## CS478 - Computational Geometry

# Implementation of Two-dimensional (2D) Convex Hull Construction Algorithms and Comparing Their Performance

**Project Final Report**

**Mehmet Feyyaz Küçük - 22003550**
**Deniz Çelik - 22003271**

**Github**
https://github.com/mfkucuk/ConvexHullProject

**Instructor: Uğur Güdükbay**

**16.05.2024**

## 1. Introduction

In this report, different approaches to constructing convex hulls will be discussed. These methods are:

- Graham's Scan Algorithm,
- Jarvis's March Algorithm,
- Quick Hull Algorithm,
- and Merge Hull Algorithm.

These algorithms will be discussed in detail using block diagrams and pseudocodes as well as the data structures that will be used to implement them. Details of other implementation details will be given such as how to visualize the convex hull in the program. First we will talk about the definition of the convex hull, and then a summary of our research on the algorithms and data structures will be discussed. After that, other implementation details will be discussed. Finally, we will conclude with a summary of the report.

## 2. Convex Hull

Given a set of points S in $R^2$, the convex hull H(S) is the smallest convex polygon that envelopes all of the points of S. A convex polygon is a polygon in which any line segment connecting any two points of the polygon (both its interior and boundary), will fall within the polygon. More formally, a subset C of $R^2$ is convex if $(1 - \lambda)x + \lambda y \in C$ when $x \in C,\ y \in C$ and $0 < \lambda < 1$ [1].

When we say the convex hull H(S) is the smallest convex polygon, we mean:

- The convex polygon P which is the convex hull H(S) of set S, there are no other polygon P' where $P \supseteq P' \supseteq S$.
- The convex hull H(S) of set S is the convex polygon P with the smallest area.
- The convex hull H(S) of set S is the convex polygon P with the smallest perimeter.

### 2.1. Extreme Points

A point p of a convex set S is called an extreme point if for no two points a, b, $a,\ b \in S$, such that p lies on the open segment ab.

**Relation between extreme points and the convex hull**

The subset E of extreme points of S is the smallest subset of S such that H(E) = H(S). This property of extreme points will be the main idea behind Graham's Scan Algorithm where the extreme points will be the vertices of the convex hull.

**Problem Definitions**

In our application, we will be utilizing Gaussian Distribution and Uniform Distribution to generate a random set of points S of 2-dimensional points and construct the convex hull H(S) for S.

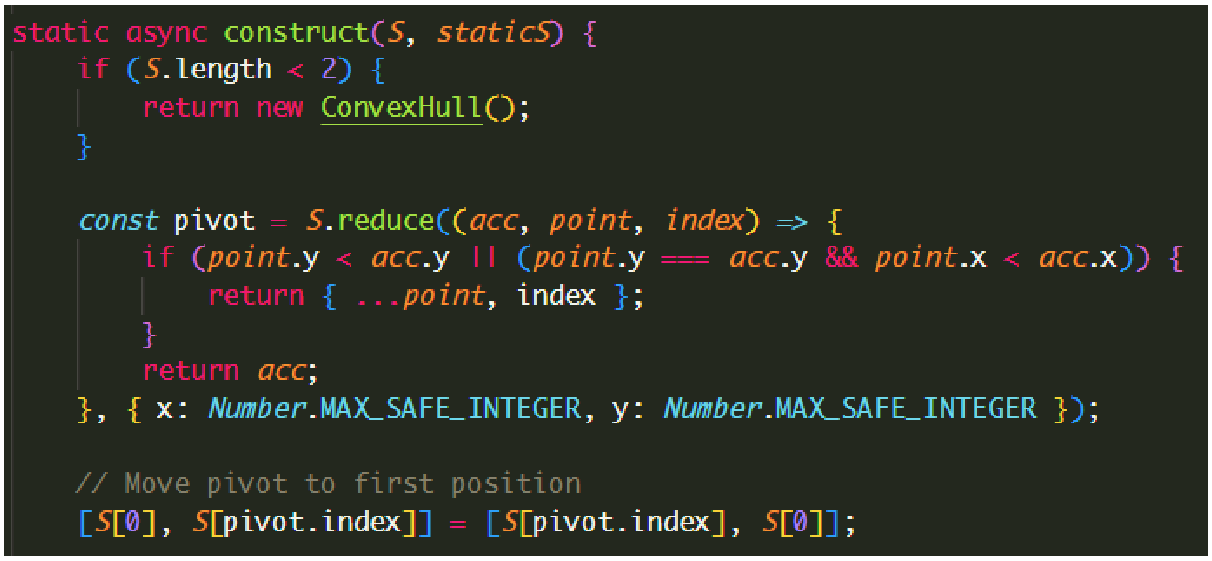
## 3. Background Research

In this section, we will talk about the algorithms for constructing convex hulls and data structures needed to implement them as well as different distributions we will be using for generating random points to feed as input.

### 3.1. Graham's Scan Algorithm

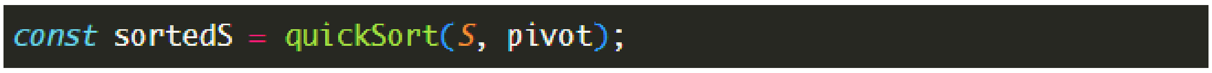
Graham's Scan is one of the methods for finding the convex hull of a finite set of points S. The algorithm has a time complexity of O(n logn) where n is the number of points in the set S. The complexity will be discussed in detail. The idea of Graham's Scan is to find E, the set of extreme points of S. Since H(S) = H(E), finding the extreme points will be sufficient to construct the convex hull. Incidentally, These extreme points will become the vertices of the convex hull, (i.e. we will connect the extreme points in a counter-clockwise fashion). The steps of the Graham's Scan Algorithm can be listed as follow:

- First we find the "pivot" point, the point with the lowest y-coordinate. If there are multiple points with the lowest x-coordinate (leftmost one), then we take the one with the lowest x-coordinate. We call this point P. This step can be done in O(n) time where n is the number of points in S.

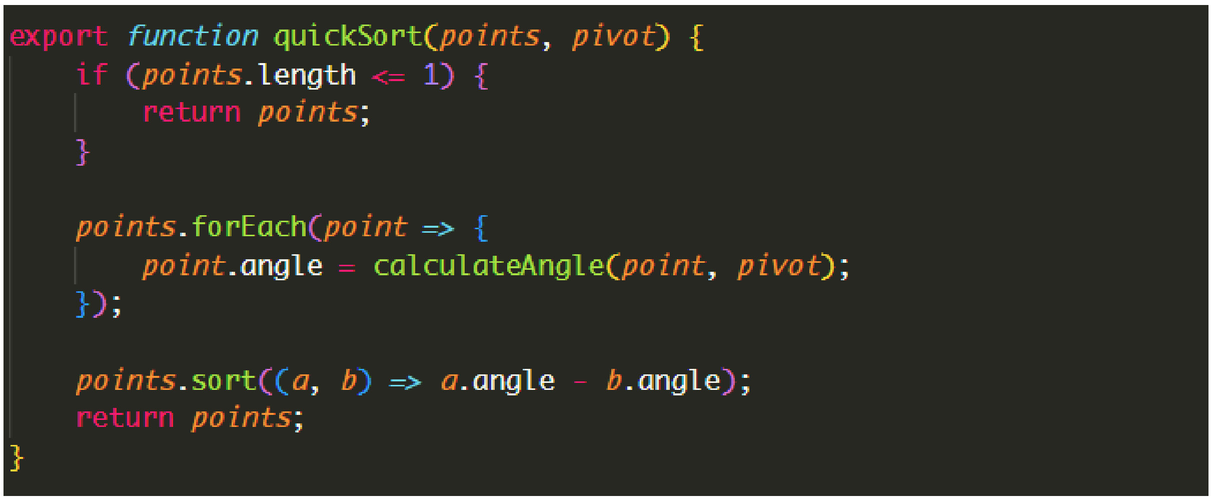


```
static async construct(S, staticS) {
    if (S.length < 2) {
        return new ConvexHull();
    }

    const pivot = S.reduce((acc, point, index) => {
        if (point.y < acc.y || (point.y === acc.y && point.x < acc.x)) {
            return { ...point, index };
        }
        return acc;
    }, { x: Number.MAX_SAFE_INTEGER, y: Number.MAX_SAFE_INTEGER });

    // Move pivot to first position
    [S[0], S[pivot.index]] = [S[pivot.index], S[0]];
```

- Then, the set of points must be sorted in an increasing order of the angle they and the point P make with the x-axis. This step can be done in O(n logn) time using a sorting algorithm like Merge Sort or Heapsort.

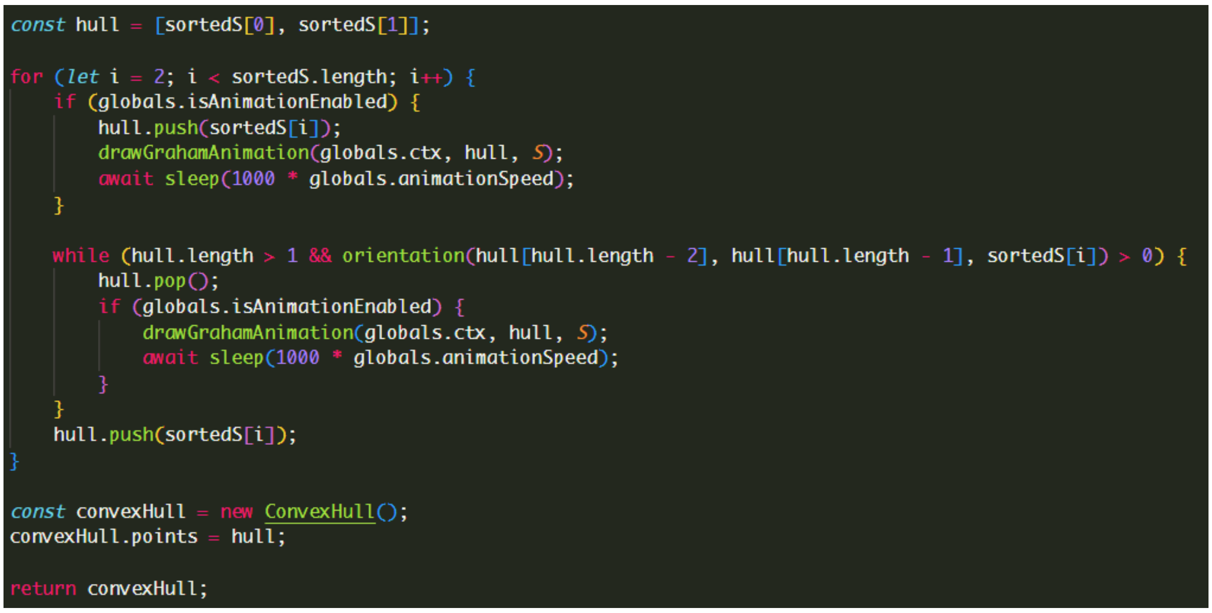


```
const sortedS = quickSort(S, pivot);
```



```
export function quickSort(points, pivot) {
    if (points.length <= 1) {
        return points;
    }

    points.forEach(point => {
        point.angle = calculateAngle(point, pivot);
    });

    points.sort((a, b) => a.angle - b.angle);
    return points;
}
```

