Computational Geometry (2025) - Convex Hull

Diego Barbieri
au802245@uni.au.dk
Aarhus University

Gioele Scandaletti
au803277@uni.au.dk
Aarhus University

Samuele Esposito
au803278@uni.au.dk
Aarhus University

Abstract

Convex Hull algorithms implementations and analysis.

Contents

1 Introduction

This report compares the results obtained from implementing some Convex Hulls algorithms in both 2D.

We decided to implement the tests in C++ due to its performance, the availability of standard libraries and ease of handling with abstract data types.

It's important to note that the performance in this case are highly dependent on the hardware and the specific implementation details. Moreover, we will run the comparison on the same machine to ensure a fair comparison, with the following specifications:

- CPU: Intel i5-1135G7, 4 cores, 8 threads, 4.2 GHz
- RAM: 8 GB
- OS: Arch Linux - Kernel 6.16.7-arch1-1

1.1 Utility Classes and Functions

To manage the 2d plane we defined a simple Point and Line class, that would handle the sidedness tests and distance calculations(see Listing 1).

```
class Point {
public:
    float x;
    float y;
    // ...
};
void showValue(const Point &person, std::ostream &os);
class Line {
public:
    Point p1;
    Point p2;
    Line(float m, float q);
    Line(const Point& p1, const Point& p2);
    // ...
};
class Triangle {
public:
    Point p1;
    Point p2;
    Point p3;
    Triangle(const Point& p1, const Point& p2, const Point& p3);
};
```

Listing 1: Point and Line class definitions

Another implementation strategy that we decided to adopt was to define a ConvexHull interface (see Listing 2) that would be implemented by each algorithm class. This way we could easily switch between algorithms and have a common method to call.

```
/* Convex Hull Interface */
template <typename T>
class ConvexHull {
public:
    /* Every algorithm must implement the lower and upper hull and merge them */
```

```
    virtual T compute(const T& points) const = 0;
};
using Points = std::vector<Point>;
```

Listing 2: ConvexHull interface definition

At last we defined some aliases to make the code more readable:

```
using TPoint = std::pair<Point, Point>;
using Points = std::vector<Point>;
```

Listing 3: Type aliases for Points and TPoint

1.2 Benchmarking strategy

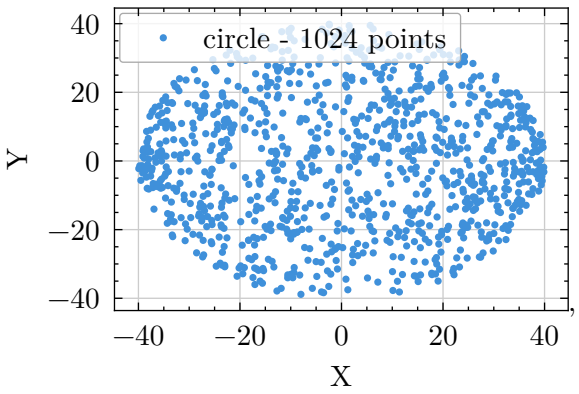
To archive the most precise results possible we decided to adopt the Google Benchmarking library made ad hoc for C++. The code reads the input from a txt file containing the points coordinates and runs the benchmark for a predefined number of iterations.

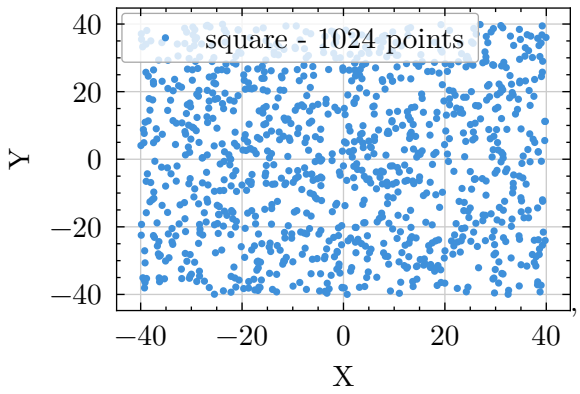
The point generation is done through a python script that creates different distributions (circle, square, parabola).

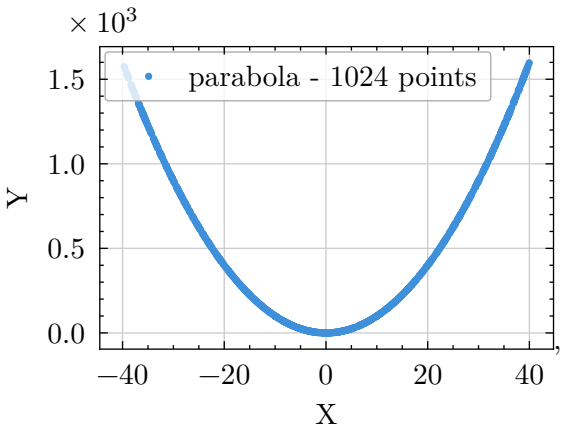
```
# ...
types = {
    "parabola": lambda x: x**2,
    "square": lambda x: np.random.uniform(LOWER_X, UPPER_X),
    "circle": lambda x: np.random.uniform(-(R**2 - x**2) ** 0.5, ((R**2 - x**2) **
0.5)),
}

for shape in types.keys():
    for size in [2**i for i in range(8, 20)]:
        LOWER_X = -10.0 * (size / 256)
        UPPER_X = 10.0 * (size / 256)
        R = 10.0 * (size / 256)

        # generate unique x values
        x_vals = set()
        limit_low, limit_high = (LOWER_X, UPPER_X) if shape != "circle" else (-R, R)

        # keep generating until enough UNIQUE x values
        while len(x_vals) < size:
            x = round(random.uniform(limit_low, limit_high), 3)
            x_vals.add(x)

        x_vals = sorted(x_vals)
        y_vals = [types[shape](x) for x in x_vals]

        # writing to files
        # ...
```

Listing 4: Python code snippet for point generation

The following lines from the script generates different distributions of points:







The results are then saved in a csv file that can be later analyzed and plotted directly from Typst.

2 Convex Hulls

2.1 Graham's Scan

2.1.1 Algorithm description

Graham's Scan is an algorithm to find the convex hull for a given set of points. The main idea of the algorithm is to consider the "top" and "bottom" halves of the hull separately, merging the results at the end. Our implementation of the algorithm can be found in Appendix A.

Exactly this final merging step brings forward some practical considerations: what is the practical performance impact of this final operation? We do some experiments around that in section Section 2.1.4.

2.1.2 Complexity Analysis

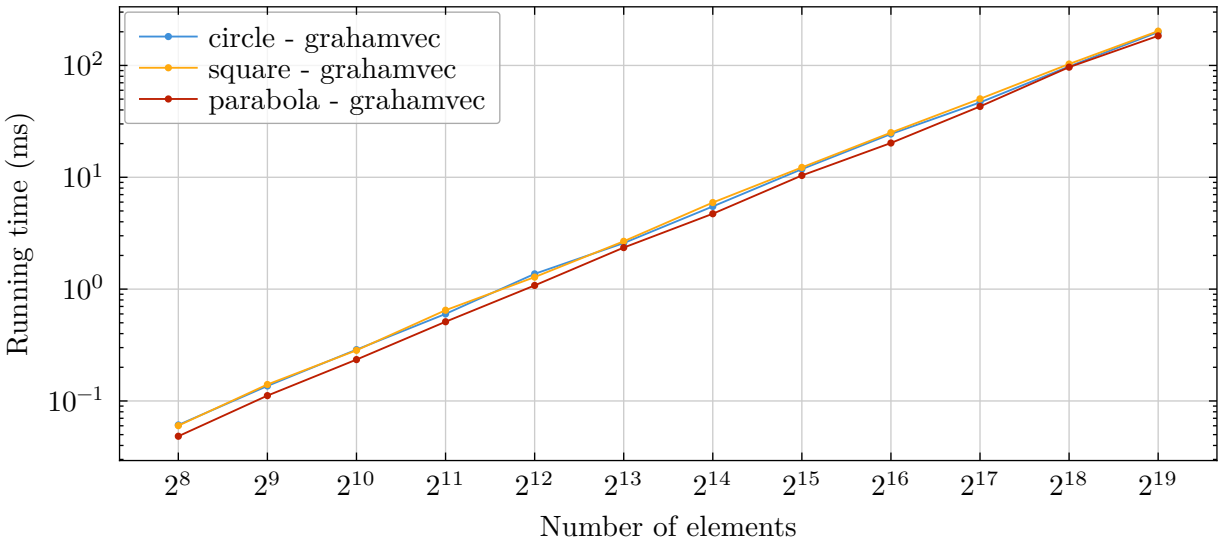
Let's analyze the time complexity of the Graham's Scan algorithm:

- The first step is sorting the points, which takes $O(n \log n)$ time.
- Then, it computes the two half-hulls. Since doing that requires iterating over all of the points, this takes $O(2n) = O(n)$ time.
- Finally, it merges the two resulting hulls, which again takes $O(n)$ time.

Thus, the overall time complexity is $O(n \log n)$.

2.1.3 Benchmark results

We expected the Graham's Scan algorithm to perform with a time complexity of $O(n \log n)$. From the benchmark though we can observe that the parabola case takes less time than the other cases. This is because the parabola, being continuous, makes the calculation of the lower hull extremely simple, since no "backtracking" is necessary.

