Implementing and Visualizing Algorithms for Computing Convex Hulls in the Plane

URS Project Report

M. Tabacman

Advisor: Dr. R. Janardan

1 Introduction

The Convex Hull of a set of input points in two dimensions is the smallest convex polygon that contains the entire point set [1][1]. Imagine the points of a set are nails on a board. If we were to stretch a rubber band around all of them and let go, the shape assumed by the band is the boundary of the convex hull.

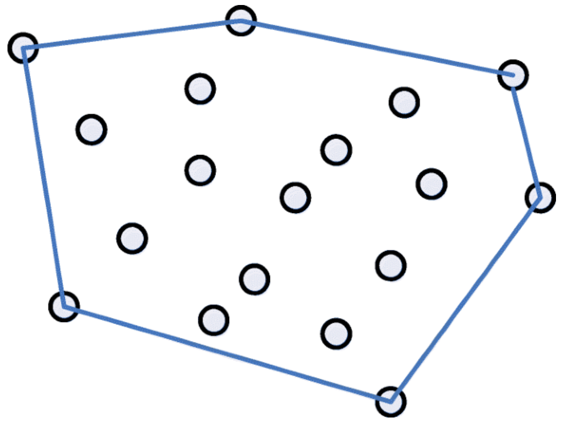


Figure 1: The lines represent the boundary of the convex hull. The points that connect these lines are the in the convex hull.

[1]A convex polygon is one where each internal angle is less than or equal to 180 degrees. For the purposes of defining the convex hull, we require that each internal angle be less than 180 degrees. In particular, this means that there cannot be three or more input points that lie on a bounding edge of the hull.

The convex hull is foundational to the area of computational geometry as it is a key component of several other problems, such as Voronoi diagrams, shape modeling and simplification, path planning, accessibility maps, and visual pattern matching.
Given its importance, a number of computer algorithms have been developed over the years to compute the convex hull. These algorithms employ different design paradigms and have different performance characteristics. This diversity of implementation makes convex hulls ripe for use as a pedagogical tool to teach students about different algorithm design paradigms. Therefore the goal and work product of this project was to implement and visualize several such algorithms.

2 General Implementation Details

In this section, we describe various aspects of the implementation, including choice of programming language, user interface, data sets, geometric primitives, and representation details.

2.1 Language

The project is written in Java, specifically, it was tested with Java 11. Newer versions like Java 16 should still work, given no dependencies are missing.

2.2 Geometric Primitives

Our algorithms make use of several basic operations, as listed below.

- Turn direction: A function that takes three points a, b, c in the plane and determines if the path determined by $a \to b \to c$ makes a left or a right turn (or neither) at the middle point b.
- Sideness test: A function that takes a line and a point and determines if the point is above, below, or on the line.
- Maximum orthogonal distance: given a line and an array of points, returns the point that is the furthest from the line (i.e., has the largest perpendicular distance to the line).

It turns out that all of these can be expressed in terms of the cross product of two vectors.

- Turn direction: To compute this, we can create two vectors from a to b and from b to c. Then, we take the cross product of these. The result is a real number. If this number is positive, then there is a right turn; if this number is negative then there is a left turn; if this number is 0 then the three points are collinear.
- Sideness test: This is pretty similar to the turn direction test. We create two vectors: One is described by the line, and the other goes from the end of this line to the point. We compute the cross product of these two vectors. If the result is positive, then the point is above the line; if it's negative the point is below the line; if it's 0 the point is collinear with the line.
- Maximum orthogonal distance: First we make the vector described by the line. Next, for every input point, we create a vector from the start of the line to that point. We compute the absolute value of the cross product. Once we've done this with every point we can see which point produced the cross product with the largest magnitude. This is the point that is furthest from the line.

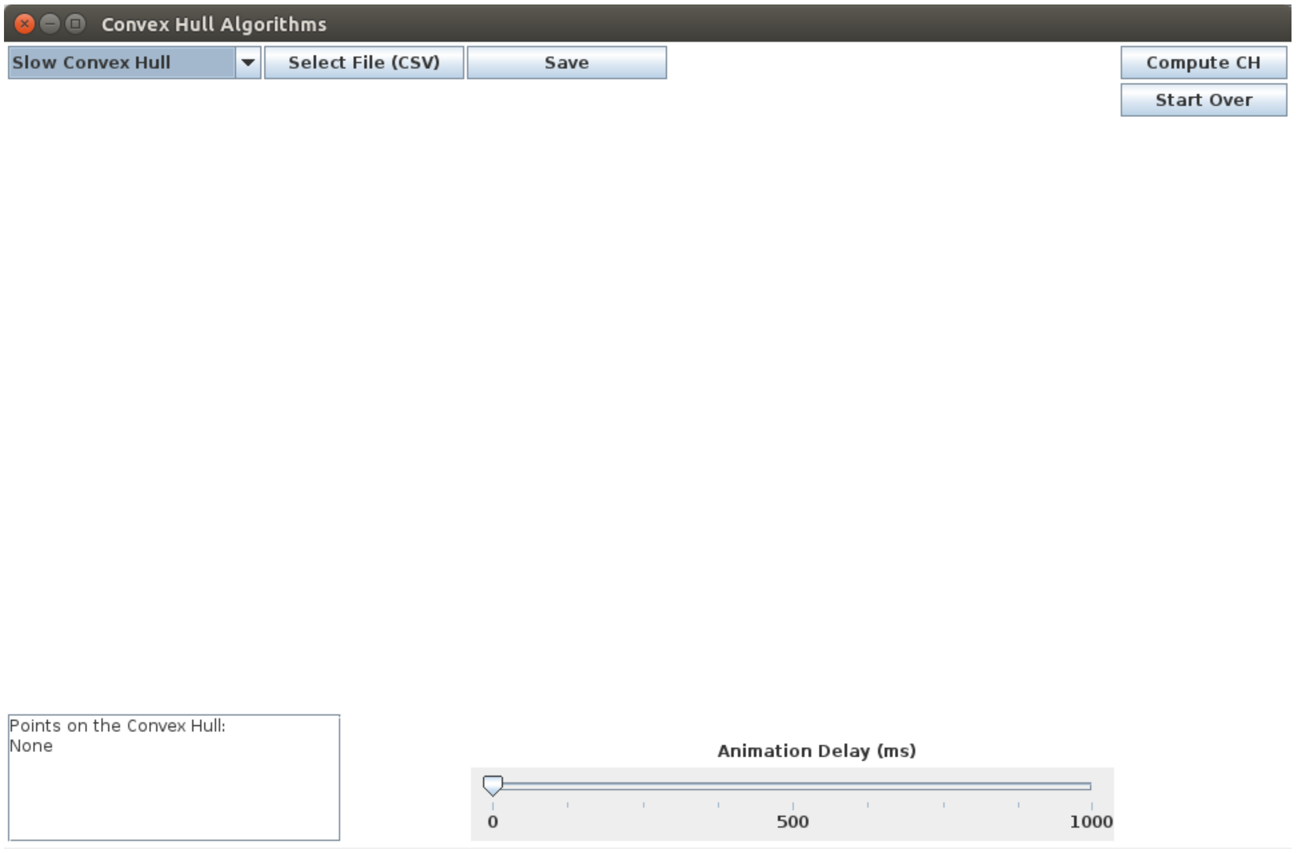
Therefore we need an efficient way to compute the cross product. While this can be done using a well-known trigonometric formula, this can be slow and lead to numerical issues. Out goal is to use simple operations, specifically addition and multiplication, to compute this in an efficient and stable way. Since we are limited to two dimensions for this project, computing the cross product is relatively simple. Given two vectors u and v each with x components $u.i$ and $v.i$ and y components $u.j$ and $v.j$,

$$u \times v = u.i * v.j - u.j * v.i$$

(Continued on next page)

2.3 User Interface

The UI contains several buttons with different utilities, such as saving a set of points, inputting a set of points, restarting the canvas, and computing the convex hull. On the bottom left of the window, there is a text area that shows what points are on the convex hull, along with their ’real’ values (the values that the computer uses to compute the convex hull). This text area also shows exceptions generated by the program.

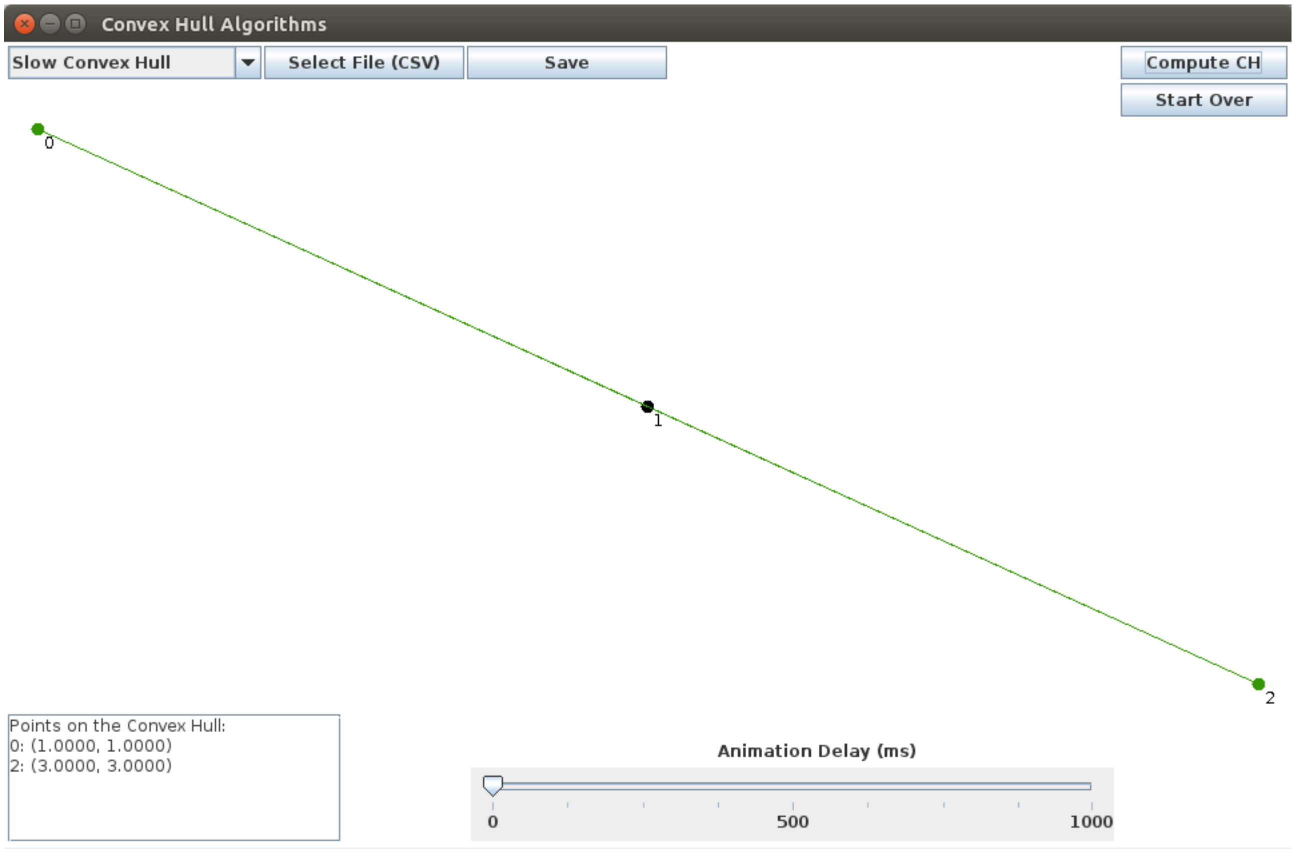


Points on the Convex Hull:
None

Figure 2: The UI described above

2.4 Input and Output

The files used as input and output are .csv files. The data is structured as a two columns of comma-separated values. For each pair of values, a new point is created using the first value as the point's *x* coordinate and the second as its *y* coordinate. By far the easiest way to create one of these files is by inputting points with the mouse and clicking the save button, but the user is free to write their own files (particularly if they desire to have collinear points as those are hard to manually input). Note that once a file is selected, the user must press "Compute CH" for the points to appear.



```
Points on the Convex Hull:
0: (1.0000, 1.0000)
2: (3.0000, 3.0000)
```

Figure 3: The .csv file with the contents
1,1
2,2
3,3
produces this output.

2.5 Scaling

At the start of this project, there was a fixed scaling factor. When input files were introduced, this became a problem, as any and all input points had to have x and y coordinates between 0 and 1 inclusive, otherwise they would not be drawn on the screen. Thus, we designed a system wherein each time a file was loaded, or each time a point was placed outside the existing plane, the program recalculated this scaling factor. Later on when window resizing was added, the program made sure to recalculate the scaling factor each time this happened as well. The result is that all points are visible at all times.

2.6 Internal Representation

There are three main lists that contain point information.

1. *pntarrReal*: contains the 'real' values of the points. This is the list that the program uses to do all computations.

2. *pntarrScale*: contains the scaled values of the points. This is the list that the program uses to paint points onto the screen and to construct lines used in animations and such. pntarrScale and pntarrReal are synchronized in the sense that the ith value of pntarrScale and pntarrReal represent the same point in different scales.

3. *outarr*: contains the indexes of the points in pntarrReal or pntarrScale. Generally, it only contains info on the points that form part of the convex hull.

2.7 Animation

During the execution of an algorithm, a point is generally in one of three states. "Processing", meaning that it is part of the current computation; "accepted" meaning that it has been found to be part of the (or a) convex hull; and "rejected" meaning that it has been found to be not part of the hull. For ease of visualizing, these points are colored blue, green and red, respectively. This is somewhat modified for recursive algorithms, as some of them use green to signify that a specific call has returned successfully.

3 Algorithms

What follows is a description of the algorithms that we implemented, along with a discussion of our specific implementations and their animation. Throughout, n denotes the number of points for which the convex hull is sought.

3.1 Slow Convex Hull ([1])

This algorithm works by generating a directed edge between every pair of points and checking for each such edge whether all remaining points are consistently on one side of the edge (say the right side). If so, the edge is a valid edge of the convex hull; otherwise, it is not. Since the algorithm checks every pair of points and for each pair checks all other points, its run time is $O(n^3)$.

3.1.1 Implementation Details

Our implementation is somewhat naive (mirroring the common description of the algorithm) in the sense that if the algorithm finds that one point is on the wrong side, it does not stop the computation. It also discriminates in the sense that points have to be on a specific side of the line for the line to be valid. To work around collinearity issues (an issue we had with all of the implemented algorithms) a helper function was designed, called *allButEnds* that takes an array of collinear points and returns all points except the end points. This is then used to remove any inappropriate points from the finalized convex hull.

3.1.2 Animation

The animation consists of a directed line between two points and a grey ray that shows an extension of that same line. Points are then individually sorted into either green or red depending on what side of the line they are on. If all points end up green, then the line highlights green and is on the convex hull.

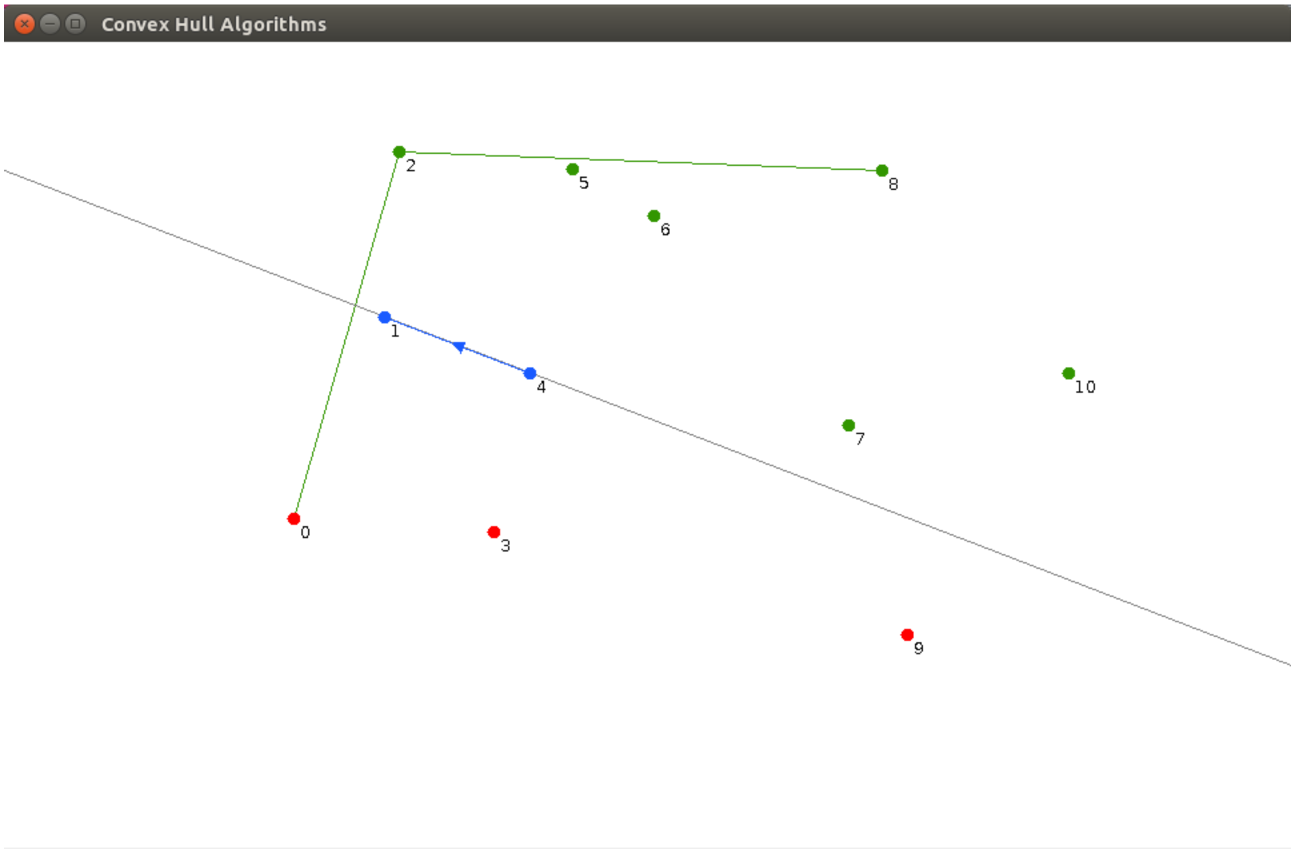


Figure 4: Slow Convex Hull processing the line 4 $\rightarrow$ 1, which ultimately is not on the convex hull.

3.2 Graham's Algorithm ([1][2])

This algorithm works by computing the convex hull as the union of an upper hull and a lower hull. (The upper hull runs from the leftmost to the rightmost point and all other points lie below it. The lower hull is symmetric.) To compute the upper hull, the algorithm considers the points in increasing x-order. It checks whether the current rightmost triple of points forms a so-called non-right turn. If it does, then it eliminates the middle point of the triple and repeats the turn test with the new rightmost triple. Otherwise, it advances the search to the next point in sorted order. When this phase terminates, the points that have survived define the boundary of the upper hull. The process is then repeated (for all points) in decreasing x-order to compute the lower hull. Its run time is $O(n \log n)$

3.2.1 Implementation Details

To avoid incorrectly adding collinear points to the output set, the turn checker checks that the angle is strictly greater than 180 degrees, otherwise some collinear points would be added. Note that collinear points should not be added as that would mean that the containing polygon has an internal angle that is 180 degrees, violating the requirement for the convex hull. At this point we also modified the implementation of the points to retain the non scaled value of the points and this helped a lot with rounding errors that arose when the scale of the points had to be changed.

3.2.2 Animation

The animation highlights the rightmost (leftmost when doing lower hull) three points to be processed in blue. It does so by drawing two lines and a curve representing the angle formed by those lines on the outside of the polygon. If this angle is greater than 180 degrees then the point is accepted and the whole highlighted part turns green. if it is less than or equal to 180 degrees, then the highlighted lines and points turn red, the middle point is removed and the next point in the sorted order is processed.

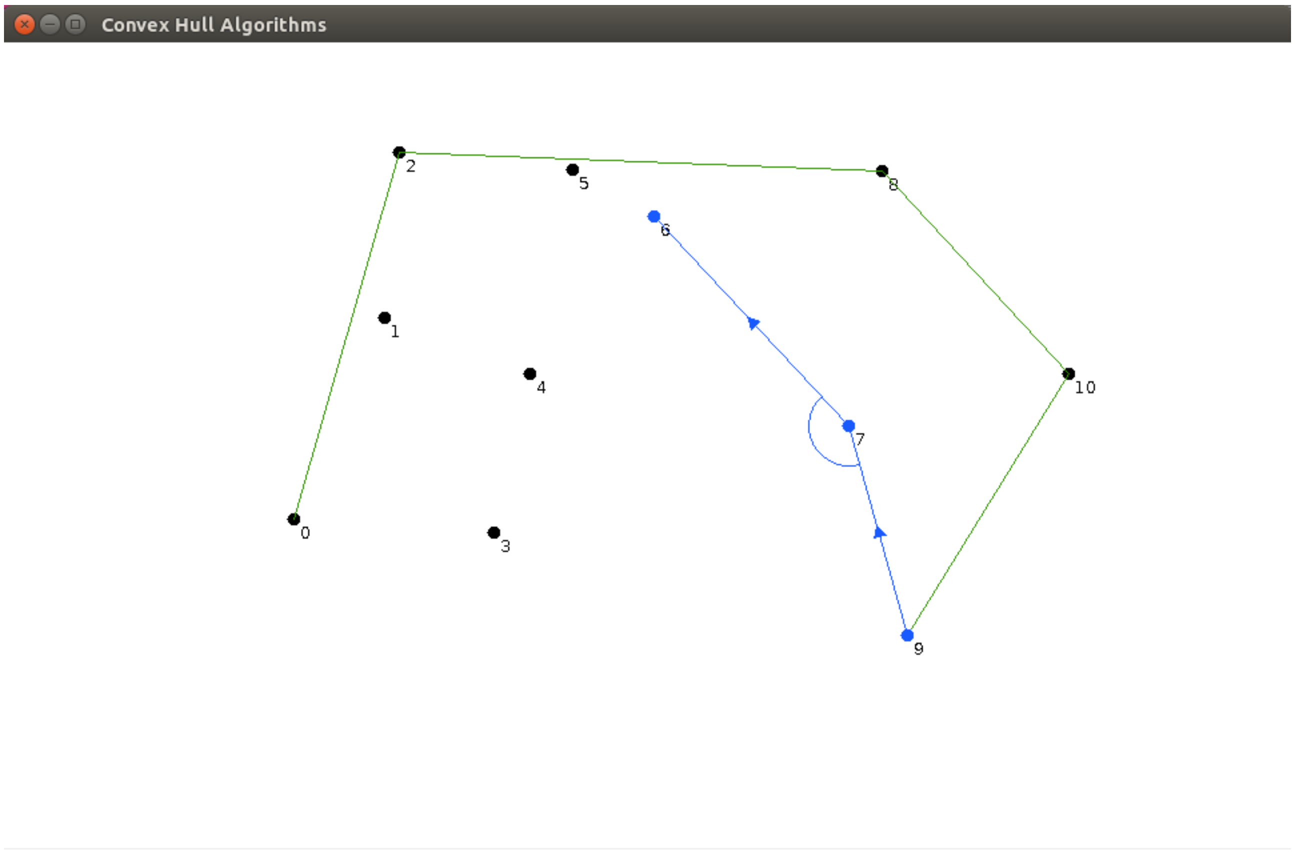


Figure 5: Graham's algorithm processing three points of the lower hull, after already having computed the top hull. Point 7 will be rejected because $9 \rightarrow 7 \rightarrow 6$ forms a left turn.

3.3 Quickhull ([3])

This algorithm employs a divide and conquer approach to compute the convex hull. The first step is to find the points with the minimum and maximum x coordinates. Using the line L formed by the two points, the set is divided into two subsets that are processed recursively, as follows: the point p that is above the line and at a maximum distance is determined. The points lying inside the triangle formed by the three points cannot be in the convex hull, therefore they can be ignored. The previous step is repeated for the lines defined by the two edges of the triangle that contain p. And so on until no points are left to process. A similar approach is employed for the points below the original line L. This computes the convex hull in $O(n \log n)$ time.

3.3.1 Implementation Details

The main function for Quickhull divides the input set in two, and calls the helper function *findHull* on both sets. The 'result', i.e. the set of points on the convex hull, is passed by reference to all functions, so findHull's return value is void. *findHull* finds the point furthest away from the line and recursively calls *findHull* until there are no more points to add.

3.3.2 Animation

A blue line divides the set of points in half. The remaining points are divided into two sets Set 1 and Set 2, depending on where they lie with respect to the blue line. The distance of every point to the line is highlighted in blue, then the furthest point is highlighted in green while the rest are red. A new blue line is created from the start point of the original line to this furthest point. This process is repeated until there are no more points to be added, at which point the line becomes green. This process is symmetrical on the right side and the lower half.

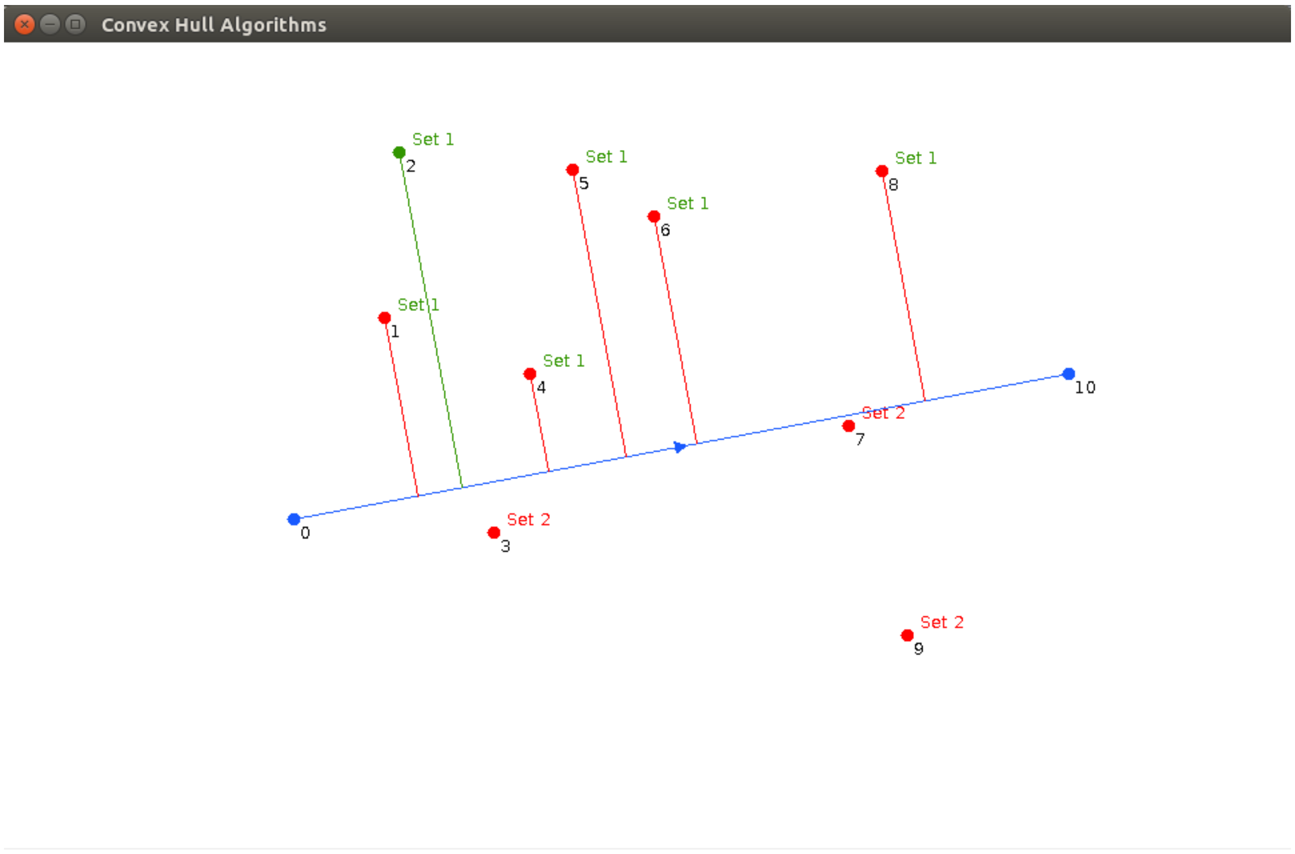


Figure 6: Quickhull deciding that Point 2 is the furthest from the line $0 \rightarrow 10$. We see also that points were divided into Set 1 and 2. Next, it will create the lines $0 \rightarrow 2$, and $2 \rightarrow 10$ and repeat the process.

3.4 Jarvis's March ([2][3])

Jarvis's March uses a technique called gift wrapping. Visually this is because the algorithm works like wrapping a piece of ribbon around a set of nails in a board. It starts by taking the leftmost point on the plane (which is definitely on the convex hull). Next, it finds the point with the smallest polar angle with respect to the first one. This is the next point on the convex hull. It continues the process until it reaches the initial point. This algorithm runs in $O(nh)$ time, where h is the number of points on the convex hull. Unlike the previous algorithm, its run time is sensitive to the size of the hull.

3.4.1 Implementation Details

This implementation does not use radial sort. (Indeed, such a sort would take $O(n \log n)$ time, which would be too expensive for small values of h, e.g., $h = O(1)$.) Instead it uses a linear search throughout the remaining points and stores the one with the smallest polar angle as its best candidate until the next smallest polar angle is found.

3.4.2 Animation

Two blue directed lines are formed between the leftmost point and the first two points in the input set. The one with the larger polar angle is marked red, and is replaced by another blue line. This process is repeated until we have checked every pair of points possible, at which point our standing candidate is the smallest polar angle. This line is highlighted green and the process begins again using the endpoint of this line as the next starting point. This is repeated until the convex hull is formed.

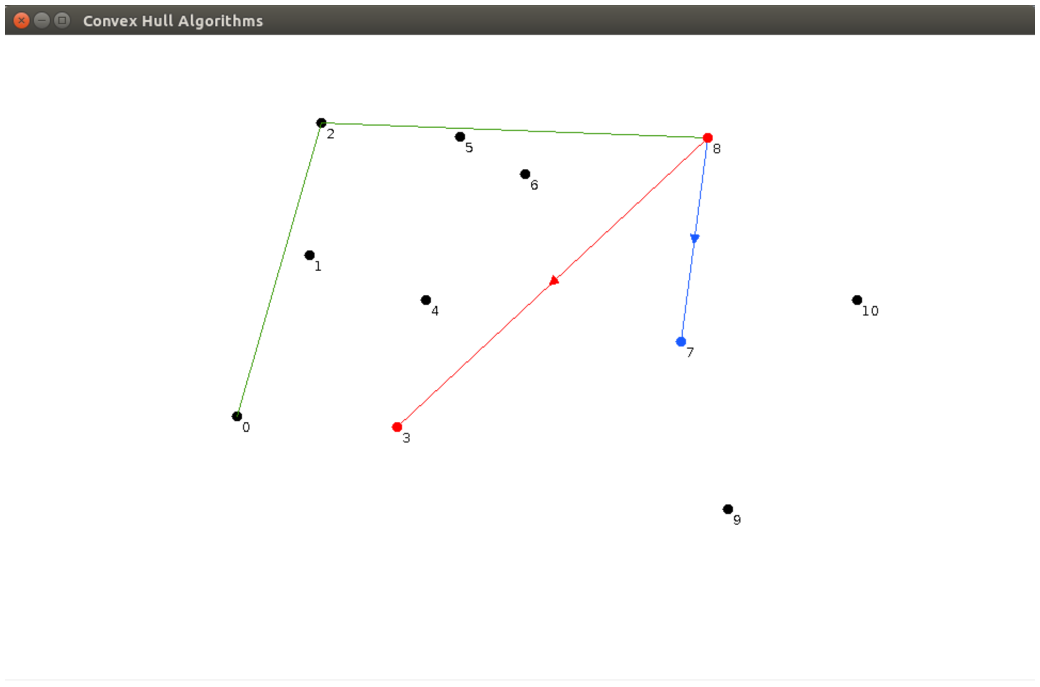


Figure 7: Jarvis's March deciding that line $8 \rightarrow 7$ has a smaller polar coordinate than line $8 \rightarrow 3$. Next it will check line $8 \rightarrow 7$ against (say) line $8 \rightarrow 9$.

3.5 Incremental Construction

This algorithm starts by computing the convex hull of the three leftmost points. Then it adds the remaining points by increasing x. It stores the points of the upper and lower hull separately in x-sorted arrays. Then, it computes the tangents from the newest point to the convex hull by scanning through the arrays and testing for tangency. This scan can either be linear or binary, leading to $O(n^2)$ or $O(n \log n)$ run times, respectively.

3.5.1 Implementation Details

The tangency scan is linear, therefore this implementation is $O(n^2)$ technically. The hardest part of this algorithm was the design of a way to systematically remove lines that no longer form part of the convex hull. In this case, because of the way we were adding lines to our internal line array, we were able to integrate this removal to the part of the algorithm where we remove points no longer on the convex hull.

3.5.2 Animation

The leftmost three points are highlighted green, as they themselves form a convex hull. Then a new point is highlighted in blue and two blue lines are drawn from that point to the constructed convex hull to its left. The point higher up on the convex hull is highlighted red and replaced by another line going from the processing point to a point on the convex hull. Process is repeated until all points on the convex hull are checked, and the remaining line is highlighted green. The process is repeated to find the upper tangent. This is repeated for all points in the set of points.

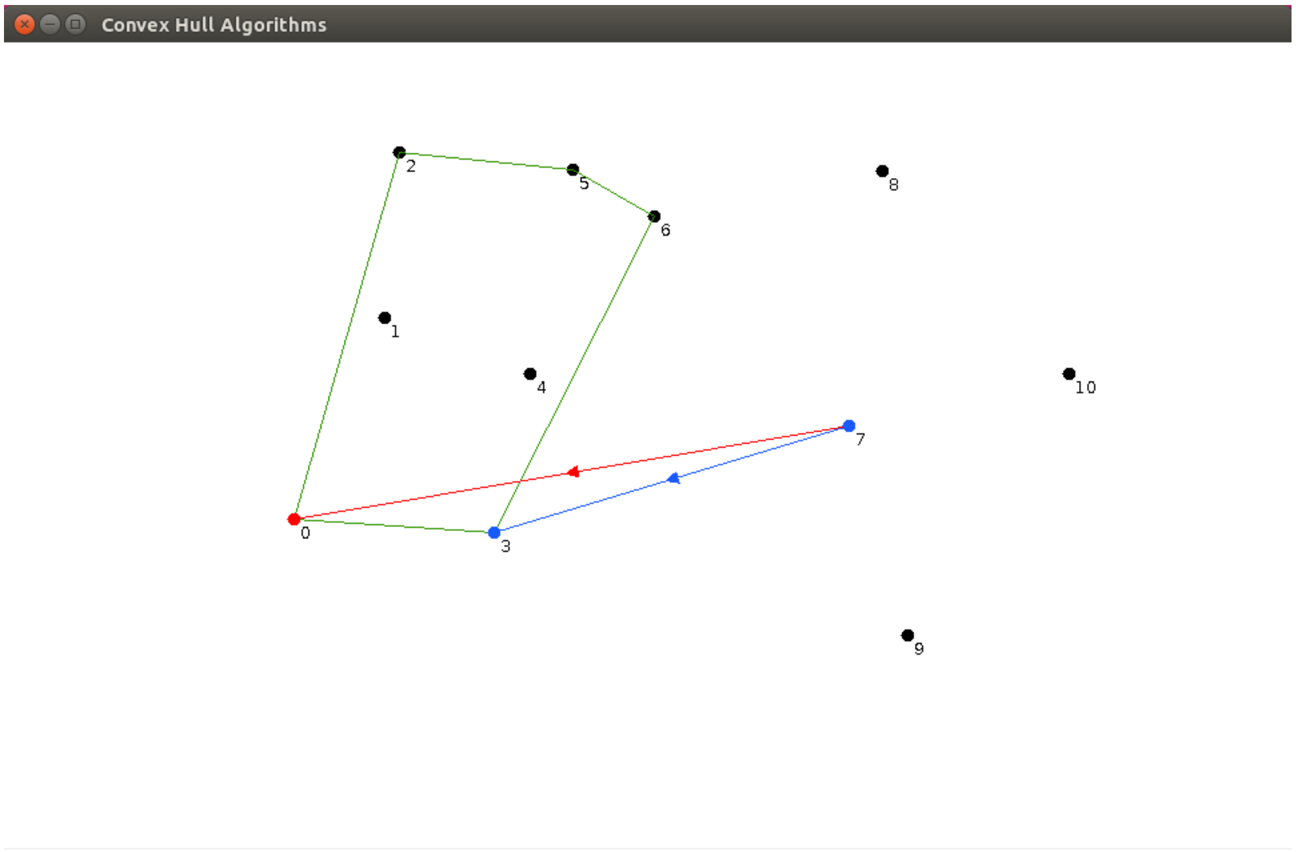


Figure 8: The incremental construction deciding that line 7 $\rightarrow$ 3 is a better candidate for lower tangent than 7 $\rightarrow$ 0. Next, the algorithm will decide this is the lower tangent and search for the upper tangent.

3.6 Divide and Conquer ([4])

The algorithm starts by dividing the set of n points into two roughly equal-size sets based on x coordinate. The 'left' set contains the first $\frac{n}{2}$ points, and the 'right' set contains the last $\frac{n}{2}$ points[2]. We then compute the convex hull of each set of points recursively. Once this is done, we need to find the upper and lower common tangents between these two convex polygons. We find the upper tangent by taking two vertices, one on the left convex hull and one on the right convex hull and connecting them. While the line intersects the left convex hull, we raise that endpoint of the line. Then, while the line intersects the right convex hull, we raise that endpoint of the line. We then check again whether the lines intersect either of the polygons and repeat as needed until tangency

[2]An issue arises when there is an odd number of points. Instead, the 'left' set gets 1 point less than the 'right' set

is achieved for both hulls. We repeat this for the lower common tangent. Then, points not on the convex hull are removed and the remaining points are combined into a single list. The base case for this algorithm is a set of 2 or 3 ponts, as they themselves form their own convex hull.

3.6.1 Implementation Details

As with the Incremental Construction, the hardest part of this algorithm was making sure all inappropriate lines were removed from the convex hull between merging steps. Unlike the Incremental Construction, the algorithm's structure did not lend itself to the systematic removal of lines. Instead, we decided it was best if it was done brute force; i.e. checking whether each line crossed the resulting convex hull. This doesn't affect the runtime unless the user chooses to animate the convex hull, as it is only done when the animation delay is greater than 0.

3.6.2 Animation

All points are highlighted blue. The set of blue points is divided in half by a grey line through the middle. The process is repeated with the points on the left until there are only two or three points left. These points are highlighted green and then we check the right side. This happens throughout the algorithm. Once we have two sets of points formed into two convex hulls (highlighted in green) the animation shows a similar tangent-finding process as the incremental construction, except this time either end may change depending on which convex hull it intersects.

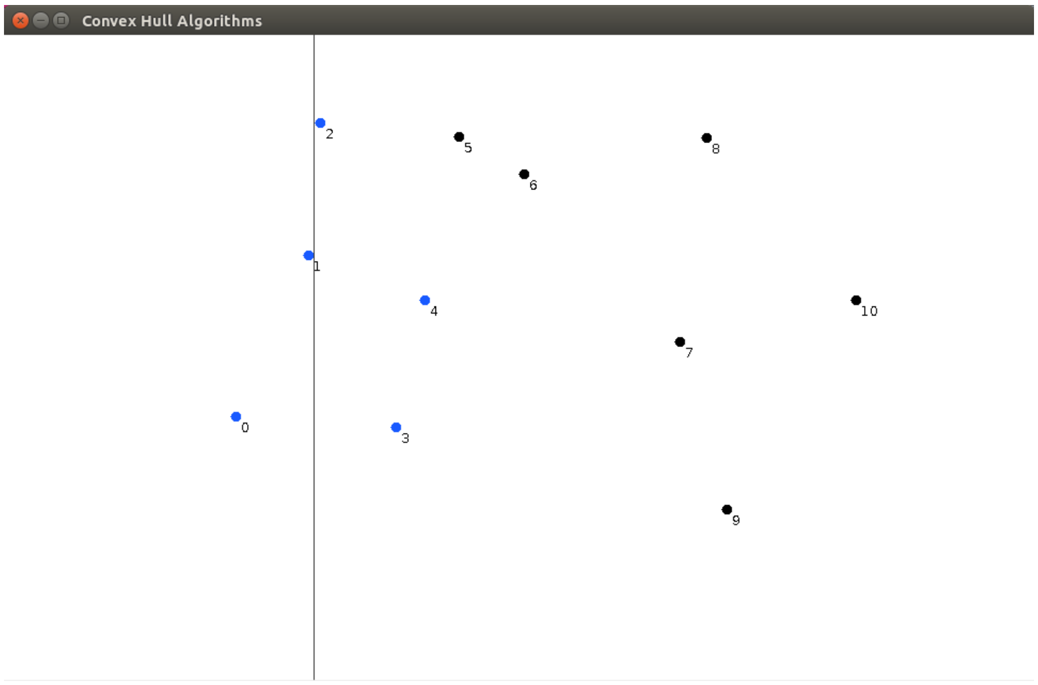


Figure 9: Divide and Conquer algorithm dividing the set of points 0, 1, 2, 3, 4 into two ’equally’ sized sets.

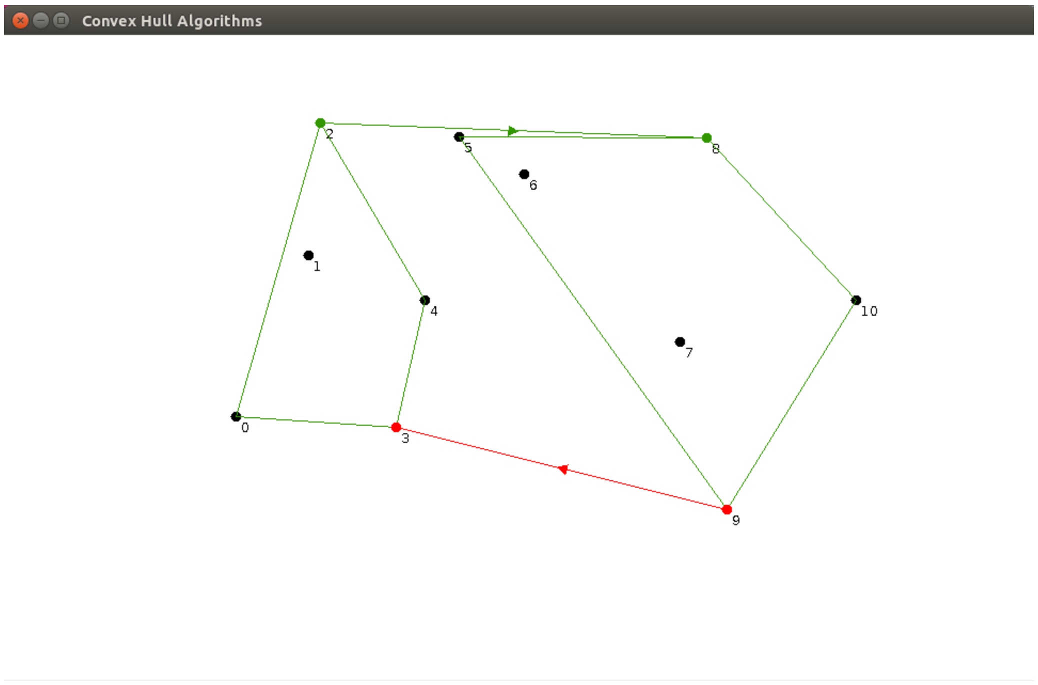


Figure 10: Divide and Conquer algorithm deciding that the line 9 $\rightarrow$ 3 cannot be the lower common tangent as (upon extension) it intersects the left convex hull.

4 Runtimes

Here is a table with real time values, measured in seconds. Each algorithm was fed a set of n random points 100 times, and then the average execution time was computed.

Table 1: Average time in seconds to compute the convex hull of a set of points of size n.

	10^2	10^3	10^4	10^5	10^6
Slow Convex Hull	0.021	11.144	**[p]** $1.6 \cdot 10^4$	**[p]** $1.6 \cdot 10^7$	**[p]** $1.6 \cdot 10^{10}$
Graham's Algorithm	0.001	0.002	0.011	0.087	1.214
Quickhull	0.001	0.002	0.006	0.044	1.006
Jarvis's March	0.002	0.017	1.288	**[p]** $7.66 \cdot 10^2$	**[p]** $7.66 \cdot 10^4$
Incremental Construction	0.002	0.008	0.1	6.83	**[p]** $3.2 \cdot 10^3$
Divide and Conquer	0.003	0.013	0.194	16.992	**[p]**$7.9 \cdot 10^4$

To calculate projected values, we take the average of the values resulting from taking each average time t and dividing it by $f(n)$ where f is defined to match the theoretical runtime of the algorithm and n is the size of the input set. Then, we take that value and multiply it by $f(m)$ where m is the input size runtime we want to project.

As an example, let's say we wanted to project the runtime of SlowHull at an input size of 10^4. We would first take the average of

$$\frac{0.021}{f(10^2)} = \frac{0.021}{(10^2)^3} = 2.1 \cdot 10^{-8}$$

and

$$\frac{11.144}{f(10^3)} = \frac{11.144}{(10^3)^3} = 1.11 \cdot 10^{-8}$$

which is

$$\frac{(2.1 + 1.11) \cdot 10^{-8}}{2} = \frac{3.21 \cdot 10^{-8}}{2} = 1.60 \cdot 10^{-8}$$

Now we take this value and multiply it by $f(10^4)$.

$$1.6 \cdot 10^{-8} \cdot (10^4)^3 = 1.6 \cdot 10^4$$

Table 2: This table details the calculated values used to project algorithm runtimes.

	Constant	Special Considerations
Slow Convex Hull	$1.6 \cdot 10^{-8}$	None
Jarvis's March	$7.66 \cdot 10^{-8}$	$f(n)$ approximated to n^2
Incremental Construction	$3.22 \cdot 10^{-9}$	First value ignored as it was too large
Divide and Conquer	$7.915 \cdot 10^{-8}$	None

These projections are approximations and are probably not very accurate, but actually running the algorithms and taking the real average would be too time consuming.

5 Work Product

The end product of this project was originally going to be a web-accessible tool to animate and visualize several convex hull algorithms. Through researching how this would be possible, most online resources recommend rewriting the entire program in JavaScript, or another browser friendly language. This would take serious amounts of time given we have no prior experience in such language. Applets and JApplets were also explored, but were ruled out because of deprecation. Other sources were taken into accounts, such as GWT and Java2Script but we have not been able to produce a web based result yet. That being said, the end result of this project is then a runnable jar file, which can be run by anyone with the Java JRE or JDK installed. This is pretty much widespread, especially among the target audience so it shouldn't be too much of a detriment. We also plan on releasing a git repository with all of the source code in it if the user desires to compile the project themselves. We have plans to try to rewrite the program in JavaScript as we are interested in learning the language and think this is a good project to work on. Until then the deliverable is perfectly capable of carrying out the task of illustrating each algorithm's design paradigm and how several different approaches can be used to solve the same problem with varying degrees of efficiency.

References

[1] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, second edition, 2000.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[3] Wikipedia contributors. Convex hull algorithms — Wikipedia, the free encyclopedia. `https://z.umn.edu/convex-hull-algorithms`, 2020. [Online; accessed 4-January-2021].

[4] Convex hull algorithms: Divide and conquer — algorithm tutor. `https://z.umn.edu/divide-and-conquer`, 2020. [Online; accessed 4-January-2021].