- Next, for each point in the sorted set S, we check if traveling to this point from the two previous points makes a “left-turn” or a “right-turn”. If it makes a “right-turn”, the point preceding the current point is discarded and determined to be inside and not on the convex hull. Then the same check is done again until we get a “left-turn”. When we get a “left-turn”, we move onto the next point.



```
const hull = [sortedS[0], sortedS[1]];

for (let i = 2; i < sortedS.length; i++) {
    if (globals.isAnimationEnabled) {
        hull.push(sortedS[i]);
        drawGrahamAnimation(globals.ctx, hull, S);
        await sleep(1000 * globals.animationSpeed);
    }

    while (hull.length > 1 && orientation(hull[hull.length - 2], hull[hull.length - 1], sortedS[i]) > 0) {
        hull.pop();
        if (globals.isAnimationEnabled) {
            drawGrahamAnimation(globals.ctx, hull, S);
            await sleep(1000 * globals.animationSpeed);
        }
    }
    hull.push(sortedS[i]);
}

const convexHull = new ConvexHull();
convexHull.points = hull;

return convexHull;
```

**Note:** Here, the first and second point in the set is guaranteed to be in the hull, so we start from the third point. Also in all of our implementations you will see *globals.isAnimationEnabled* and sleeping the code for some time. These parts are related to the algorithm animation.

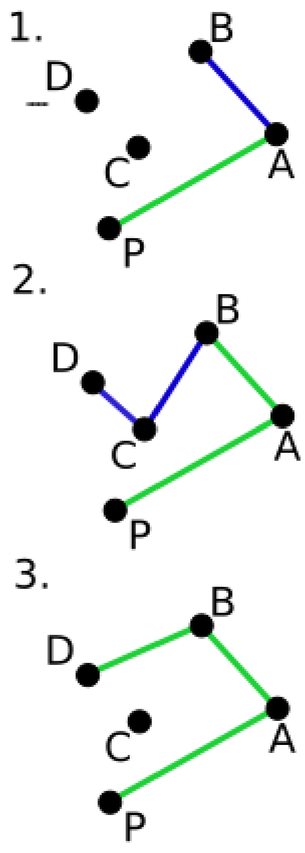


**Fig. 1.** Visual demonstration of step 3 of Graham's Scan

- The algorithm terminates when we reach the point we started with, in which case the convex hull will be successfully constructed.

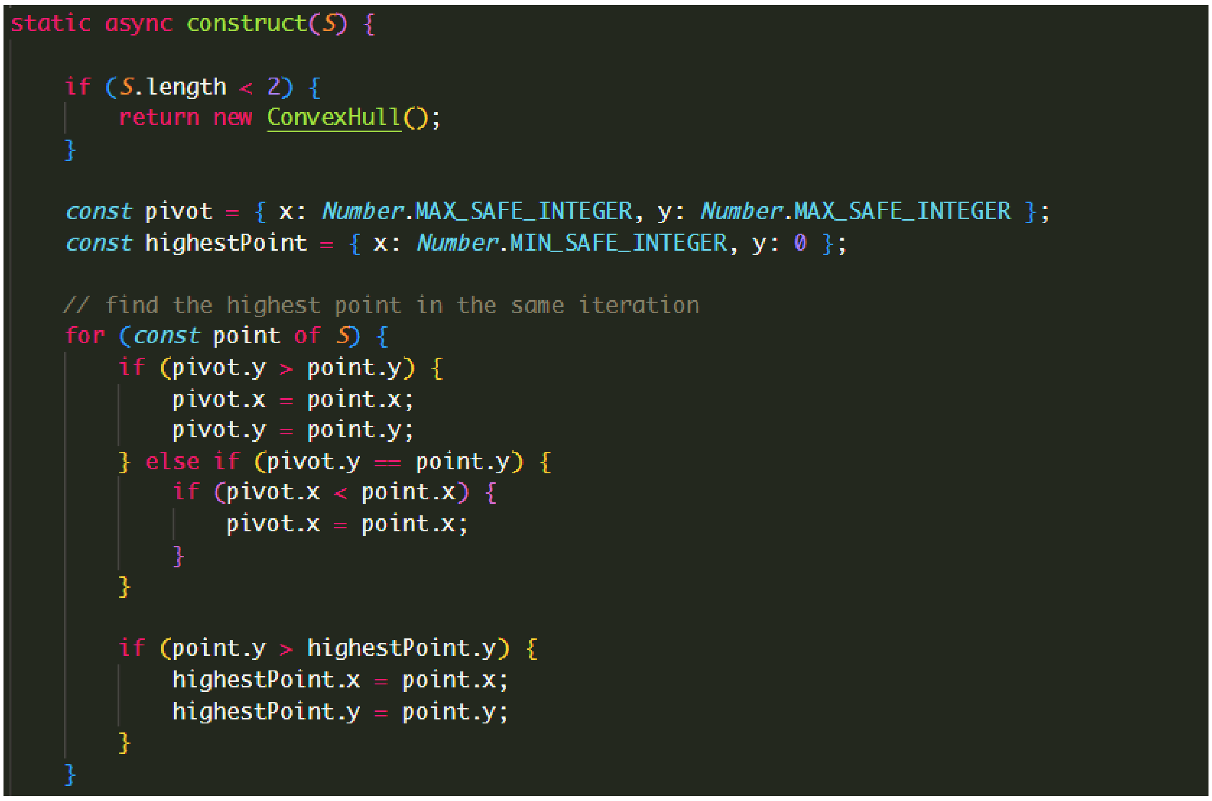
**Pseudocode**
Here is the pseudocode we designed by following the steps above:

```
procedure GrahamScan(S)
        let P0 be the leftmost lowest y-coordinate point in S // (Step 1)

        let sorted_S = Heapsort(S) // (Step 2) or MergeSort(S)  (w.r.t. P0)

        foreach P in S (Step 3)
                while P and two points preceding P in sorted_S make a right-turn
                remove the point before P from sorted_S
            endwhile
        endfor

        return sorted_S (Step 4)
end
```

### 3.2. Jarvis's March Algorithm

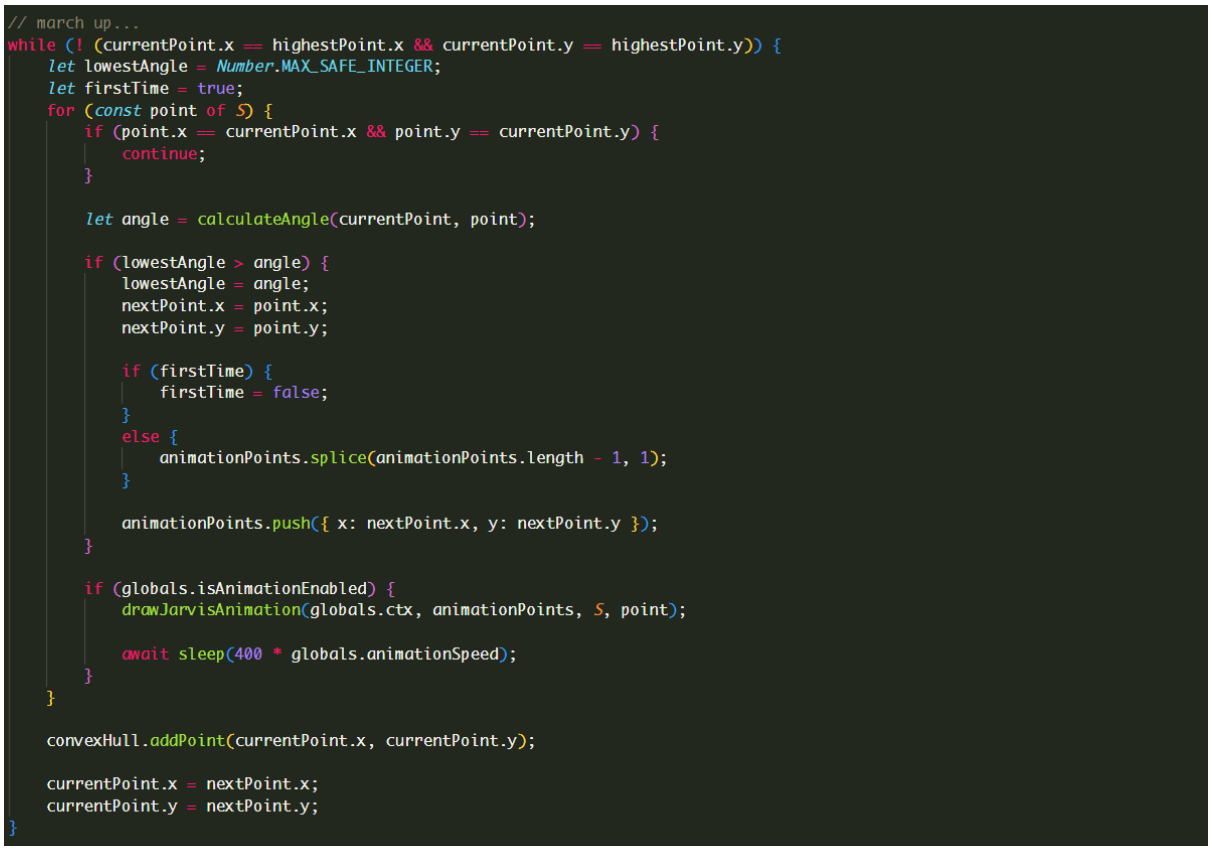
Jarvis's March algorithm operates on edges of the convex hull. While Graham's Scan was finding extreme points which would be the vertices of the convex hull, Jarvis's March finds the edges of the convex hull H(S). We can determine if for p and q where $p, q \in S$, the line segment pq is an edge of H(S) or not. If all the points in $S - \{p, q\}$ lie on the pq line segment or to one side of it, we can say that the line segment pq is an edge of the convex hull. Another clever trick we can use is if the pq line segment is an edge for the convex hull, then p and q must be endpoints for some other line segments that are also an edge for the convex hull. The steps of the Jarvis's March can be listed as follows:

- First we find the "pivot" point, the point with the lowest y-coordinate. If there are multiple points with the highest x-coordinate (rightmost one), then we take the one with the lowest x-coordinate. We call this point P. This step can be done in O(n) time where n is the number of points in S.



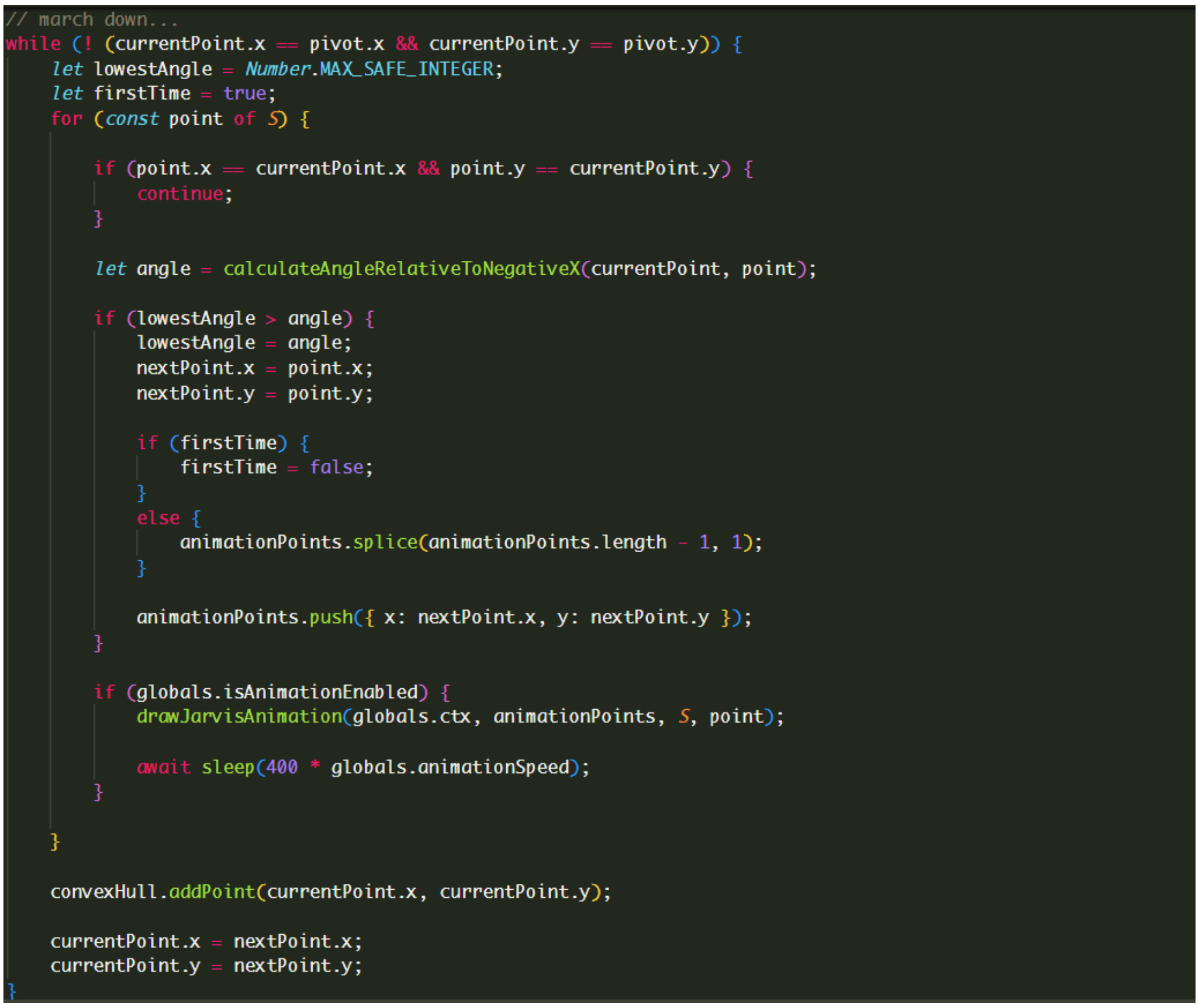
```
static async construct(S) {

    if (S.length < 2) {
        return new ConvexHull();
    }

    const pivot = { x: Number.MAX_SAFE_INTEGER, y: Number.MAX_SAFE_INTEGER };
    const highestPoint = { x: Number.MIN_SAFE_INTEGER, y: 0 };

    // find the highest point in the same iteration
    for (const point of S) {
        if (pivot.y > point.y) {
            pivot.x = point.x;
            pivot.y = point.y;
        } else if (pivot.y == point.y) {
            if (pivot.x < point.x) {
                pivot.x = point.x;
            }
        }

        if (point.y > highestPoint.y) {
            highestPoint.x = point.x;
            highestPoint.y = point.y;
        }
    }
```

- Starting from P, we go through the set S and find the point which makes the smallest polar angle with the current point with respect to the x axis. The line segment from the current point to that point will be an edge of the convex hull. Each successive point will be the endpoint of the edge we found in the last iteration. This step will continue until we reach the point with the highest y-coordinate.



```
// march up...
while (! (currentPoint.x == highestPoint.x && currentPoint.y == highestPoint.y)) {
    let lowestAngle = Number.MAX_SAFE_INTEGER;
    let firstTime = true;
    for (const point of S) {
        if (point.x == currentPoint.x && point.y == currentPoint.y) {
            continue;
        }

        let angle = calculateAngle(currentPoint, point);

        if (lowestAngle > angle) {
            lowestAngle = angle;
            nextPoint.x = point.x;
            nextPoint.y = point.y;

            if (firstTime) {
                firstTime = false;
            }
            else {
                animationPoints.splice(animationPoints.length - 1, 1);
            }

            animationPoints.push({ x: nextPoint.x, y: nextPoint.y });
        }

        if (globals.isAnimationEnabled) {
            drawJarvisAnimation(globals.ctx, animationPoints, S, point);

            await sleep(400 * globals.animationSpeed);
        }
    }

    convexHull.addPoint(currentPoint.x, currentPoint.y);

    currentPoint.x = nextPoint.x;
    currentPoint.y = nextPoint.y;
}
```

- After that, we will continue in the same fashion except we will look for the point which makes the polar angle with the current point with respect to the negative x axis. This step will continue until we reach P.



```
// march down...
while (! (currentPoint.x == pivot.x && currentPoint.y == pivot.y)) {
    let lowestAngle = Number.MAX_SAFE_INTEGER;
    let firstTime = true;
    for (const point of S) {

        if (point.x == currentPoint.x && point.y == currentPoint.y) {
            continue;
        }

        let angle = calculateAngleRelativeToNegativeX(currentPoint, point);

        if (lowestAngle > angle) {
            lowestAngle = angle;
            nextPoint.x = point.x;
            nextPoint.y = point.y;

            if (firstTime) {
                firstTime = false;
            }
            else {
                animationPoints.splice(animationPoints.length - 1, 1);
            }

            animationPoints.push({ x: nextPoint.x, y: nextPoint.y });
        }

        if (globals.isAnimationEnabled) {
            drawJarvisAnimation(globals.ctx, animationPoints, S, point);

            await sleep(400 * globals.animationSpeed);
        }

    }

    convexHull.addPoint(currentPoint.x, currentPoint.y);

    currentPoint.x = nextPoint.x;
    currentPoint.y = nextPoint.y;
}
```

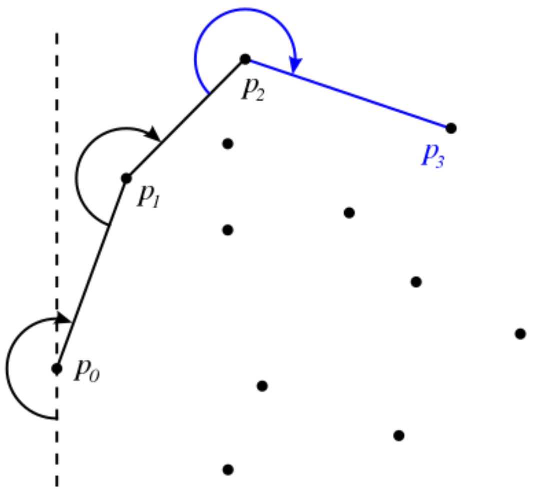


**Fig. 2.** Visual demonstration of Jarvis's March algorithm

## Pseudocode

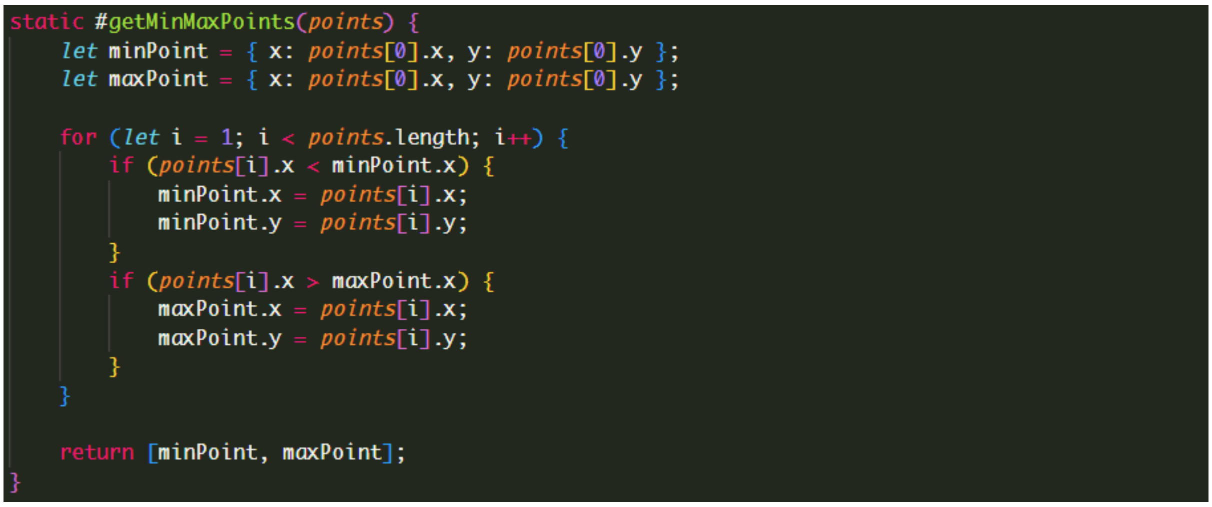
Here is the pseudocode we designed by following the steps above:

```
procedure JarvisMarch(S)
    let P0 be the rightmost lowest y-coordinate point in S

    current_point = P0

    H = []

    while current_point is not highest y-coordinate in S
        let p be the point where P0 makes the lowest polar angle with it wrt x axis
        add p to H
    endwhile

    while current_point is not P0
        let p be the point where P0 makes the lowest polar angle with it wrt negative x axis
        add p to H
    endwhile
    return H
end
```

### 3.3. QuickHull

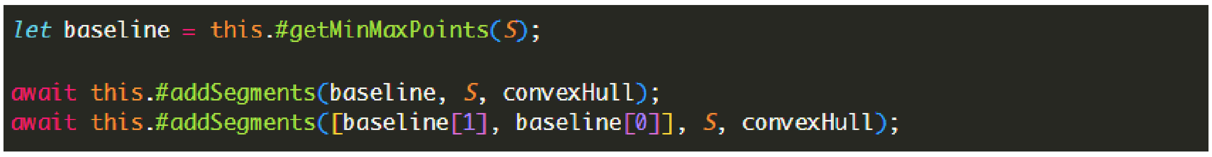
The QuickHull algorithm works on the same principle as the quicksort algorithm. The algorithm, similar to quicksort, has an expected time complexity of O(N logN), and a worst case scenario complexity of $O(N^2)$. The idea is to recursively partition the point set S, finding the convex hulls of each subset. The hull of each subset is constructed by concatenating the convex hulls of the subsets of the subset, much like the quicksort. When this process reaches to set S, the convex hulls of every subset have been concatenated. The convex hull of set S is produced by concatenating the largest remaining convex hulls. The steps of the QuickHull is as follows:

- First, find the points with the minimum and maximum x coordinates. If there are multiple maximum or minimum, choose the points according to their y coordinates.



```
static #getMinMaxPoints(points) {
    let minPoint = { x: points[0].x, y: points[0].y };
    let maxPoint = { x: points[0].x, y: points[0].y };

    for (let i = 1; i < points.length; i++) {
        if (points[i].x < minPoint.x) {
            minPoint.x = points[i].x;
            minPoint.y = points[i].y;
        }
        if (points[i].x > maxPoint.x) {
            maxPoint.x = points[i].x;
            maxPoint.y = points[i].y;
        }
    }

    return [minPoint, maxPoint];
}
```

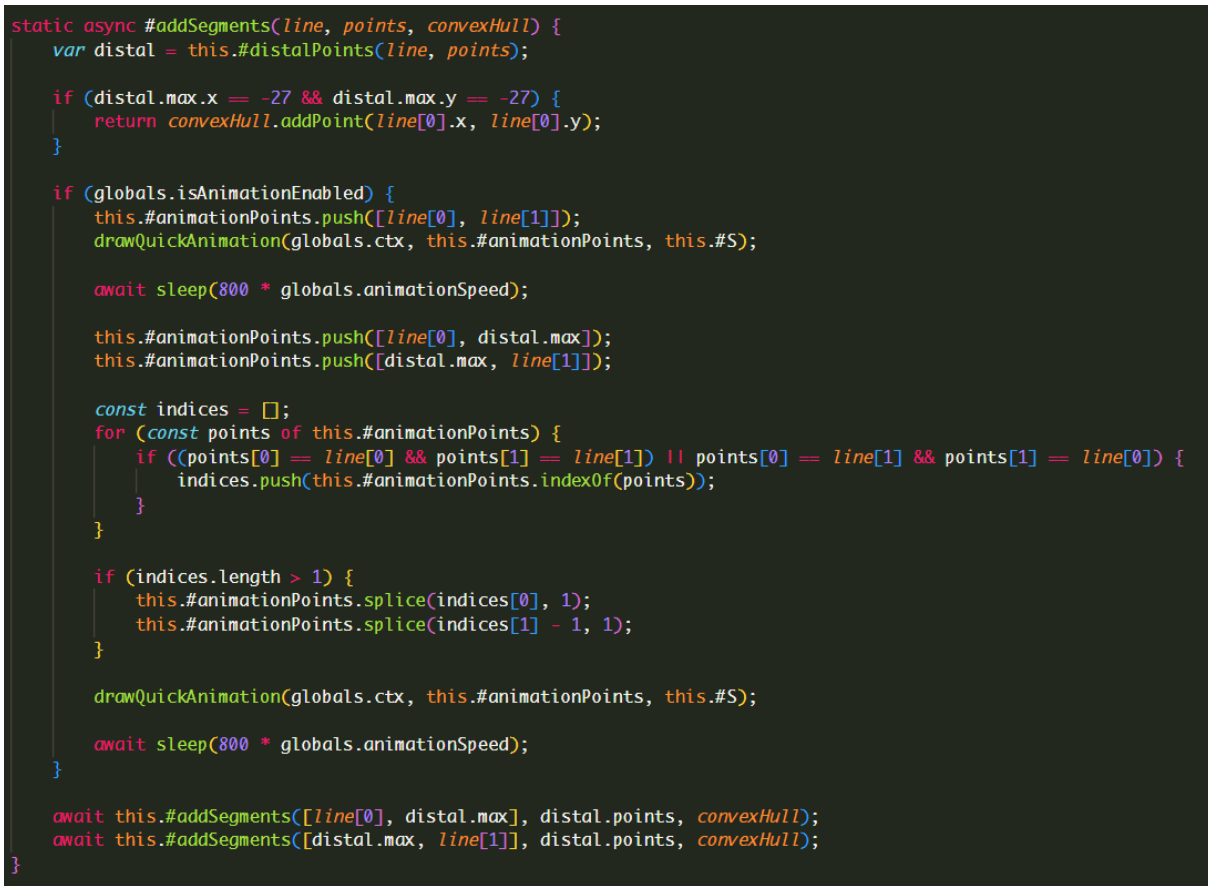
- Use the line formed by these points to separate the rest of the points into two groups. These two groups will be processed recursively.



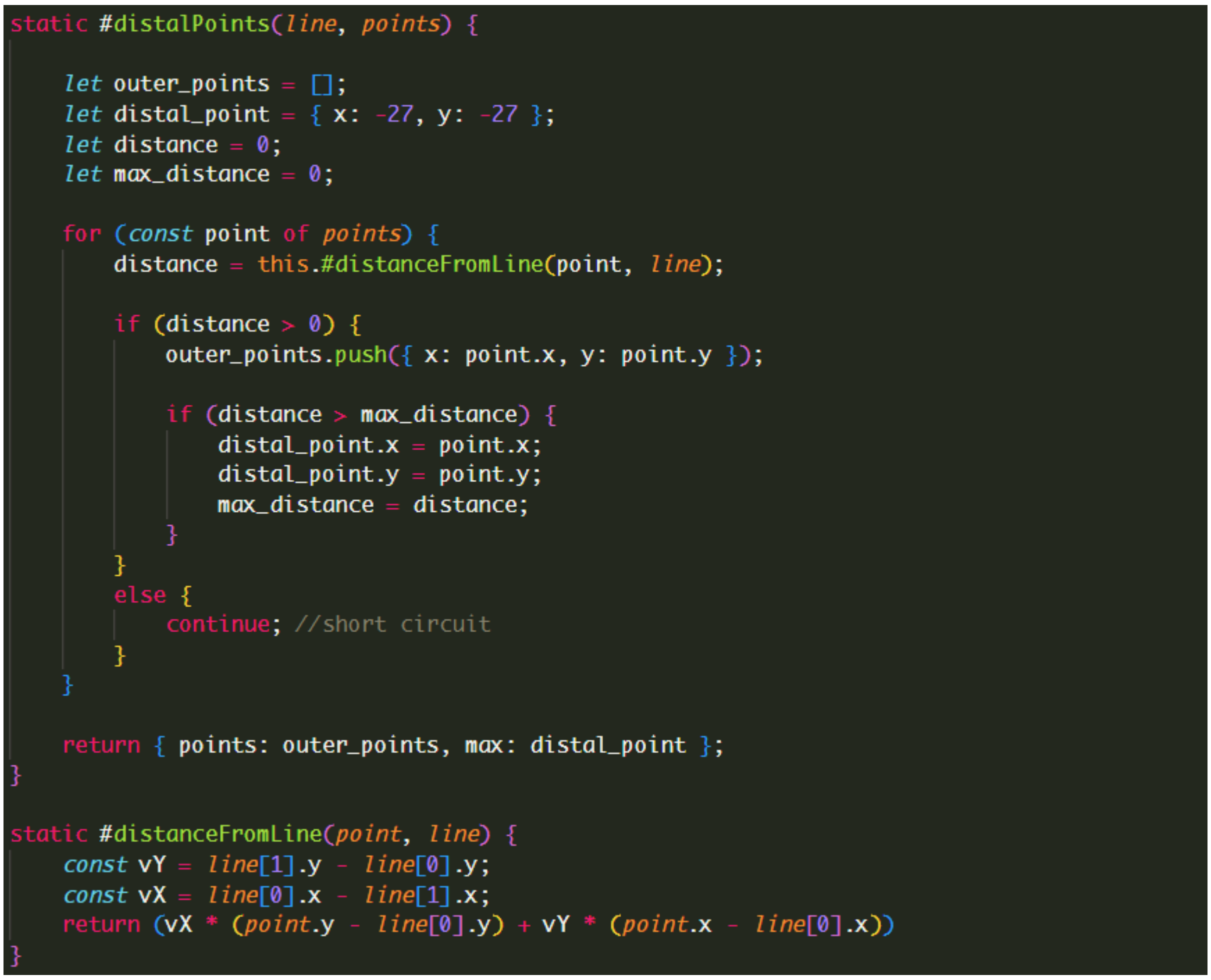
```
let baseline = this.#getMinMaxPoints(S);

await this.#addSegments(baseline, S, convexHull);
await this.#addSegments([baseline[1], baseline[0]], S, convexHull);
```

**Note:** Here, the first call to *#addSegments* is made for the points above the formed line. The second call to *#addSegments* which flips the points on the line, is made for the points below the formed line.

- Select the point that is the furthest away from the line, which forms a triangle. The points in the triangle can't be on the convex hull, they can be ignored for the next steps.



```
static async #addSegments(line, points, convexHull) {
    var distal = this.#distalPoints(line, points);

    if (distal.max.x == -27 && distal.max.y == -27) {
        return convexHull.addPoint(line[0].x, line[0].y);
    }

    if (globals.isAnimationEnabled) {
        this.#animationPoints.push([line[0], line[1]]);
        drawQuickAnimation(globals.ctx, this.#animationPoints, this.#S);

        await sleep(800 * globals.animationSpeed);

        this.#animationPoints.push([line[0], distal.max]);
        this.#animationPoints.push([distal.max, line[1]]);

        const indices = [];
        for (const points of this.#animationPoints) {
            if ((points[0] == line[0] && points[1] == line[1]) || points[0] == line[1] && points[1] == line[0]) {
                indices.push(this.#animationPoints.indexOf(points));
            }
        }

        if (indices.length > 1) {
            this.#animationPoints.splice(indices[0], 1);
            this.#animationPoints.splice(indices[1] - 1, 1);
        }

        drawQuickAnimation(globals.ctx, this.#animationPoints, this.#S);

        await sleep(800 * globals.animationSpeed);
    }

    await this.#addSegments([line[0], distal.max], distal.points, convexHull);
    await this.#addSegments([distal.max, line[1]], distal.points, convexHull);
}
```



```
static #distalPoints(line, points) {

    let outer_points = [];
    let distal_point = { x: -27, y: -27 };
    let distance = 0;
    let max_distance = 0;

    for (const point of points) {
        distance = this.#distanceFromLine(point, line);

        if (distance > 0) {
            outer_points.push({ x: point.x, y: point.y });

            if (distance > max_distance) {
                distal_point.x = point.x;
                distal_point.y = point.y;
                max_distance = distance;
            }
        }
        else {
            continue; //short circuit
        }
    }

    return { points: outer_points, max: distal_point };
}

static #distanceFromLine(point, line) {
    const vY = line[1].y - line[0].y;
    const vX = line[0].x - line[1].x;
    return (vX * (point.y - line[0].y) + vY * (point.x - line[0].x))
}
```

- Recursively repeat the previous step, using the new lines formed until all the points that weren’t in any triangles have been chosen.

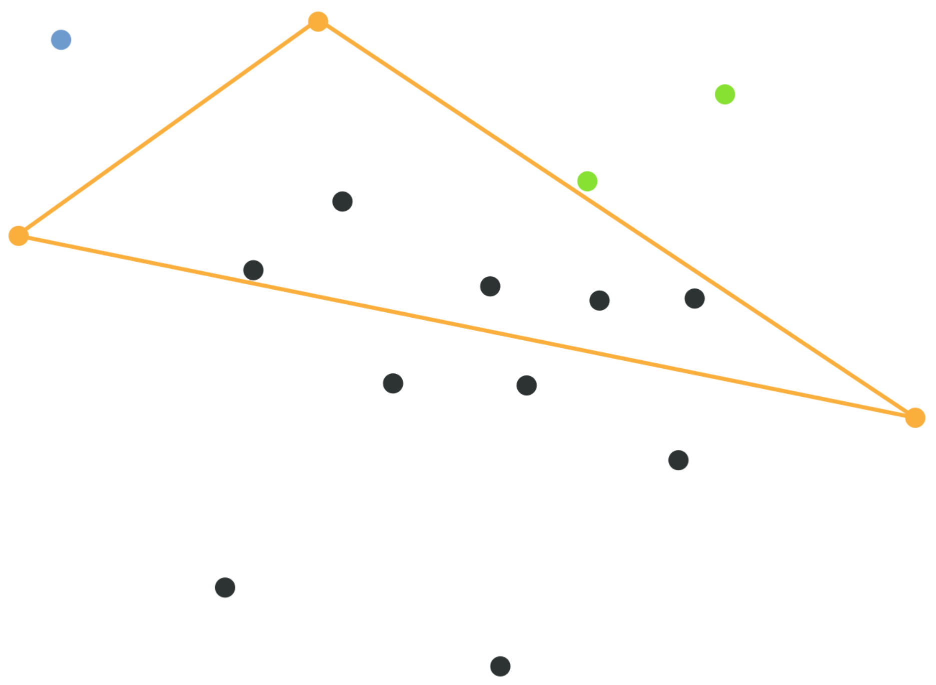


**Fig. 3.** Visual Demonstration of the QuickHull Algorithm

**Pseudocode**

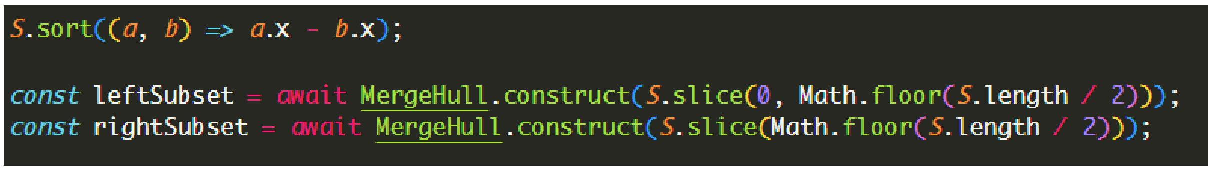
Here is the pseudocode we designed by following the steps above:

```
procedure QuickHull(S)
    let P_min = point with minimum x-coordinate in S
    let P_max = point with maximum x-coordinate in S

    for each point P in S
        if P.x == P_min.x and P.y < P_min.y
            P_min = P
        else if P.x == P_max.x and P.y > P_max.y
            P_max = P

    let convex_hull = []
    convex_hull += FindHull(S, P_min, P_max)
    convex_hull += FindHull(S, P_max, P_min)

    return convex_hull
end

procedure FindHull(S, P1, P2)
    let hull = []
    if S is empty
        return hull

    let farthest_point = point in S with maximum distance to line P1P2
    let S1 = points inside triangle P1, P2, and farthest_point
    let S2 = points inside triangle P2, P1, and farthest_point

    hull += FindHull(S1, P1, farthest_point)
    hull += [farthest_point]
    hull += FindHull(S2, farthest_point, P2)

    return hull
end
```

### 3.4. Merge Hull

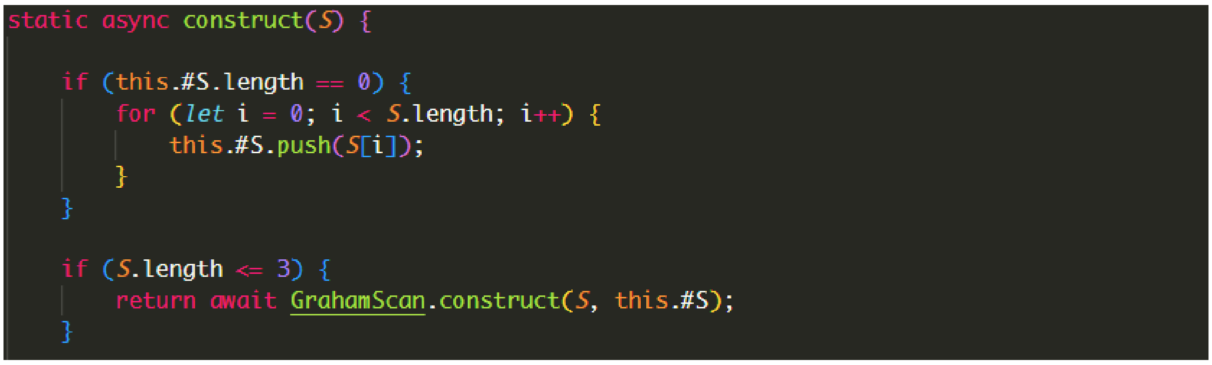
The Merge Hull algorithm functions using the same principle as merge sort, dividing the points into smaller groups, constructing smaller convex hulls and merging them together. The steps for a set of points S is as follows:

- Divide S into two subsets of similar size, using their x coordinates and sorting them into a set. Repeat the step recursively until subsets of size 3 or less remain.



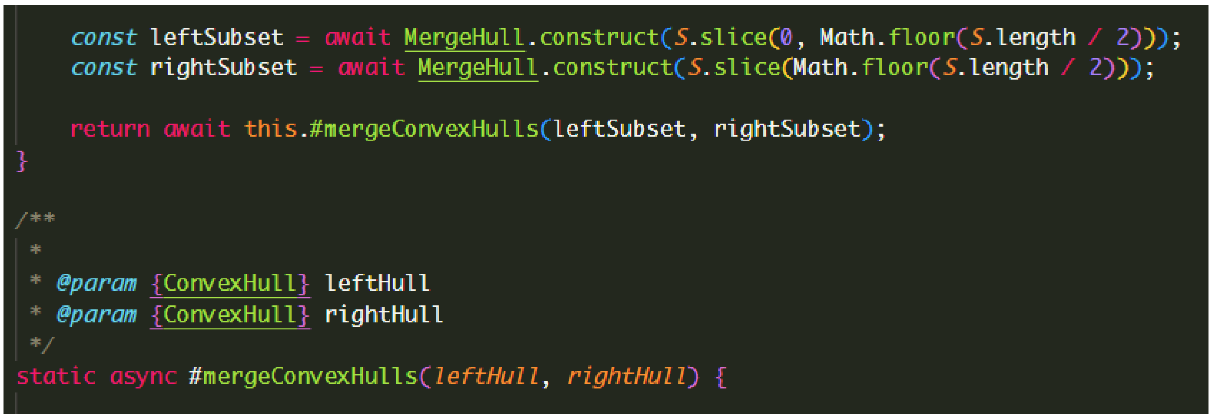
```javascript
S.sort((a, b) => a.x - b.x);

const leftSubset = await MergeHull.construct(S.slice(0, Math.floor(S.length / 2)));
const rightSubset = await MergeHull.construct(S.slice(Math.floor(S.length / 2)));
```

- When the subsets are small enough, use a convex hull construction algorithm like the Graham's Scan to construct the hull for the subsets.



```javascript
static async construct(S) {

    if (this.#S.length == 0) {
        for (let i = 0; i < S.length; i++) {
            this.#S.push(S[i]);
        }
    }

    if (S.length <= 3) {
        return await GrahamScan.construct(S, this.#S);
    }
```

- Start merging the convex hulls to construct the hull for S.



```
    const leftSubset = await MergeHull.construct(S.slice(0, Math.floor(S.length / 2)));
    const rightSubset = await MergeHull.construct(S.slice(Math.floor(S.length / 2)));

    return await this.#mergeConvexHulls(leftSubset, rightSubset);
}

/**
 *
 * @param {ConvexHull} leftHull
 * @param {ConvexHull} rightHull
 */
static async #mergeConvexHulls(leftHull, rightHull) {
```

**Pseudocode**

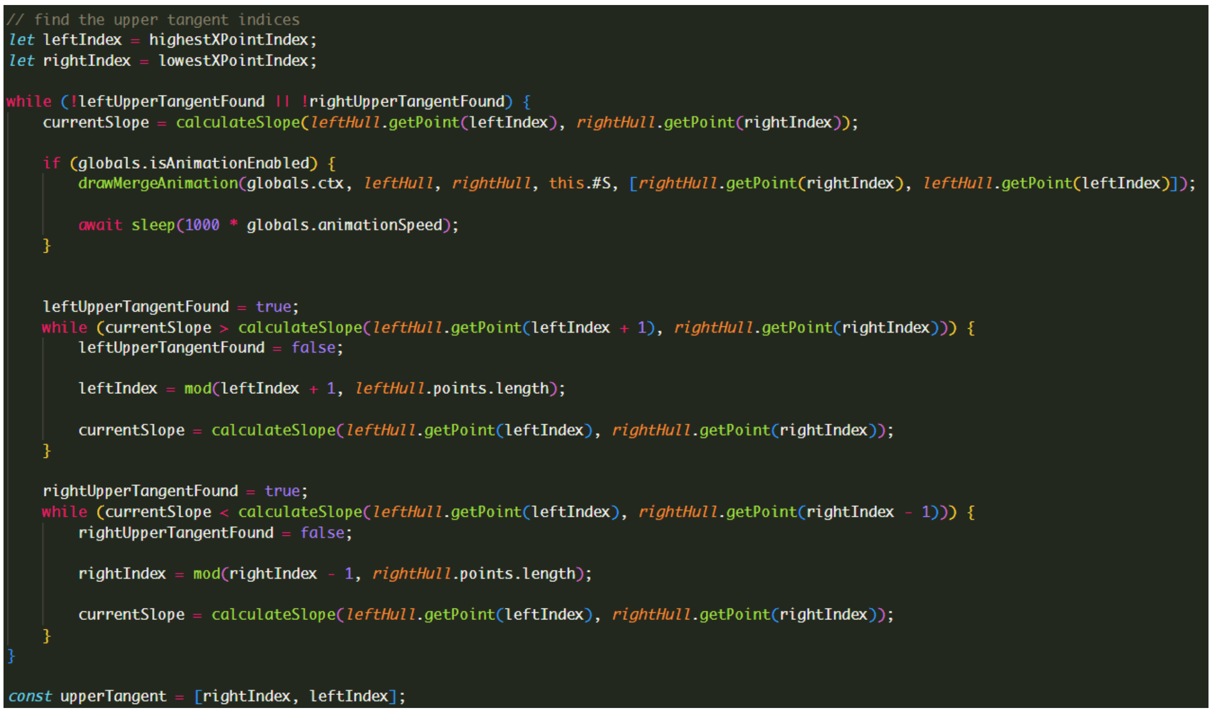
Here is the pseudocode we designed by following the steps above:

```
procedure merge_hull(points)
   sorted_points = sort(points)
   if len(sorted_points) <= 3:
      return graham_scan(sorted_points)
   left_subset = sorted_points[:len(sorted_points)//2]
   right_subset = sorted_points[len(sorted_points)//2:]
   left_hull = merge_hull(left_subset)
   right_hull = merge_hull(right_subset)
   merged_hull = merge_convex_hulls(left_hull, right_hull)
   return merged_hull
end
```

**Merge Algorithm For Merge Hull**

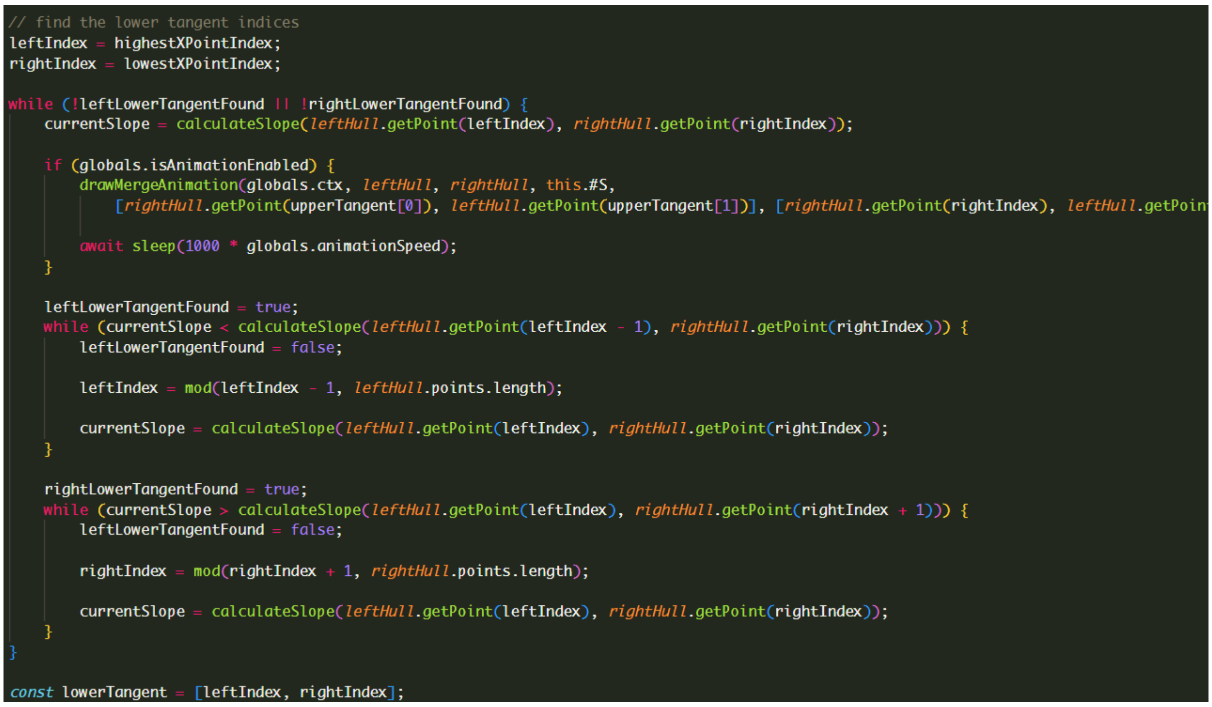
The merge algorithm for merging left and right hulls for merge hull is the following:

- Draw a line between the highest x coordinate point of the left hull and lowest x coordinate point of the right hull.
- To find the upper tangent, move the point on the left hull counter-clockwise as long as the slope of the line decreases. Likewise move the point on the right hull clockwise as long as the slope of the line increases.



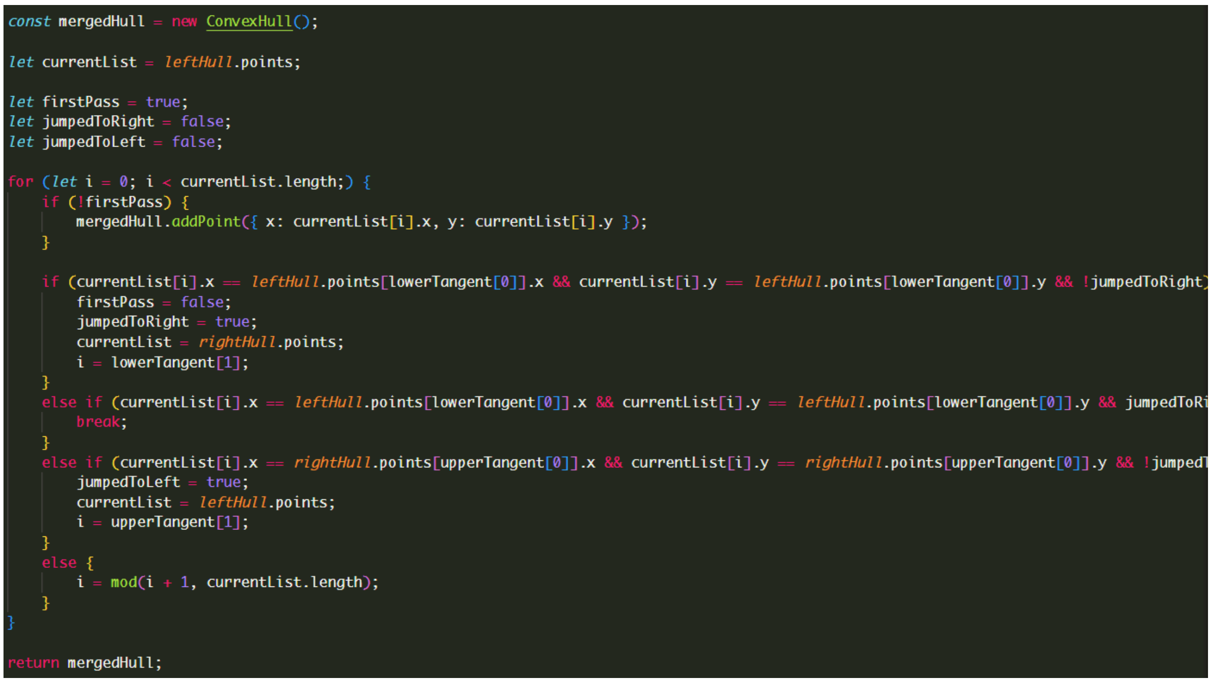
```
// find the upper tangent indices
let leftIndex = highestXPointIndex;
let rightIndex = lowestXPointIndex;

while (!leftUpperTangentFound || !rightUpperTangentFound) {
    currentSlope = calculateSlope(leftHull.getPoint(leftIndex), rightHull.getPoint(rightIndex));

    if (globals.isAnimationEnabled) {
        drawMergeAnimation(globals.ctx, leftHull, rightHull, this.#S, [rightHull.getPoint(rightIndex), leftHull.getPoint(leftIndex)]);

        await sleep(1000 * globals.animationSpeed);
    }


    leftUpperTangentFound = true;
    while (currentSlope > calculateSlope(leftHull.getPoint(leftIndex + 1), rightHull.getPoint(rightIndex))) {
        leftUpperTangentFound = false;

        leftIndex = mod(leftIndex + 1, leftHull.points.length);

        currentSlope = calculateSlope(leftHull.getPoint(leftIndex), rightHull.getPoint(rightIndex));
    }

    rightUpperTangentFound = true;
    while (currentSlope < calculateSlope(leftHull.getPoint(leftIndex), rightHull.getPoint(rightIndex - 1))) {
        rightUpperTangentFound = false;

        rightIndex = mod(rightIndex - 1, rightHull.points.length);

        currentSlope = calculateSlope(leftHull.getPoint(leftIndex), rightHull.getPoint(rightIndex));
    }
}

const upperTangent = [rightIndex, leftIndex];
```

- Draw a line between the highest x coordinate point of the left hull and lowest x coordinate point of the right hull.
- To find the lower tangent, move the point on the left hull clockwise as long as the slope of the line increases. Likewise move the point on the right hull counter-clockwise as long as the slope of the line decreases.



```
// find the lower tangent indices
leftIndex = highestXPointIndex;
rightIndex = lowestXPointIndex;

while (!leftLowerTangentFound || !rightLowerTangentFound) {
    currentSlope = calculateSlope(leftHull.getPoint(leftIndex), rightHull.getPoint(rightIndex));

    if (globals.isAnimationEnabled) {
        drawMergeAnimation(globals.ctx, leftHull, rightHull, this.#S,
            [rightHull.getPoint(upperTangent[0]), leftHull.getPoint(upperTangent[1])], [rightHull.getPoint(rightIndex), leftHull.getPoin

        await sleep(1000 * globals.animationSpeed);
    }

    leftLowerTangentFound = true;
    while (currentSlope < calculateSlope(leftHull.getPoint(leftIndex - 1), rightHull.getPoint(rightIndex))) {
        leftLowerTangentFound = false;

        leftIndex = mod(leftIndex - 1, leftHull.points.length);

        currentSlope = calculateSlope(leftHull.getPoint(leftIndex), rightHull.getPoint(rightIndex));
    }

    rightLowerTangentFound = true;
    while (currentSlope > calculateSlope(leftHull.getPoint(leftIndex), rightHull.getPoint(rightIndex + 1))) {
        leftLowerTangentFound = false;

        rightIndex = mod(rightIndex + 1, rightHull.points.length);

        currentSlope = calculateSlope(leftHull.getPoint(leftIndex), rightHull.getPoint(rightIndex));
    }
}

const lowerTangent = [leftIndex, rightIndex];
```

- When you find both the upper tangent and the lower tangent, since the left and right hulls are guaranteed to be ordered counter-clockwise, (because we use Graham's Scan in the base step), iterate through the points but also by jumping over the tangents.



```
const mergedHull = new ConvexHull();

let currentList = leftHull.points;

let firstPass = true;
let jumpedToRight = false;
let jumpedToLeft = false;

for (let i = 0; i < currentList.length;) {
    if (!firstPass) {
        mergedHull.addPoint({ x: currentList[i].x, y: currentList[i].y });
    }

    if (currentList[i].x == leftHull.points[lowerTangent[0]].x && currentList[i].y == leftHull.points[lowerTangent[0]].y && !jumpedToRight
        firstPass = false;
        jumpedToRight = true;
        currentList = rightHull.points;
        i = lowerTangent[1];
    }
    else if (currentList[i].x == leftHull.points[lowerTangent[0]].x && currentList[i].y == leftHull.points[lowerTangent[0]].y && jumpedToRi
        break;
    }
    else if (currentList[i].x == rightHull.points[upperTangent[0]].x && currentList[i].y == rightHull.points[upperTangent[0]].y && !jumped
        jumpedToLeft = true;
        currentList = leftHull.points;
        i = upperTangent[1];
    }
    else {
        i = mod(i + 1, currentList.length);
    }
}

return mergedHull;
```

```
procedure merge_convex_hulls(left_hull, right_hull)
    upper_tangent, lower_tangent = compute_tangents(left_hull, right_hull)
    merged_hull = merge_hulls(left_hull, right_hull, upper_tangent, lower_tangent)
    return merged_hull
end

procedure compute_tangents(left_hull, right_hull)
    upper_tangent = compute_upper_tangent(left_hull, right_hull)
    lower_tangent = compute_lower_tangent(left_hull, right_hull)
    return upper_tangent, lower_tangent
end

procedure merge_hulls(left_hull, right_hull, upper_tangent, lower_tangent)
    merged_hull = remove_unnecessary_parts(left_hull, right_hull, upper_tangent,
lower_tangent)
    return merged_hull
end
```

## 3.5. Data Structures

We will simply use an ordered list of points to store the convex hull. The list will contain the vertices of the convex hull in a counter-clockwise order.

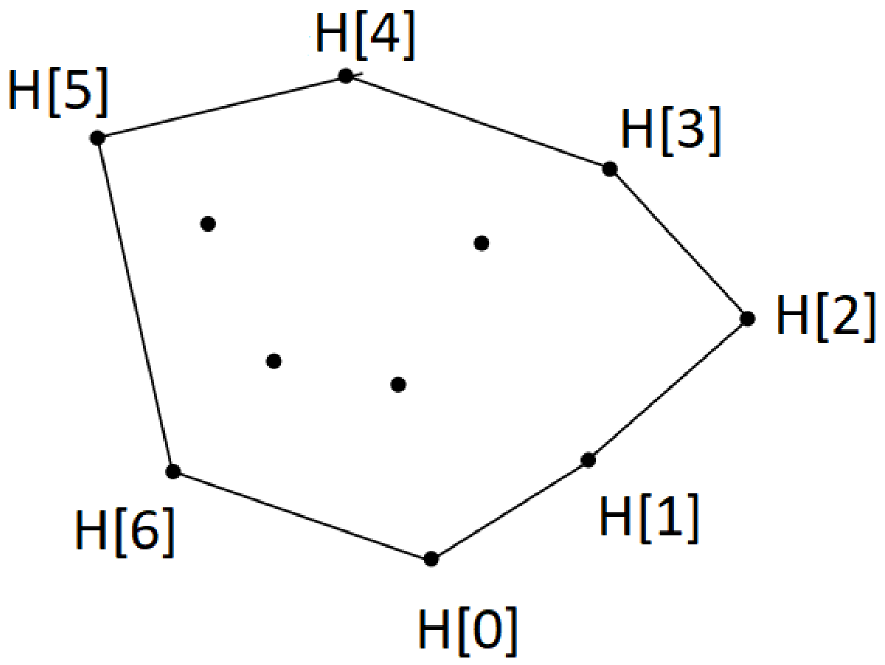
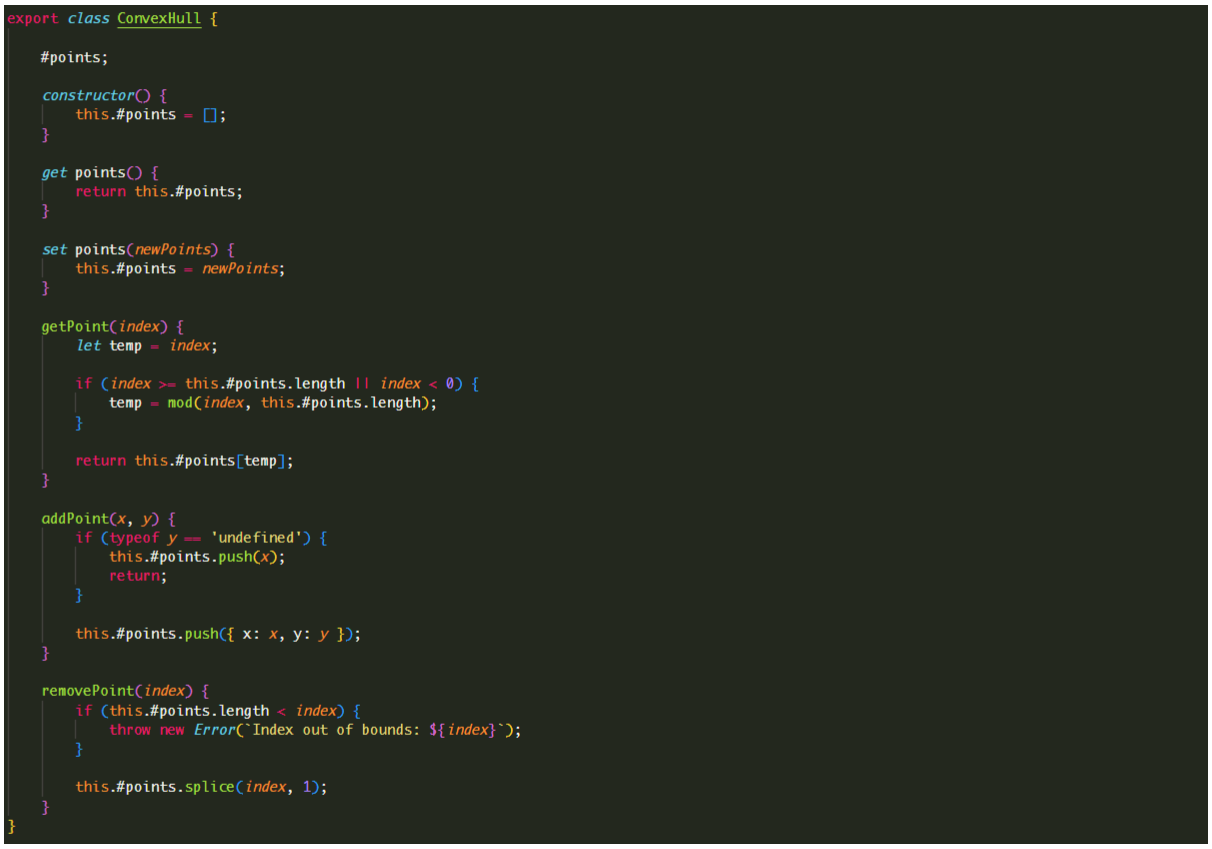


**Fig. 4.** Ordered list for storing a convex hull

And here is our actual implementation for the convex hull structure:

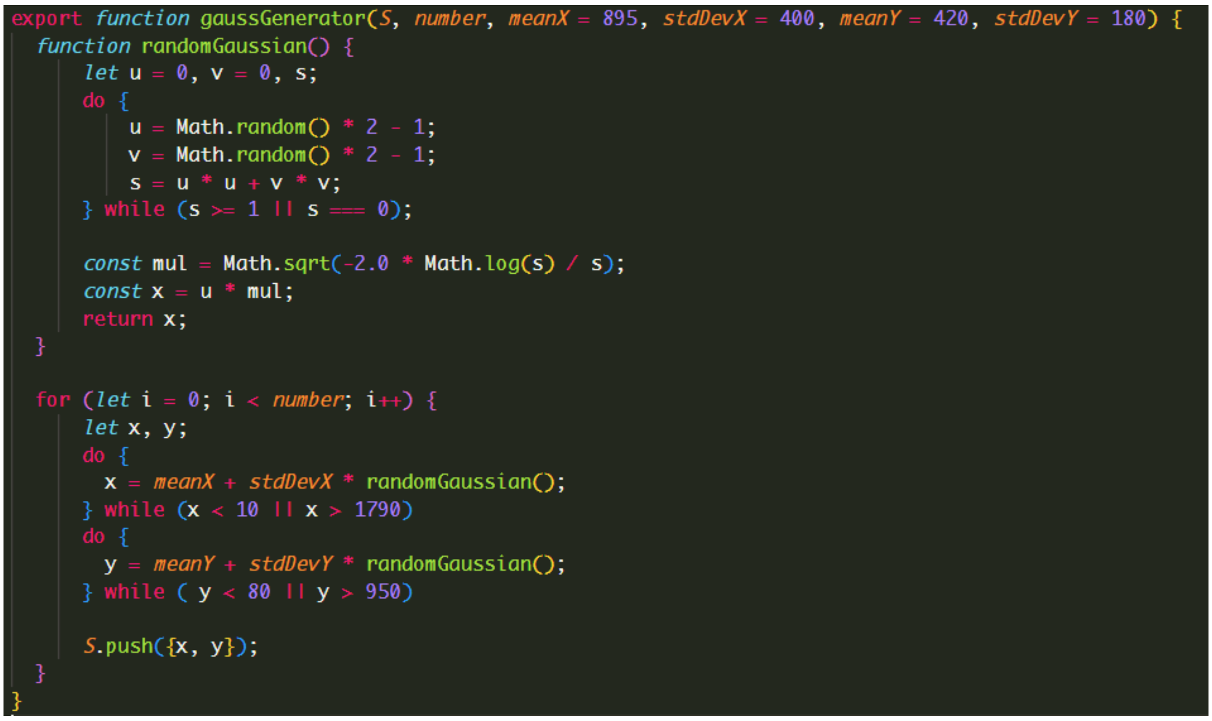


```javascript
export class ConvexHull {

    #points;

    constructor() {
        this.#points = [];
    }

    get points() {
        return this.#points;
    }

    set points(newPoints) {
        this.#points = newPoints;
    }

    getPoint(index) {
        let temp = index;

        if (index >= this.#points.length || index < 0) {
            temp = mod(index, this.#points.length);
        }

        return this.#points[temp];
    }

    addPoint(x, y) {
        if (typeof y === 'undefined') {
            this.#points.push(x);
            return;
        }

        this.#points.push({ x: x, y: y });
    }

    removePoint(index) {
        if (this.#points.length < index) {
            throw new Error(`Index out of bounds: ${index}`);
        }

        this.#points.splice(index, 1);
    }
}
```

## 4. Other Implementation Details

### 4.1. Gaussian (Normal) Distribution

We used Box-Muller Transformation to implement a function in JavaScript which will generate random numbers based on a gaussian distribution:



```
export function gaussGenerator(S, number, meanX = 895, stdDevX = 400, meanY = 420, stdDevY = 180) {
  function randomGaussian() {
      let u = 0, v = 0, s;
      do {
        u = Math.random() * 2 - 1;
        v = Math.random() * 2 - 1;
        s = u * u + v * v;
      } while (s >= 1 || s === 0);

      const mul = Math.sqrt(-2.0 * Math.log(s) / s);
      const x = u * mul;
      return x;
  }

  for (let i = 0; i < number; i++) {
      let x, y;
      do {
        x = meanX + stdDevX * randomGaussian();
      } while (x < 10 || x > 1790)
      do {
        y = meanY + stdDevY * randomGaussian();
      } while ( y < 80 || y > 950)

      S.push({x, y});
  }
}
```

### 4.2. Uniform Distribution

The function (Math.random) used to generate random numbers in JavaScript already uses uniform distribution [2].

### 4.3. Rendering

For visualizing the algorithms, we will be using the 2D context of the HTML canvas. The 2D context will provide us tools to draw points and lines on the canvas. The HTML canvas runs at 60 frames per second so it will also allow us to do step-by-step animations of the algorithms [3].

```
<html>
<body>
  <canvas id="myCanvas" width="200" height="200" />
  <script>
    var canvas = document.getElementById("myCanvas");
    var ctx = canvas.getContext("2d");
    ctx.fillStyle = "red";
    ctx.fillRect(0, 0, 200, 200);
    ctx.fillStyle = "green";
    ctx.fillRect(50, 50, 100, 100);
    ctx.fillStyle = "blue";
    ctx.fillRect(75, 75, 50, 50);
  </script>
</body>
</html>
```

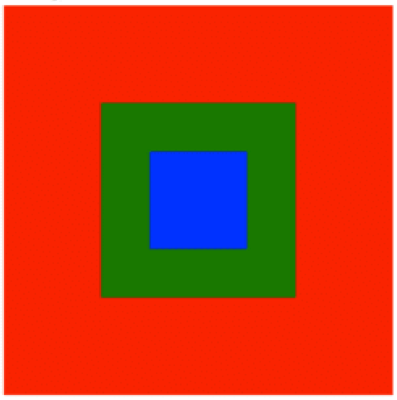
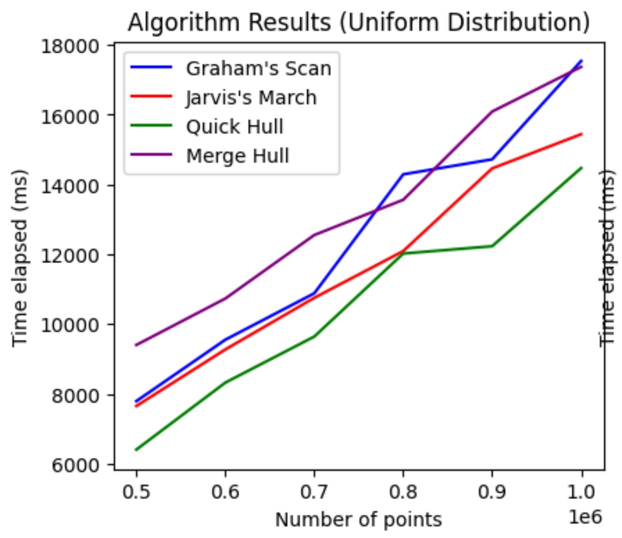
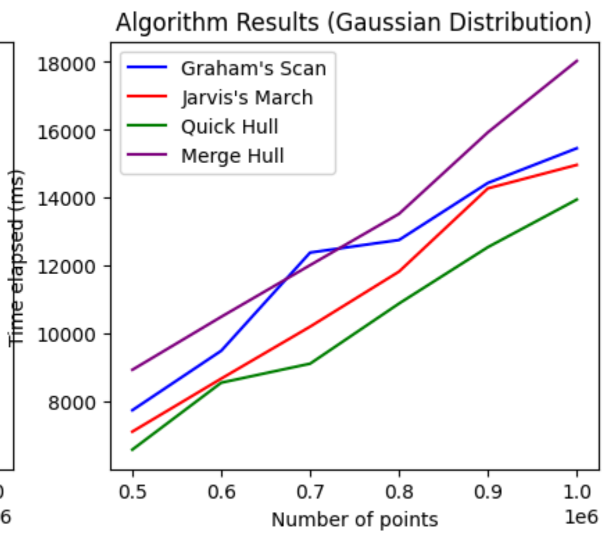


**Fig. 5.** Creating and using 2D context for HTML Canvas in JavaScript

## 5. Results

We used Matplotlib with Python to compare the runtime running time of the different algorithms [4]. Our analysis was made with point counts between 500K - 1M with 100K increments. Also, our running time is in terms of milliseconds, (ms).





Here are important remarks drawn from the runtime analysis:

- Quick Hull is the fastest algorithm in practice with both uniform and gaussian distribution.
- All the algorithms generally performed better with gaussian distribution than uniform distribution.
- Jarvis's March performed better when there were less points on the convex hull.

## 6. Conclusion

In conclusion, the convex hull is a very important structure in Computational Geometry. It is used to find the smallest convex polygon enclosing a set of points. Our aim is to implement different algorithms for constructing convex hulls and comparing their performances. We will also visualize the algorithms step-by-step and allow dynamic convex hulls, (adding or removing points after they are constructed). Finally, we will record our findings and prepare a report and a presentation.

## 7. References

[1] Rockafellar, R. Tyrrell (1970), *Convex Analysis*, Princeton Mathematical Series, vol. 28, Princeton, N.J.: Princeton University Press, 0274683

[2] Math.random() - JavaScript
| MDN
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/random

[3] Canvas API - Web APIs | MDN -
https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API

[4] Matplotlib — Visualization with Python | Matplotlib: Visualization with Python -
https://matplotlib.org