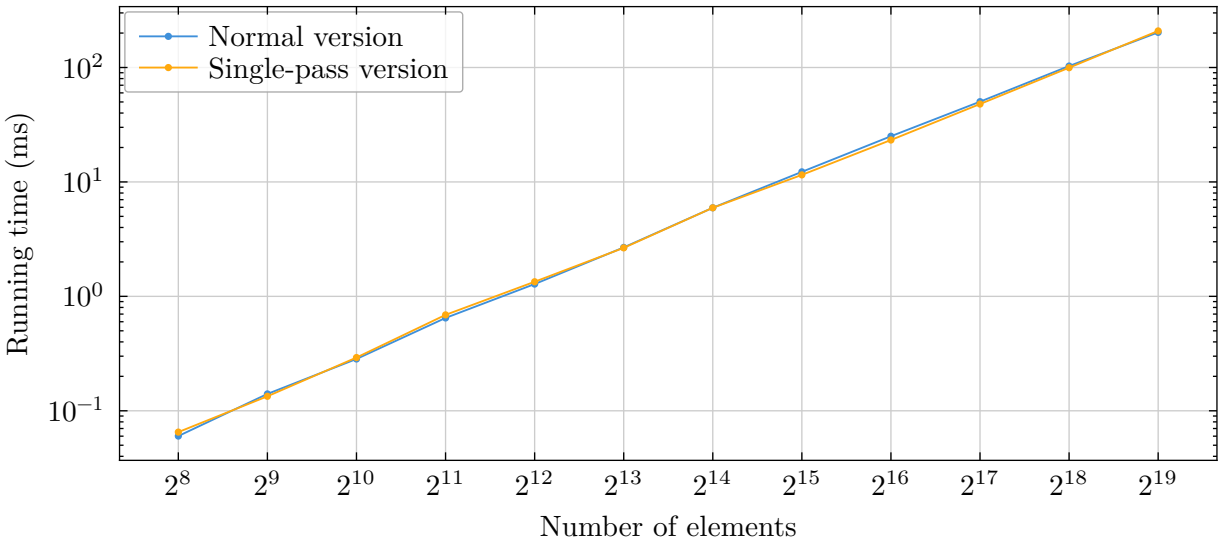


2.1.4 Graham's Scan in a single pass

We also experimented with removing the final step of the algorithm, where the two halves are merged together to form the final hull. To this end, we developed a variation of the algorithm that works with containers that support both insertion from the back and insertion from the front.

The main idea was to only loop over the points once, inserting them both at the front and at the back of the container; afterwards, we check both ends of the container for the turning direction they have.

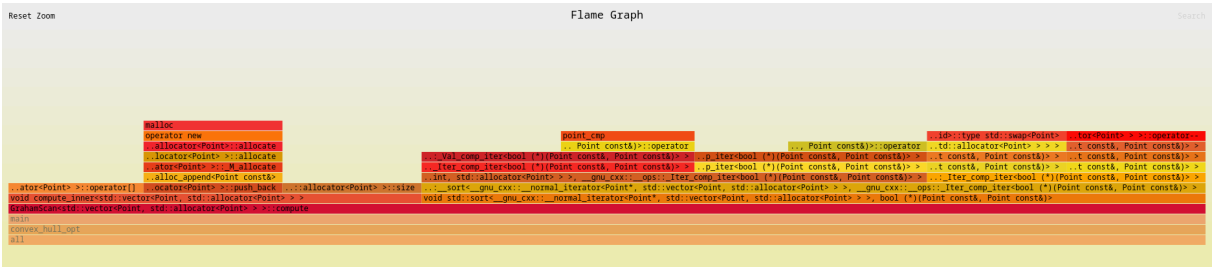
The code (in this example, using the `std::deque` container) can be found in Appendix B.



From the benchmark above we can see that this change doesn't affect the running time. We hypothesized that this was due to differences in the cache behaviour, either between the two

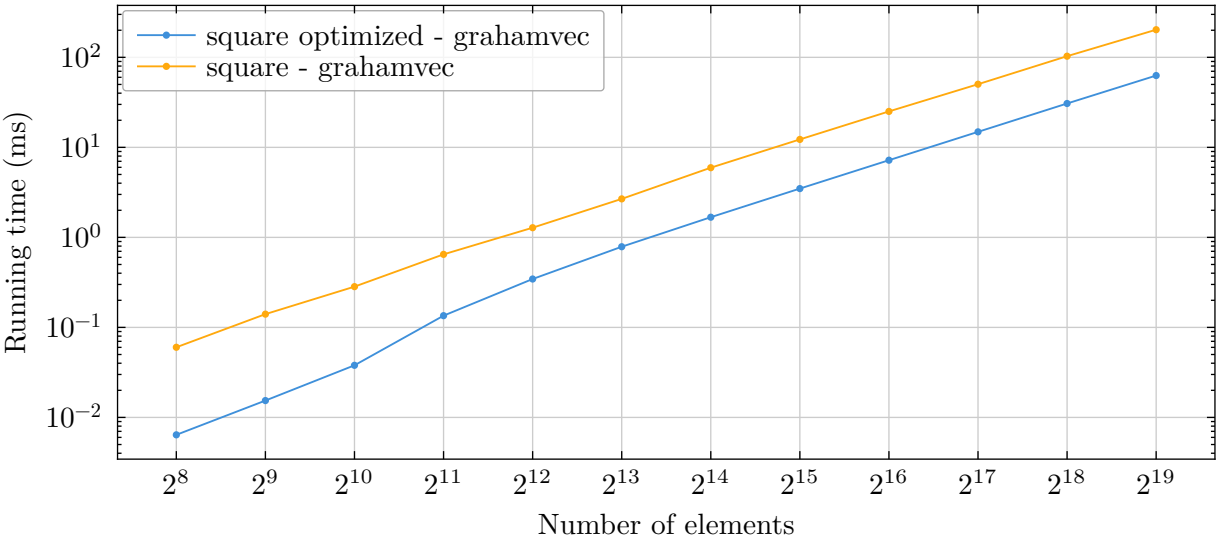
underlying data structures, or because we insert separately at the beginning and at the end. We tried running `perf stat` on our code, and found that that was not what was happening; instead, the cache misses increased by just 3%.

We tried collecting a FlameGraph[1] to investigate, and found that, in the "normal" case, the merging takes so little time that it is invisible on the graph.



2.1.5 Optimizations

In order to have a more fair comparison we decided to also benchmark the code compiled with optimizations turned on. The results are shown below.



2.2 QuickHull Algorithm

2.2.1 Implementation

In the following section, we will present the implementation of the QuickHull algorithm and the benchmarks we performed to evaluate its performance.

Figure 2: QuickHull working principle

[1]https://www.brendangregg.com/flamegraphs.html

Firstly, we will introduce some of the utility function created to support the core algorithm(see Listing 5).

```
std::pair<TPoint, TPoint> findExtremePointsCases(const Points &points) {
  Point leftPointYMin = points[0];
  Point leftPointYMax = points[0];
  Point rightPointYMin = points[0];
  Point rightPointYMax = points[0];

  for (const auto &p : points) {
    if (p.x < leftPointYMin.x) { // new leftmost point found
      leftPointYMin = p;
      leftPointYMax = p;
    } else if (p.x == leftPointYMin.x) { // same x-coordinate as current leftmost,
check y-coordinates
      if (p.y < leftPointYMin.y) {
        leftPointYMin = p;
      }
      if (p.y > leftPointYMax.y) {
        leftPointYMax = p;
      }
    }

    if (p.x > rightPointYMax.x) { // new rightmost point found
      rightPointYMin = p;
      rightPointYMax = p;
    } else if (p.x == rightPointYMax.x) { // same x-coordinate as current rightmost,
check y-coordinates
      if (p.y < rightPointYMin.y) {
        rightPointYMin = p;
      }
      if (p.y > rightPointYMax.y) {
        rightPointYMax = p;
      }
    }
  }
  return {{leftPointYMin, leftPointYMax}, {rightPointYMin, rightPointYMax}};
}
```

Listing 5: Function to find extreme points with vertical alignment cases

The function shown above finds the extreme points in a set of 2D points, specifically the leftmost and rightmost points, while handling cases where multiple points share the same x-coordinate but have different y-coordinates (i.e., vertical alignments). The full algorithm implementation is shown below Appendix C.

As we have mentioned before, the compute function is the core of all the ConvexHull classes. In this particular case, it initializes the QuickHull algorithm by finding the extreme points and partitioning the input points into upper and lower subsets. It then calls the recursive functions to compute the upper and lower hulls, finally combining them into a single convex hull.

A lot of the extra steps are made to cover edge cases where multiple points share the same x-coordinate, ensuring that the algorithm correctly identifies and processes these points without introducing duplicates in the final hull.

The recursive functions used to compute the upper and lower hulls are shown below Appendix D.

2.2.2 Complexity Analysis

Let's analyze the time complexity of the QuickHull algorithm:

- The algorithm starts by finding the extreme points, which takes $O(n)$ time.
- The partitioning of points into upper and lower subsets also takes $O(n)$ time.
- The recursive step involves finding the point with the maximum distance from the line segment, which takes O(n) time in the worst case.
- Then we recur two times on subsets of points $O(2T(\frac{n}{2}))$.

Let's now set up the recurrence relation for the time complexity:

$$T(n) = T(k) + T(n-k) + O(n) \tag{1}$$

Where k is the number of points in one of the subsets. In the worst case, k can be as large as $n-1$, leading to unbalanced partitions. This results in a time complexity of $O(n^2)$ in the worst case.

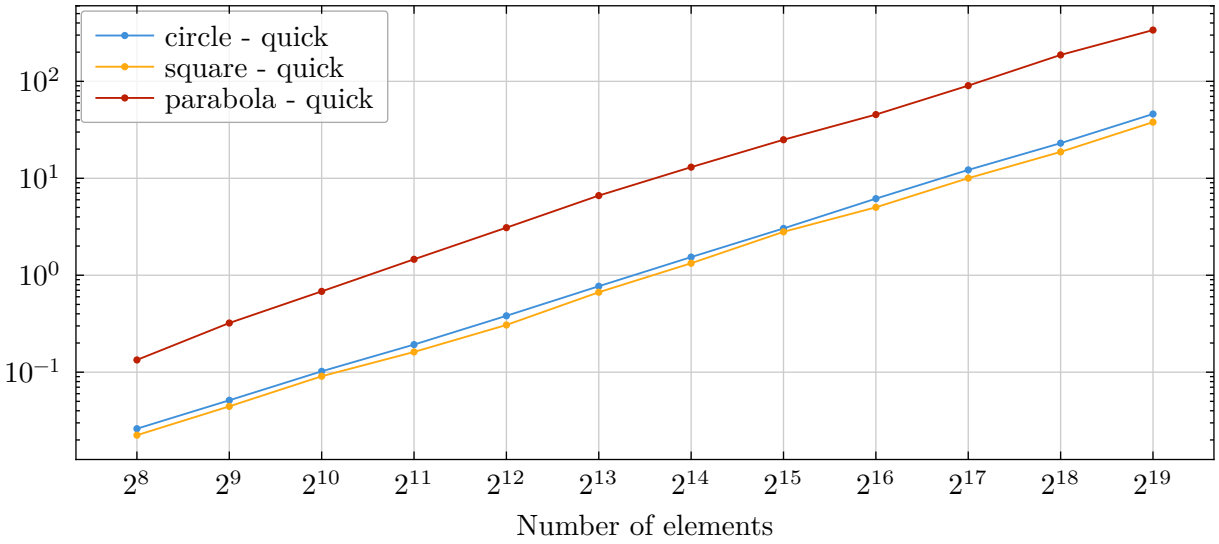
2.2.3 Benchmarks

In our benchmarks, we expect the QuickHull algorithm to perform efficiently on average, it's conjectured that on average the algorithms runs in $O(n \log n)$ time when the input precision is restricted to $O(\log n)$ (see quickhull-average).

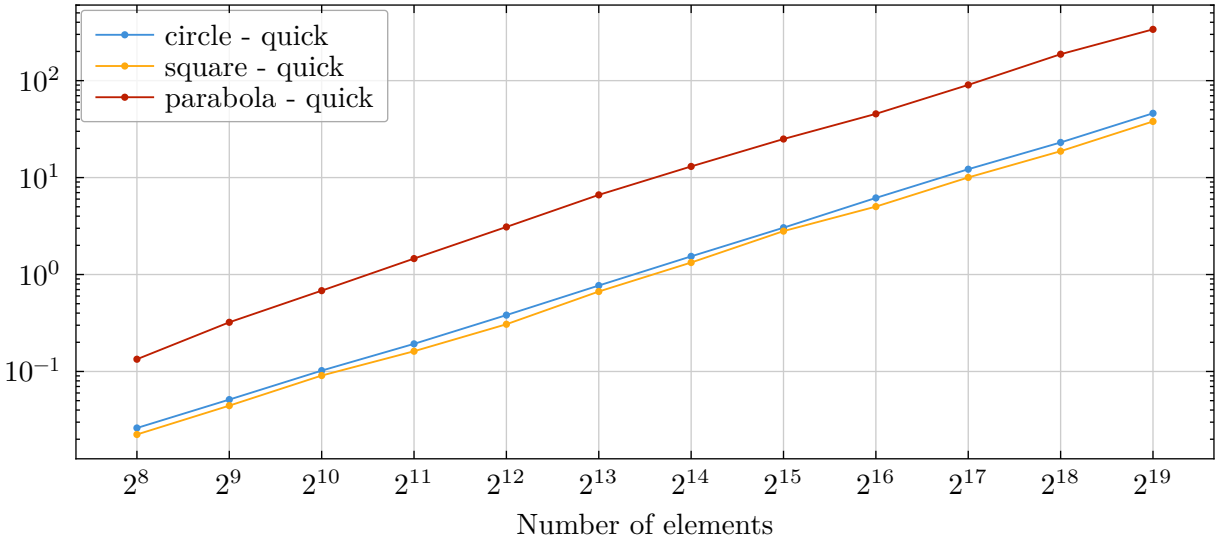
However, for some testcases such as a parabola distribution, we might observe performance degradation towards the worst-case scenario of $O(n^2)$.

The results of our benchmarks shown in the diagram below respects our expectations, with QuickHull performing well on circular and square distributions, while showing increased computation time on the parabola distribution as the number of points increases.

Following are the results for the QuickHull algorithm:

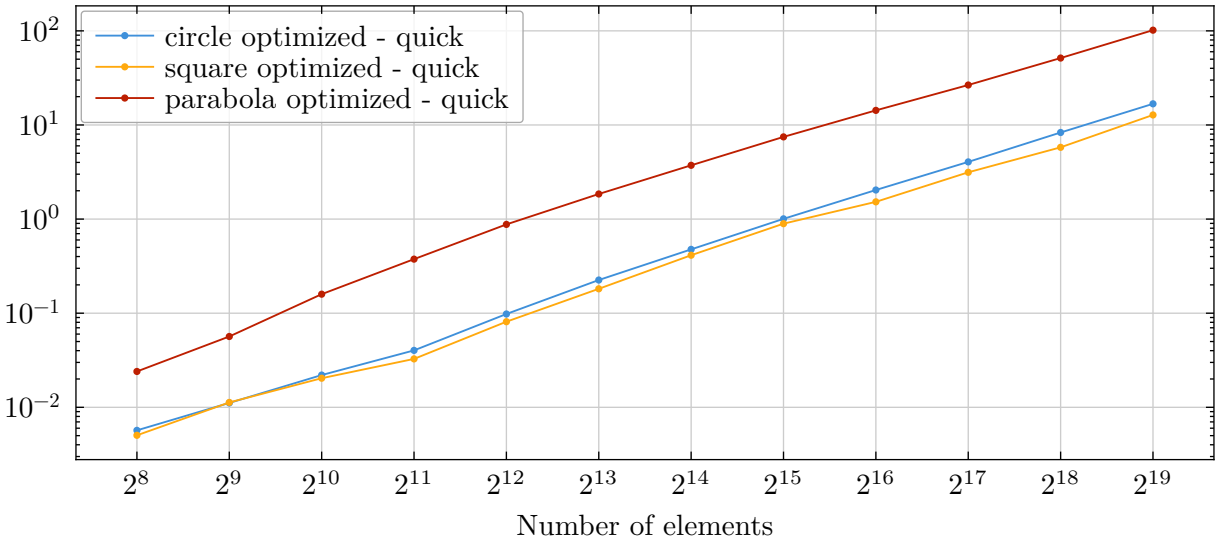


And in log-log scale.



2.2.4 Optimizations

In order to have a more fair comparison we decided to test the algorithm on also a compiler-optimized version with the -O3 flag enabled. The results are shown below.



As we could expect, the optimization provided by the compiler improved the performance by a 5-10x factor across all the different distributions. But the overall behavior of the algorithm remained the same.

In addition, we've also measured time taken for each function inside the implementation with a flame graph (see Figure 3).

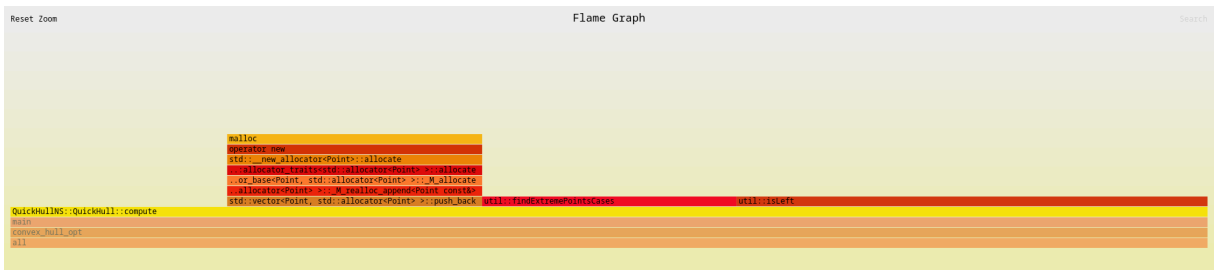


Figure 3: Flame graph for QuickHull algorithm

The results reported respect our expectations, as the majority is equally split between the `isLeft`, `findExtremePointsCases`, `std::push_back` functions.

2.3 Marriage Before Conquest Algorithm

2.3.1 Implementation

In the following section, we will present the implementation of the Marriage Before Conquest algorithm and the benchmarks we performed to evaluate its performance.

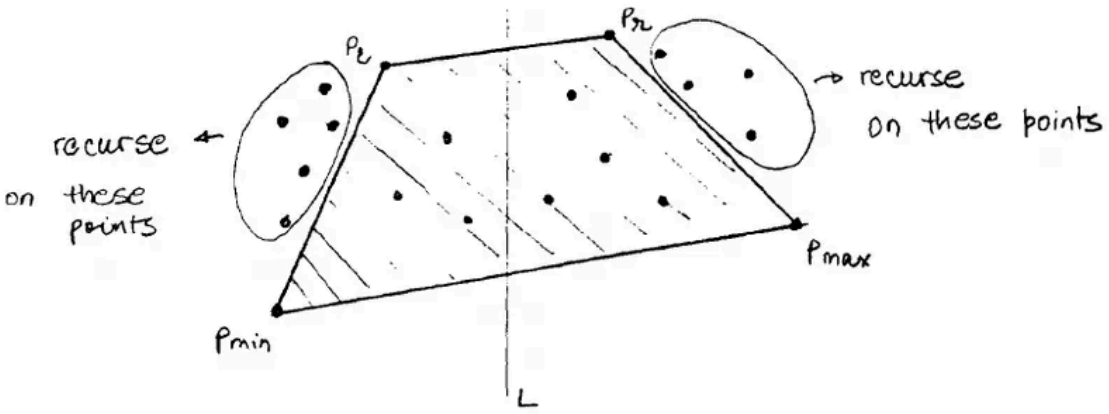


Figure 4: Marriage Before Conquest working principle

Firstly, we will introduce some of the utility function created to support the core algorithm(see).

```
bool isLeft(const Point &p1, const Point &p2, const Point &p3) {
  const double dx_32 = p3.x - p2.x;
  const double dy_12 = p1.y - p2.y;
  const double dy_32 = p3.y - p2.y;
  const double dx_12 = p1.x - p2.x;

  return (dx_32 * dy_12) > (dy_32 * dx_12);
}
```

Listing 6: Function to find the relative position of a point with respect to a line segment

The function above checks if a point p3 is to the left of the line segment formed by points p1 and p2. This is useful for determining the relative position of points during the construction of the convex hull.

```
Line findExtremePoints(const Points &points, bool upper) {
  // Find leftmost and rightmost points with highest y in case of ties
  Point minPoint = points[0];
  Point maxPoint = points[0];
  if (upper) {
    for (const auto &p : points) {
      if (p.x < minPoint.x || (p.x == minPoint.x && p.y > minPoint.y)) {
        minPoint = p;
      }
      if (p.x > maxPoint.x || (p.x == maxPoint.x && p.y > maxPoint.y)) {
        maxPoint = p;
      }
    }
    return Line{minPoint, maxPoint};
```

```
  } else {
    for (const auto &p : points) {
      if (p.x < minPoint.x || (p.x == minPoint.x && p.y < minPoint.y)) {
        minPoint = p;
      }
      if (p.x > maxPoint.x || (p.x == maxPoint.x && p.y < maxPoint.y)) {
        maxPoint = p;
      }
    }
    return Line{maxPoint, minPoint};
  }
}
```

Listing 7: Function to find extreme horizontal points with upper or lower hull cases

The function shown above finds the extreme points in a set of 2D points, specifically the leftmost and rightmost points, while handling cases where multiple points share the same x-coordinate but have different y-coordinates, based on whether we are considering the upper or lower hull.

The algorithm implementation is shown below Appendix E.

The compute function implements the core logic of the Marriage Before Conquest algorithm. It first checks if the number of points is less than or equal to two, in which case it simply returns the points as they already form a convex hull. For larger sets of points, it initializes an empty hull and shuffles the input points randomly to ensure a good distribution for the algorithm's performance. It then calls the recursive functions for constructing the upper and lower hulls, finally ensuring that the leftmost point is not duplicated in the final hull.

The MBCUpperRecursive function is a recursive implementation of the upper hull construction in the Marriage Before Conquest algorithm. It handles base cases for small sets of points and recursively finds the upper bridge, partitioning the points into left and right subsets for further processing, after finding a valid bridge.

The *findUpperBridge* function identifies the upper bridge for a given set of points in the Marriage Before Conquest algorithm. It initializes the bridge with two points and iteratively refines it by checking the relative positions of other points, ensuring that the bridge correctly represents the upper boundary of the convex hull.

The algorithm for finding the upper bridge takes $O(n)$ expected time, but in the worst case, it can take $O(n^2)$ time due to the nested loops when updating the bridge points. That's why the algorithm relies on randomization (in the *compute* function) to achieve better average performance.

It is worth noting that the implementation of the lower hull construction and the corresponding bridge finding function follows a similar logic, adjusted for the lower hull case.

2.3.2 Complexity Analysis

Let's analyze the time complexity of the Marriage Before Conquest algorithm.

The algorithm starts by shuffling the points, which takes $O(n)$ time.

Finding a bridge takes $O(n)$ time. Points that cannot contribute to the convex hull on either side of the median line are discarded. The algorithm then proceeds recursively on the remaining points to compute the upper and lower parts of the convex hull.

For a balanced partitioning on the median

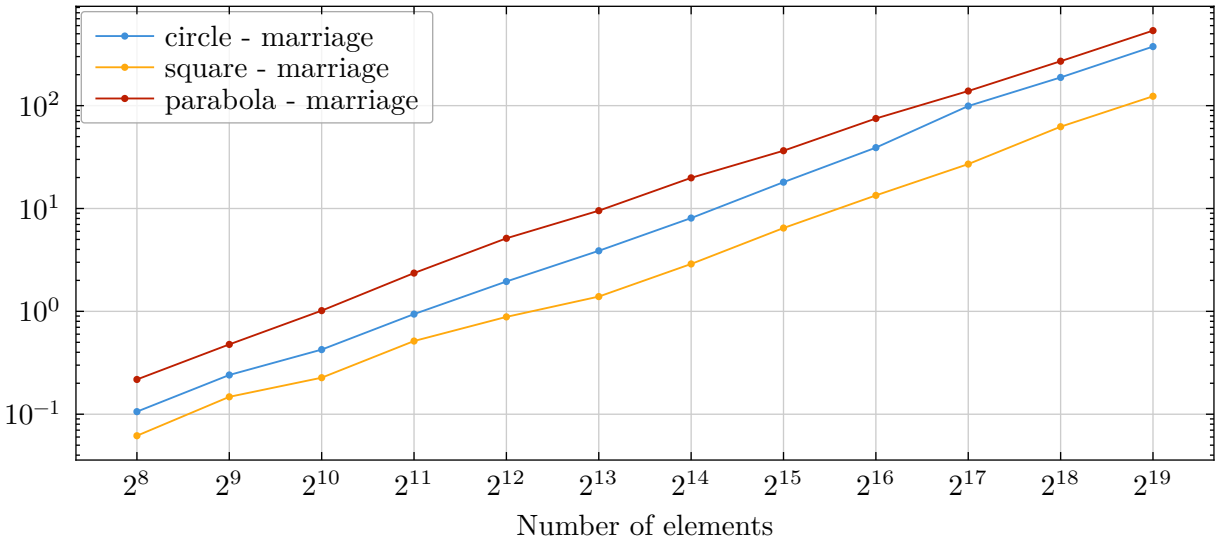
At each recursion level i, the algorithm solves at most 2^i subproblems, each containing at most $\frac{n}{2^i}$ points. Since each subproblem identifies a single edge of the convex hull, the total number of subproblems is bounded by h, the number of points on the hull. In the worst case, when no points can be discarded early, the recursion depth is $O(\log h)$, and each level processes $O(n)$ points in linear time. This results in an overall time complexity of $O(n \log h)$. This makes the Marriage Before Conquest algorithm an output-sensitive algorithm, as its performance depends on the size of the output (the number of points on the convex hull) rather than just the input size.

2.3.3 Benchmarks

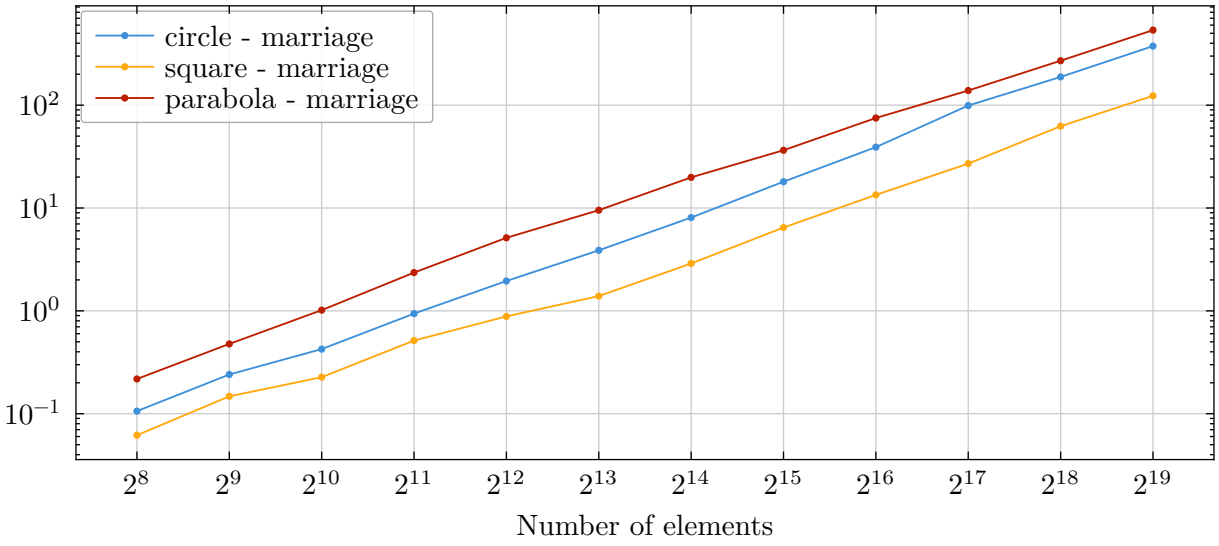
In our benchmarks, we expect the Marriage Before Conquest algorithm to perform efficiently on average, with a time complexity of $O(n \log n)$ for random distributions of points.

The results of our benchmarks shown in the diagram below respects our expectations, with Marriage Before Conquest performing well on circular and square distributions, while showing increased computation time on the parabola distribution as the number of points on the hull increases, since it is output-sensitive.

Following up the results for the Marriage Before Conquest algorithm in linear scale:

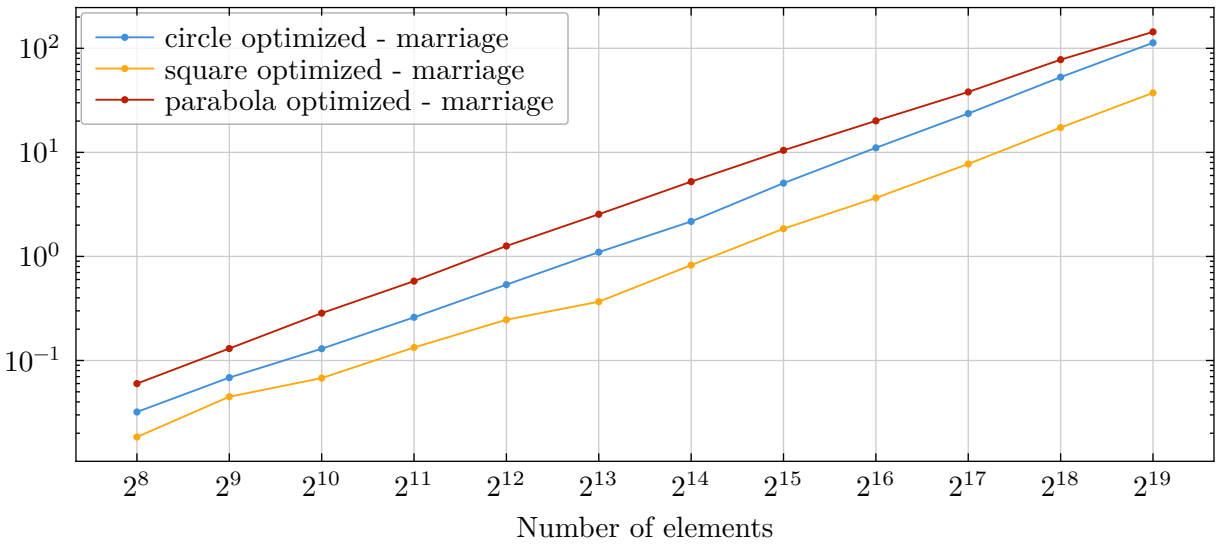


And in log-log scale.



2.3.4 Optimizations

In order to have a more fair comparison we decided to test the algorithm on also a compiler-optimized version with the -O3 flag enabled. The results are shown below.



As we could expect, the optimization provided by the compiler improved the performance by a small constant factor across all the different distributions, so overall the behavior of the algorithm remained the same.

In addition, we've also measured time taken for each function inside the implementation with a flame graph (see Figure 5).

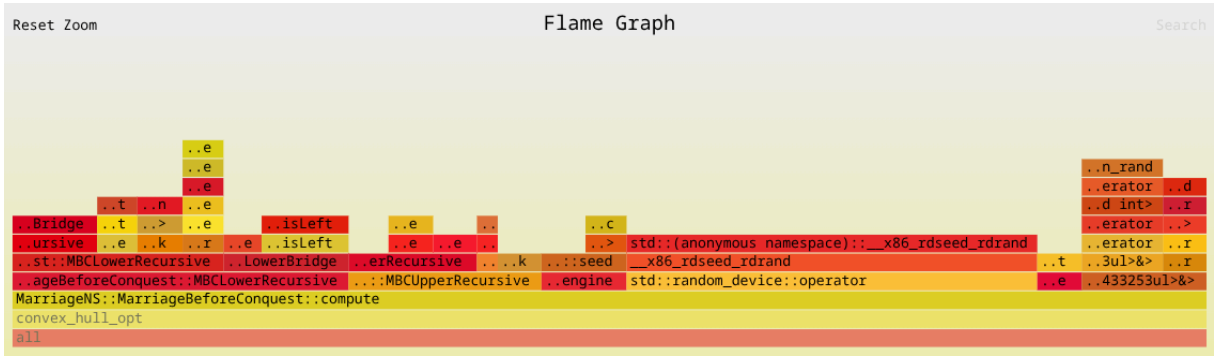
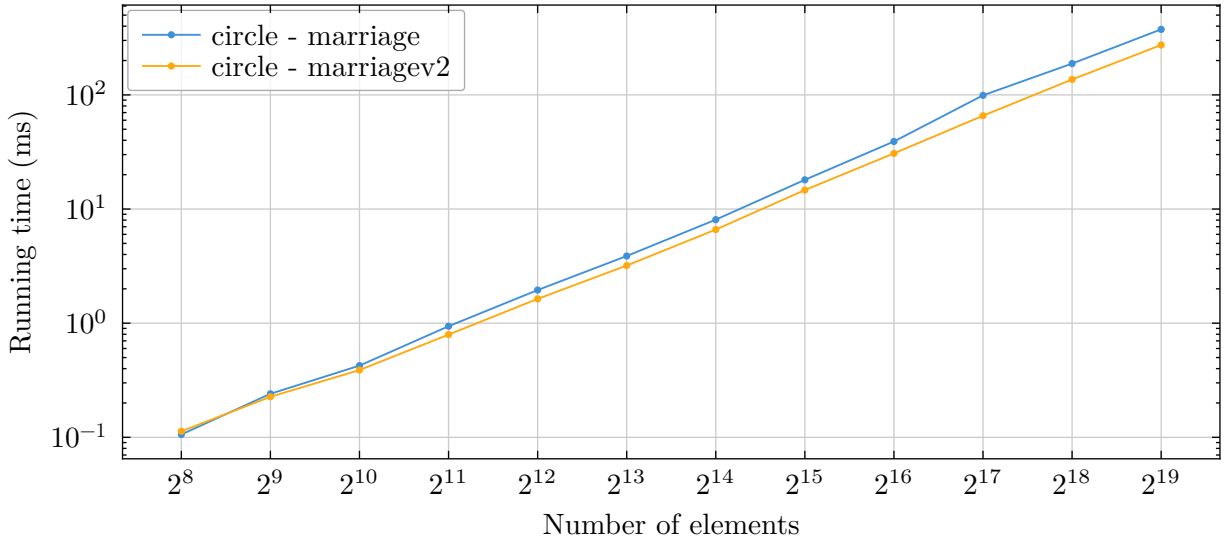


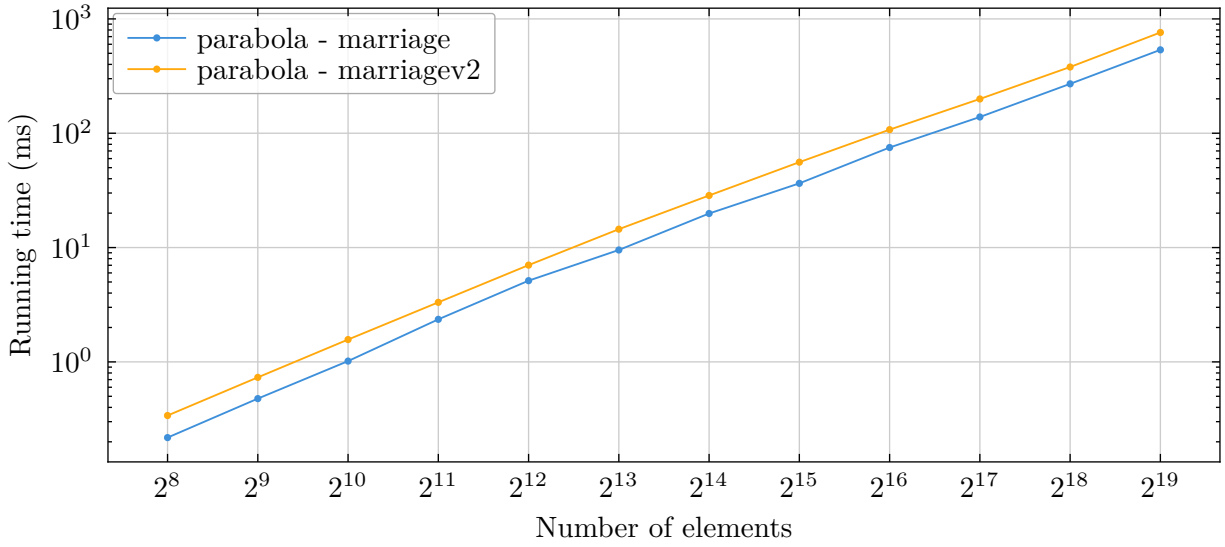
Figure 5: Flame graph for Marriage Before Conquest algorithm

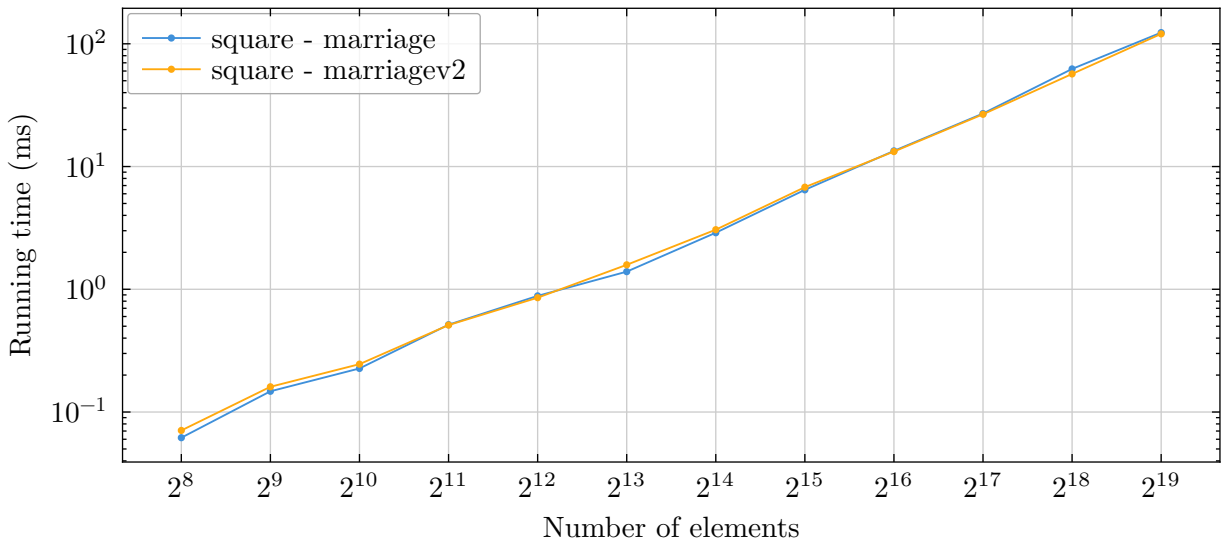
The results reported show that the `std::shuffle` function consumes a significant portion of the total execution time, indicating that the randomization step is a notable factor in the algorithm's performance. Thus, we changed the PRNG to a faster one, to reduce the overhead introduced by the shuffling process.

2.3.5 Second version

We also implemented a second version of the Marriage Before Conquest algorithm, add an extra pruning step before solving the LP subproblem: we find the point p_l with the smallest x-coordinate (if there are more than one, take the one with the largest y-coordinate) and the point p_r with the largest x-coordinate (if there are more than one, take the one with the largest y-coordinate), then prune all the points that lie under the line segment $\overline{p_l p_r}$





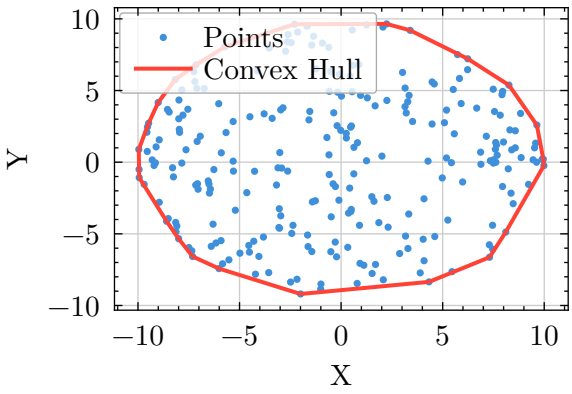


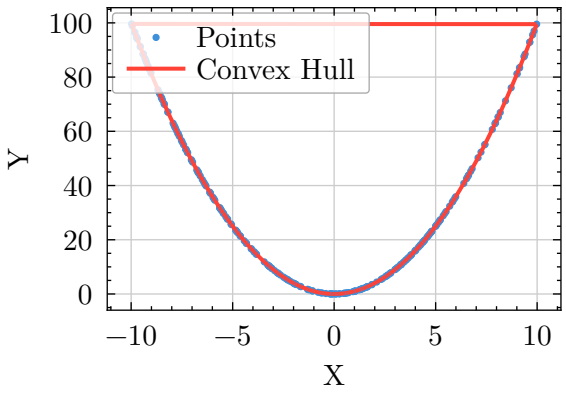
As we can see we get a small improvement in performance only in the circle distribution, while in the other distributions the performance are even worse. For the parabola distribution this is expected since the pruning step does not remove any point and in the lower hull.

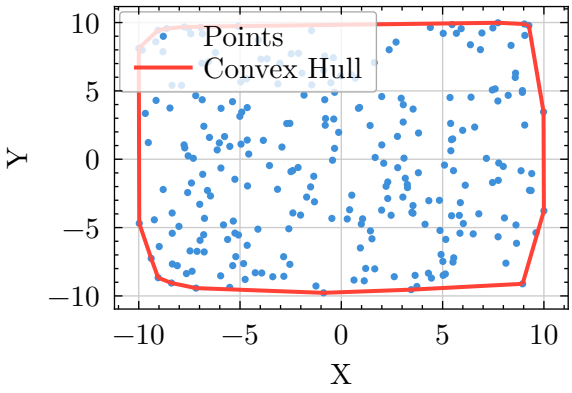
3 Conclusion

3.0.1 correctness and Visual Results

Following up are some visualizations of the convex hulls computed by one of the algorithms to give a visual proof of correctness.







The actual correctness of the algorithms has been verified at each iteration through a unit test that checks that each point is either on the hull or inside it.

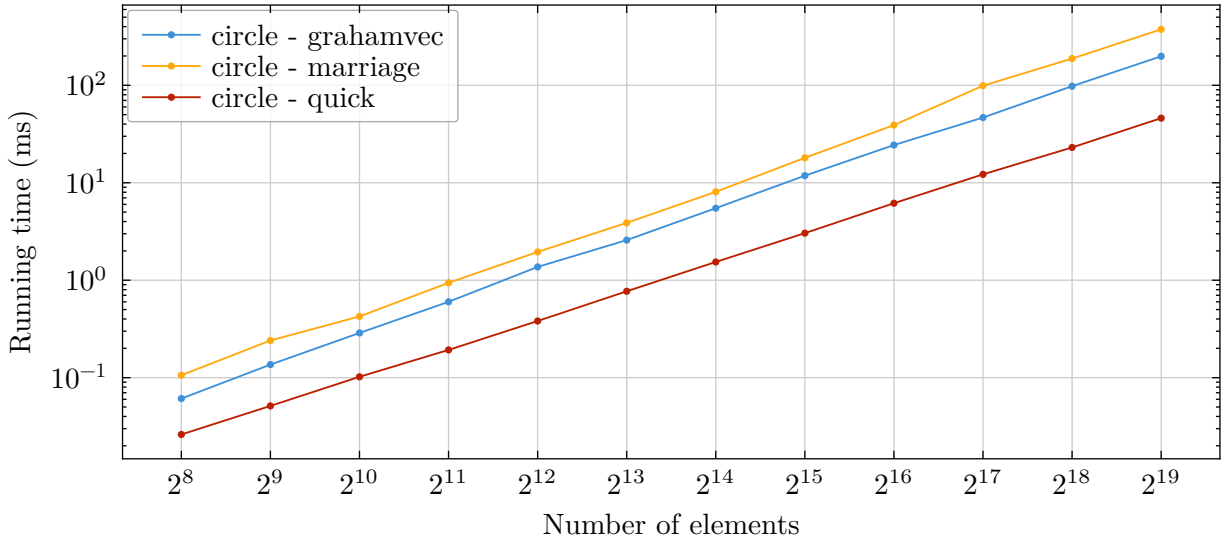
```
template <typename T>
bool is_valid_hull(const T &hull, const Points &points) {
  // check that the hull is convex
  size_t n = hull.size();
  if (n < 3) {
    return false; // A hull must have at least 3 vertices
  }

  auto it = hull.begin();
  for (size_t i = 0; i < n; ++i) {
    auto it2 = it;
    auto i1 = *it2;
    if (++it2 == hull.end()) it2 = hull.begin();
    auto i2 = *it2;
    if (++it2 == hull.end()) it2 = hull.begin();
    auto i3 = *it2;

    if (util::isLeft(i1, i2, i3)) {
      return false; // Hull is not convex
    }
  }
```

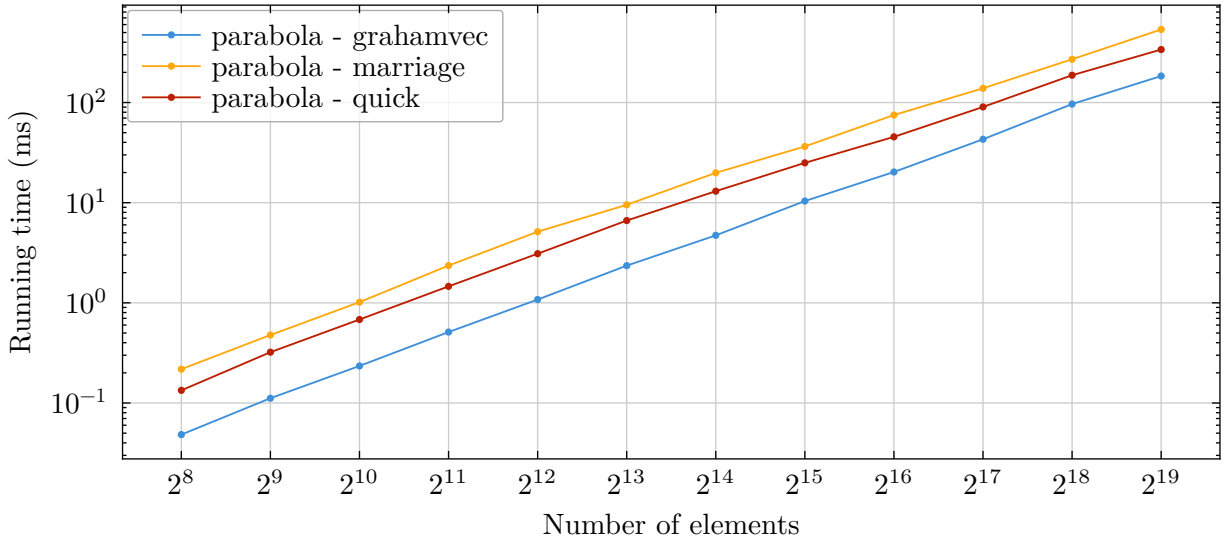
```
  // check that all points are either on the hull or inside it
  for (const auto &p : points) {
    bool on_hull = false;

    // Check if point is on hull
    for (const auto &hp : hull) {
      if (p == hp) {
        on_hull = true;
        break;
      }
    }

    if (!on_hull) {
      // Check if point is inside hull
      // A point is inside if it's on the same side of ALL edges
      if (!is_valid_inside<T>(hull, p)) {
        return false;
      }
    }
  }

  return true;
}
```

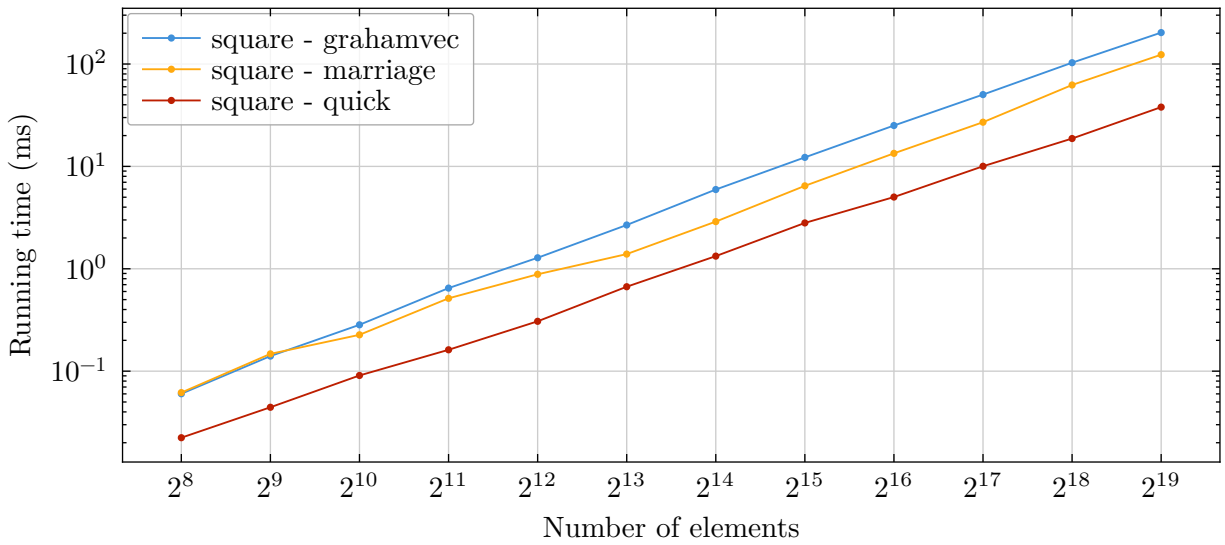
Listing 8: Unit test code snippet for hull correctness verification

3.0.2 Benchmark Results

Comparing all the algorithms implemented we can see how they perform on different distributions of points and different sizes.

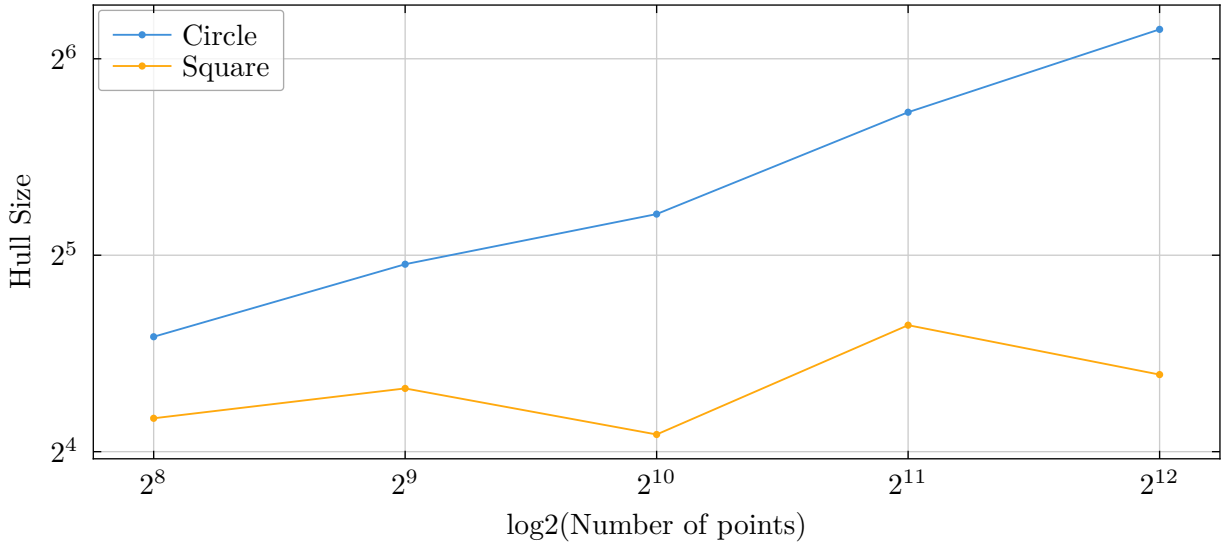


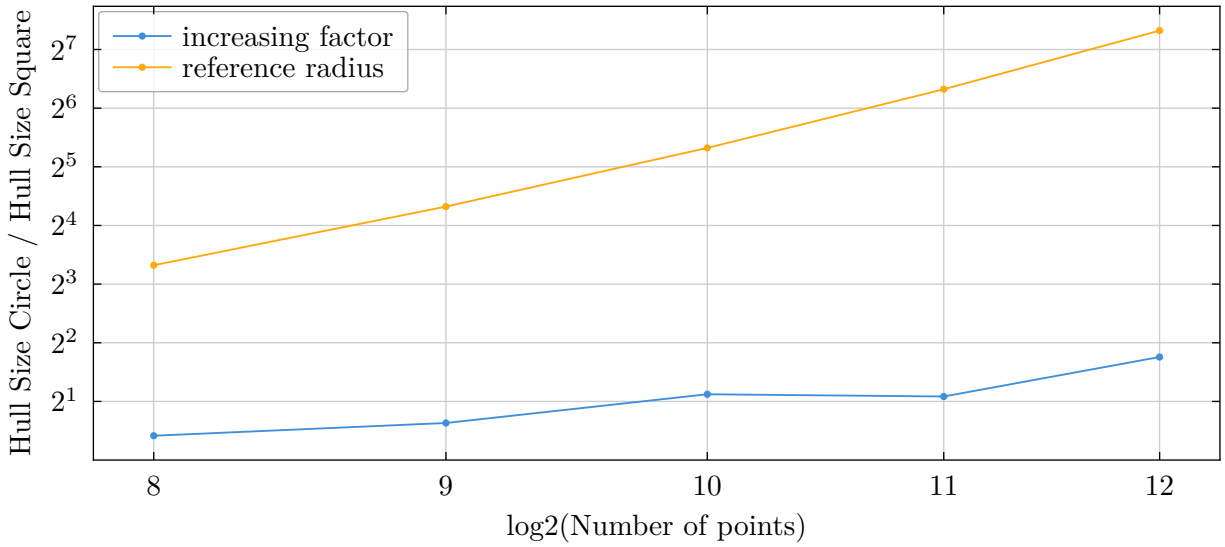




3.0.3 Comparison of Hull Sizes

Plot the difference in hull sizes generated by the three algorithms implemented.


Circle
Square
Hull Size
2^6
2^5
2^4
2^8
2^9
2^{10}
2^{11}
2^{12}
log2(Number of points)


increasing factor
reference radius
Hull Size Circle / Hull Size Square
2^7
2^6
2^5
2^4
2^3
2^2
2^1
8
9
10
11
12
log2(Number of points)

Appendix A.

Graham's Scan

```
bool point_cmp(const Point &a, const Point &b) {
  if (a.x != b.x) {
    return a.x < b.x;
  } else {
    return a.y > b.y;
  }
}

bool turns(float side, const Point &a, const Point &b, const Point &c) {
  return side * util::sidedness(Line(a, c), b) <= 0;
}

void compute_half(vector<Point> const& points, vector<Point> &half, float side) {
  half.clear();
  half.push_back(points[0]);
  half.push_back(points[1]);
  for (size_t i = 2; i < points.size(); i++) {
    while (half.size() >= 2) {
      Point const& m1 = half.back();
      Point const& m2 = *std::prev(half.end(), 2);

      if (turns(side, m2, m1, points[i]))
        half.pop_back();
      else
        break;
    }

    half.push_back(points[i]);
  }
}

vector<Point> GrahamScan::compute(const vector<Point> &points) const {
  if (points.size() <= 2) points;

  vector<Point> pts(points.begin(), points.end());
  std::sort(pts.begin(), pts.end(), point_cmp);

  T upper; compute_half(pts, upper, 1.0);
  T lower; compute_half(pts, lower, -1.0);

  lower.pop_back();
  for (int i = lower.size() - 2; i >= 0; i--) {
    upper.push_back(lower.back());
    lower.pop_back();
  }

  return upper;
}
```

Appendix B.

Graham's Scan - single pass version

```
std::deque<Point> GrahamScan::compute(std::vector<Point> const& points) const {
  if (points.size() <= 2)
    return std::deque<Point>(points.begin(), points.end());

  std::vector<Point> pts(points.begin(), points.end());
  std::sort(pts.begin(), pts.end(), point_cmp);

  std::deque<Point> res;
  res.push_back(pts[0]);
  res.push_back(pts[1]);
  res.push_front(pts[1]);
  int uh = 2, lh = 2;
  for (size_t i = 2; i < pts.size(); i++) {
    // Upper
    while (uh >= 2) {
      Point const& m1 = res.back();
      Point const& m2 = *std::prev(res.end(), 2);
      if (turns(1.0, m2, m1, pts[i])) {
        res.pop_back();
        uh -= 1;
      } else {
        break;
      }
    }
    res.push_back(pts[i]);
    uh += 1;

    // Lower
    while (lh >= 2) {
      Point const& m1 = res.front();
      Point const& m2 = * ++res.begin();
      if (turns(-1.0, m2, m1, pts[i])) {
        res.pop_front();
        lh -= 1;
      } else {
        break;
      }
    }
    res.push_front(pts[i]);
    lh += 1;
  }

  res.pop_back();
  std::rotate(res.begin(), res.begin()+lh-1, res.end());
  return res;
}
```

Appendix C.

QuickHull Algorithm

```
Points QuickHull::compute(const Points &points) const {
  Points upper_points = Points();
  Points lower_points = Points();
  Points hull = Points();

  Point q1upper, q2upper;
  Point q1lower, q2lower;

  auto [q1, q2] = util::findExtremePointsCases(points);
  if (q1.first.y == q1.second.y) {
    q1upper = q1lower = q1.first;
  } else {
    q1upper = q1.second;
    q1lower = q1.first;
  }

  if (q2.first.x == q2.second.x) {
    q2upper = q2lower = q2.first;
  } else {
    q2upper = q2.second;
    q2lower = q2.first;
  }

  hull.push_back(q1upper);

  for (const auto &p : points) {
    if (util::isLeft(q1upper, q2upper, p)) {
      upper_points.push_back(p);
    }
    if (util::isLeft(q2lower, q1lower, p)) {
      lower_points.push_back(p);
    }
  }

  QuickHull::findHullRecursive(q1upper, q2upper, upper_points, hull);
  if (hull.empty() || !(hull.back() == q2upper))
    hull.push_back(q2upper);

  if (hull.empty() || !(hull.back() == q2lower))
    hull.push_back(q2lower);

  QuickHull::findHullRecursive(q2lower, q1lower, lower_points, hull);

  if (hull.empty() || !(hull.back() == q1lower || hull.front() == q1lower))
    hull.push_back(q1lower);

  return hull;
}
```

Appendix D.

QuickHull Recursive Function

```
void QuickHull::findHullRecursive(const Point &p1, const Point &p2,
                                  const Points &points, Points &hull) const {
  /* No more points left */
  if (points.empty()) {
    return;
  }
  if (points.size() == 1) {
    hull.push_back(points[0]);
    return;
  }

  /* 1. Find the point q on one side of s that has the largest distance to s. */
  double maxDistance = -1.0;
  Point q;
  for (const auto &p : points) {
    double distance = util::partial_distance(Line(p1, p2), p);
    if (distance > maxDistance) {
      maxDistance = distance;
      q = p;
    }
  }

  /* 2. Add q to the convex hull */
  // NOTE: This is done after the recursive calls to maintain the correct order

  /* 3. Partition the remaining points into two subsets Pℓ and Pr */
  Points leftSet = Points();
  Points rightSet = Points();
  for (const auto &p : points) {
    if (util::isLeft(p1, q, p)) {
      leftSet.push_back(p);
    } else if (util::isLeft(q, p2, p)) {
      rightSet.push_back(p);
    }
  }

  /* 4. Recurse on the two subsets */
  // if bottom hull i recurr on the right side first

  findHullRecursive(p1, q, leftSet, hull);
  hull.push_back(q);
  findHullRecursive(q, p2, rightSet, hull);
}
```

Appendix E.

Marriage Before Conquest Algorithm

```
Points MarriageBeforeConquest::compute(const Points &points) const {

  if (points.size() <= 2) {
    return points;
  }

  Points hull = Points();

  std::random_device rd;
  std::default_random_engine rng(rd());

  std::vector<Point> shuffledPoints = points;
  std::shuffle(shuffledPoints.begin(), shuffledPoints.end(), rng);

  MBCUpperRecursive(shuffledPoints, hull);

  MBCLowerRecursive(shuffledPoints, hull);
  if (hull.front() == hull.back()) {
    hull.pop_back(); // remove last point to avoid duplication of leftmost point
  }
  return hull;
}
```

Appendix F.

Marriage Before Conquest Recursive Function

```
void MarriageBeforeConquest::MBCUpperRecursive(const Points &points, Points &hull)
const {
  /* If points.size() < 3, add them to the hull */
  if (points.empty()) {
    return;
  } else if (points.size() == 1) {
    hull.push_back(points[0]);
    return;
  } else if (points.size() == 2) {
    // add first the leftmost point
    if (points[0].x < points[1].x) {
      hull.push_back(points[0]);
      hull.push_back(points[1]);
    } else if (points[0].x > points[1].x) {
      hull.push_back(points[1]);
      hull.push_back(points[0]);
    } else {
      // same x, add the lower one first
```

```
      if (points[0].y < points[1].y) {
        hull.push_back(points[1]);
      } else {
        hull.push_back(points[0]);
      }
    }
    return;
  }

  Line bridge = findUpperBridge(points);

  if (bridge.p1 == bridge.p2) {
    hull.push_back(bridge.p1);
    return;
  }

  Points leftSet;
  Points rightSet;

  for (const auto &p : points) {
    if (p.x <= bridge.p1.x) {
      leftSet.push_back(p);
    } else if (p.x >= bridge.p2.x) {
      rightSet.push_back(p);
    }
  }

  MBCUpperRecursive(leftSet, hull);
  MBCUpperRecursive(rightSet, hull);
}
```

Appendix G.

Marriage Before Conquest Find Upper Bridge Function

```
Line MarriageBeforeConquest::findUpperBridge(const Points &points) const {

  Point p1, p2;
  p1 = points[0];
  Line bridge = {p1, p1};
  Point maxY = points[0];

  for (size_t i = 1; i < points.size(); ++i) {
    p2 = points[i];
    if (p2.y > maxY.y) {
      maxY = p2;
    }
    if (p1.x != p2.x) {
      if (p1.x < p2.x) {
        bridge = {p1, p2};
      } else {
```

```
        bridge = {p2, p1};
      }
      break;
    }
  }

  if (bridge.p1.x == bridge.p2.x) {

    bridge = {maxY, maxY};
    return bridge;
  }

  float midX = (bridge.p1.x + bridge.p2.x) / 2.0f;

  for (size_t i = 0; i < points.size(); ++i) {
    const auto &p = points[i];
    if (util::isLeft(bridge, p)) {

      if (p.x < midX) {
        bridge.p1 = p;
        for (size_t j = 0; j < i; ++j) {
          const auto &q = points[j];
          if (q.x >= midX) {
            if (util::isLeft(bridge, q)) {
              bridge.p2 = q;
            }
          }
        }
      } else {
        bridge.p2 = p;
        for (size_t j = 0; j < i; ++j) {
          const auto &q = points[j];
          if (q.x <= midX) {
            if (util::isLeft(bridge, q)) {
              bridge.p1 = q;
            }
          }
        }
      }
    }
  }

  return bridge;
}
```