run $\varphi'$ times. The test and update in lines 8 and 10 respectively run in $O(d \lg \varphi')$ for all $\varphi'$, so the overall complexity of those two steps is $O(d \lg \varphi') \ast \varphi'$. Step 9, the actual gift wrapping, takes $O(d^3) + O(nd)$. We can combine all these different analysis’s to conclude that the gift wrapping algorithm has the recurrence $T(d,n) = O(n^*\varphi) + O(\varphi' \lg \varphi')$. $T(d,n)$ was proven to be $O(n^{(d^2)})$ by Grünbaum in 1967 [1].

The $O(n^{(d^2)})$ running time makes the gift wrapping algorithm one of the faster n-space algorithms. There are a few others of note, but the complexity of dealing with n-space is such that they will not be dealt with as completely as the other algorithms seen so far.

The package wrapping algorithm stood as the only efficient n-space algorithm for quite some time. Then, in 1981 [2], Kallay thought of a new idea. Rather than determine the hull by looking at the whole set at once, he decided to look at a subset of the points and then increase the subset until it was the entire set. His algorithm starts by finding a convex hull from a random point (in two dimensions this would be analogous to finding
the triangle formed by the random point and any two other points in the same quadrant). Then, the algorithm considers each point $p$ in the set. If $p$ is external to the current hull, it constructs the cone formed by the addition of $p$, and removes all points that fall within the shadow of $p$ from the current convex hull, while adding $p$. An example of this is shown in figure 13.

Another algorithm for determining convex hulls is the merge algorithm. This one works in all dimensions up to 3. Start by sorting the set of points by $x$-coordinate, and in a tie by $y$, and in a tie by $z$. Split the list in half, and you get two disjoint sets. Find the convex hull of each set and you have two non-intersecting hulls. Merge them together and get one convex hull for all points. The algorithm is quite similar to merge sort, and works in the same running time of $O(n \lg n)$ [1]. An example of the merge step is shown in figure 14.

**Approximation Algorithms**

If you have ever played a 3D video game and seen some odd physics at work, it was most likely the result of the program’s convex hull approximation algorithm. Characters half in walls and not dead, objects passing through each other, and little collisions that should happen and don’t are all the result of convex hull approximations. These applications generally create the hulls on the fly, and therefore need them to be
created in as little time as possible with as few resources as possible. Rather than make a complete convex hull, sometimes an approximation of the hull is sufficient.

One very fast approximation for two dimensional convex hulls works as follows. Take the point set S, and find the points with the lowest and highest x coordinate. Divide the area between the two points into k subdivisions of length $\frac{1}{k}$. Scan through all the points and find the maximum and minimum y valued point in each subdivision. Those points become the convex hull approximation.

Examining the steps in the algorithm, it is easy to see why it is so fast. With the exception of the scanning of the entire point set, all operations are $O(1)$. Performing $O(n)$ constant operations leaves us with an $O(n)$ running time. This is much faster than any of the exact convex hull algorithms, but in exchange for this speed increase we have some error associated with the hull. It is possible for a point that is on the actual convex hull to not be in the approximate hull, and vice versa. However, when using the algorithm above, any point not contained in the convex hull is at most a distance of $1/k$ from the hull [5]. This is due to the max/min properties within the k subdivisions of the points used to approximate the hull. So when your character is half in the wall and half out, you will now know that the approximation uses a really small k.

**Conclusion**

As was mentioned earlier, there are a lot of algorithms for finding the convex hull. Those discussed in this paper are some of the more groundbreaking ones at their time, and they trace the development of algorithms from the slow and dirty brute force to the faster and more elegant algorithms used today. There will probably be no further development of convex hull algorithms, due to the fact that optimal algorithms already
exist for both n-space and 2D/3D point sets. Use of these algorithms will continue on indefinitely though. The convex hull is one of the most basic structures of computational geometry, used as a stepping stone to solving much harder and more complex problems.
Works Cited:


