Algorithm QUICKERHULL in code

Jyrki Katajainen[1,2]

[1] *Department of Computer Science, University of Copenhagen, Universitetsparken 5, 2100 Copenhagen East, Denmark*
[2] *Jyrki Katajainen and Company, 3390 Hundested, Denmark;*
jyrki@di.ku.dk

Abstract. This report together with an accompanying `tar` ball contains the programs discussed and benchmarked in the paper "Certification of QUICKERHULL: An algorithm for finding convex hulls in the plane". With this source code it should be possible for others to reproduce the reported experimental results.

Keywords. Computational geometry, convex hull, algorithm, QUICKHULL, QUICKERHULL, implementation, robustness, performance

CPH STL Report 2019-1, March 2019.

Contents

Abstract . . . 1
1 Introduction . . . 4
2 Interface . . . 4
3 Concepts . . . 5
3.1 Point . . . 5
Requirements . . . 5
Possible implementation . . . 5
3.2 Iterator . . . 6
Requirements . . . 6
Current implementation . . . 6
3.3 Range . . . 7
Requirements . . . 7
Additional library support . . . 7
Current implementation . . . 7
3.4 Polygon . . . 7
Requirements . . . 8
Possible implementation . . . 8
4 Common subroutines . . . 8
4.1 `std::iter_swap` . . . 8
Effect . . . 8
Parameters . . . 8
Return value . . . 8
Possible implementation . . . 8
4.2 `baseline::parallel_iter_swap` . . . 8
Effect . . . 9
Parameters . . . 9
Return value . . . 9
Current implementation . . . 9
4.3 `baseline::find_poles` . . . 9
Effect . . . 10
Parameters . . . 10
Return value . . . 10
Current implementation . . . 10
4.4 `baseline::swap_ranges` . . . 10
Effect . . . 10
Parameters . . . 10
Return value . . . 11
Current implementation . . . 11
4.5 `std::partition` . . . 11

Effect . . . 11
Parameters . . . 12
Return value . . . 12
Possible implementation . . . 12
4.6 `baseline::partition_above_below` . . . 12
Effect . . . 12
Parameters . . . 12
Return value . . . 13
Naive implementation . . . 13
4.7 `opt2::eliminate` . . . 13
Effect . . . 13
Parameters . . . 13
Return value . . . 13
Naive implementation . . . 14
5 Highlights of the optimizations . . . 14
6 Use of the `makefile` . . . 15
7 CPU measurements . . . 15
8 Conclusion . . . 17
References . . . 17
A Source code . . . 18
A.1 Copyright notice . . . 18
A.2 Release date . . . 18
B Different variants of QUICKHULL . . . 19
B.1 `baseline.h++` . . . 19
B.2 `move_opt.h++` . . . 23
B.3 `opt1.h++` . . . 26
B.4 `opt2.h++` . . . 29
B.5 `opt3.h++` . . . 34
B.6 `opt4.h++` . . . 41
C Micro-benchmarks . . . 46
C.1 `partition_baseline.h++` . . . 46
C.2 `partition_opt1.h++` . . . 46
D Helpers . . . 48
D.1 `point.h++` . . . 48
D.2 `validation.h++` . . . 60
E Drivers . . . 68
E.1 `check-driver.c++` . . . 68
E.2 `test-driver.c++` . . . 68
E.3 `driver.c++` . . . 78
F Makefile . . . 88
F.1 `makefile` . . . 88

1. Introduction

Assume that a multiset S of n points in the Euclidean plane $\mathbb{Z}^2$ is given. Each point p is specified by its Cartesian coordinates $(p.x, p.y)$, both being integers. The *convex hull* $\mathcal{H}(S)$ of S is the smallest convex set enclosing all the points of S. In the convex-hull problem, the goal is to find the smallest description of the boundary of the convex hull, i.e. the output is a convex polygon, the vertices of which—so-called *extreme points*—are from S.

In a previous report [8]—that we call a *catalogue of convex-hull programs*—we released the source code of several convex-hull algorithms: PLANE-SWEEP [3], TORCH [10], DIVIDE & CONQUER [15], QUICKHULL [6], POLES-FIRST [1], THROW-AWAY [5], INTROHULL [7], and BUCKETING [9]. The underlying implementations are discussed in depth in two of our earlier papers [7, 9].

In the catalogue we also challenged the readers to come up with an algorithm that is more efficient than any of the algorithms presented. The QUICKERHULL algorithm by Hoang and Linh [11] is the first such challenger. In the abstract of their paper they concluded that "the modifications reduce the computation time of the original QUICKHULL algorithm by a factor of three on average". In our certification [13], we could verify that QUICKERHULL is faster than QUICKHULL.

This package is mainly produced for review purposes. The reason is that the code relies on the CPH-STL integers, which have not yet been made publicly available. If you want to use the code, you have to install some other multiple-precision arithmetic package—until the CPH-STL integers have been released.

2. Interface

All the algorithms in this package rearrange the points in the input sequence in place. In addition to this side effect, the algorithms return an iterator to the first interior point. Thus, for two iterators `i` and `k`, if the range $[\![\texttt{i}\,..\,\texttt{k}]\!)$ specifies the input and `j` is the returned output, the range $[\![\texttt{i}\,..\,\texttt{j}]\!)$ contains the extreme points in circular order (clockwise or counterclockwise starting from any extreme point) and the range $[\![\texttt{j}\,..\,\texttt{k}]\!)$ contains the interior points eliminated during the computation (in arbitrary order).

Each algorithm is implemented in its own namespace which provides the function `solve` that can be used to compute the convex hull for a given sequence, and the function `check` that can be used to verify the correctness of the corresponding solver. Both of these functions have two overloaded versions: for the first version the input sequence is specified by two iterators, as for many other generic algorithms in the C++ standard library, and for the second version the input sequence is specified as an iterator range. The use of iterator ranges is experimental at the moment. The interfaces of these functions are as follows:

```
template<typename I>
requires cphmpl::is_iterator<I>
I solve(I first, I past);

template<typename R>
requires cphmpl::specifies_range<R>
cphmpl::iterator<R> solve(R& sequence);

template<typename I>
requires cphmpl::is_iterator<I>
bool check(I first, I past);

template<typename R>
requires cphmpl::specifies_range<R>
bool check(R const& sequence);
```

That is, the interface of an in-place solver is similar to that of the generic function `std::partition` in the C++ standard library.

3. Concepts

Many of our initial programming errors were due to the fact that we were not accurate enough when defining the interfaces of the functions. We found that with conceptual design it was easier to get the programs correct. In this section we define the few concepts that we have used. The geometric concepts (`is_point`, `is_polygon`, etc.) are defined in the file `point.h++` and the other concepts (`is_iterator`, `specifies_range`, etc.) are defined in the file `cphmpl/functions.h++`, which is part of the CPH MPL [12].

3.1 Point

Defined in file "point.h++"

```
template<typename P>
concept bool is_point;
```

Requirements

A point must have a type member `coordinate`, the attributes `x` and `y`, the function members **operator**== and **operator**≠, and it must be default constructible and it must support parameterized construction.

Possible implementation

```
template<typename P>
concept bool is_point =
requires (P p) {
  typename P::coordinate;
  {p.x} → typename P::coordinate;
```

```
  {p.y} → typename P::coordinate;
  {p == q} → bool;
  {p != q} → bool;
  P();
  P(p.x, p.y);
};
```

3.2 Iterator

Defined in file "cphmpl/functions.h++"

```
template<typename I>
concept bool is_iterator;
```

Requirements

An iterator specifies a position in a sequence of elements. In our code, we normally assume that the underlying sequences provide random access to their elements. In many cases a weaker form of iterators would be sufficient, but that had required more careful programming.

Current implementation

```
namespace cphmpl::helper {

  template<typename T>
  class is_iterator {
  private:

    static char test(...);

    template<typename U,
      typename = typename std::iterator_traits<U>::difference_type,
      typename = typename std::iterator_traits<U>::pointer,
      typename = typename std::iterator_traits<U>::reference,
      typename = typename std::iterator_traits<U>::value_type,
      typename = typename
    ↪ std::iterator_traits<U>::iterator_category>
    static long test(U&&);

  public:

    static constexpr bool value =
    ↪ std::is_same_v<decltype(test(std::declval<T>())), long>;
  };
}

namespace cphmpl {

  template<typename T>
  concept bool is_iterator = helper::is_iterator<T>::value;
```

```
}
```

3.3 Range

Defined in file "cphmpl/functions.h++"

```
template<typename R>
concept bool specifies_range;
```

Requirements

A *range* specifies a sequence of elements. To get access to its elements, it should support the functions `std::begin`, `std::cbegin`, `std::end`, `std::cend`, `std::size`, and `std::empty`. With this abstraction, the programs are independent of the representation of the data. Implicitly, we assume that a sequence is stored in a `std::array`, in a `std::vector`, in a C array, or in any other container—or part of it—that supports (random-access) iterators. With more careful programming, for many functions, weaker form of iterators would be sufficient.

Additional library support

In file "cphstl/ranges.h++", there is a class that can be used to create a range for any pair of iterators referring to the same sequence. For more advanced support of ranges, see the headers `<experimental/ranges/*>` [https://en.cppreference.com].

Current implementation

```
namespace cphmpl {

  template<typename R>
  concept bool specifies_range =
  requires(R&& t) {
    std::begin(t);
    std::end(t);
    std::cbegin(t);
    std::cend(t);
    std::size(t);
    std::empty(t);
  };
}
```

3.4 Polygon

Defined in file "point.h++"

```
template<typename G>
concept bool is_polygon;
```

Requirements

A polygon is a range of points. In a normalized representation, the lexicographical smallest point comes first and the other points are given in clockwise order thereafter.

Possible implementation

```
template<typename G>
concept bool is_polygon =
  cphmpl::specifies_range<G> and is_point<cphmpl::value<G>>;
```

4. Common subroutines

In this section we describe the common subroutines used in most variants of QUICKHULL discussed in this package. After describing these commonalities it should easier to see the differences in the optimizations.

4.1 std::iter_swap

Defined in file <algorithm>, see [https://en.cppreference.com]

```
template<typename ForwardIt1, typename ForwardIt2>
constexpr void iter_swap(ForwardIt1 a, ForwardIt2 b);
```

Effect

Move the element at position `a` to position `b`, and vice versa.

Parameters

a, b: iterators to the elements to be swapped

Return value

(none)

Possible implementation

```
template<typename I1, typename I2>
requires cphmpl::is_iterator<I1> and cphmpl::is_iterator<I2>
constexpr void iter_swap(I1 a, I2 b) {
  using std::swap;
  swap(*a, *b);
}
```

4.2 baseline::parallel_iter_swap

Defined in file "baseline.h++"

```
template<typename Pair1, typename Pair2>
constexpr void parallel_iter_swap(Pair1 a, Pair2 b);
```

Effect

Given two pairs of iterators $a = (a_1, a_2)$ and $b = (b_1, b_2)$, carry out the operations $\texttt{iter\_swap}(a_1, b_1)$ and $\texttt{iter\_swap}(a_2, b_2)$ in parallel. In particular, handle the situation where some of these iterators are the same. Precondition: $a_1 \neq a_2$ and $b_1 \neq b_2$.

Parameters

a, b: Two pairs of iterators to the elements to be swapped

Return value

(none)

Current implementation

```
template<typename T, typename I = typename std::tuple_element<0,
    ↪ T>::type>
requires cphmpl::is_pair<T, I, I> and cphmpl::is_iterator<I>
constexpr void parallel_iter_swap(T a, T b) {
  I a1;
  I a2;
  std::tie(a1, a2) = a;
  I b1;
  I b2;
  std::tie(b1, b2) = b;
  assert(a1 ≠ a2 and b1 ≠ b2);
  std::iter_swap(a1, b1);
  if (a1 == b2) {
    std::iter_swap(a2, b1);
  }
  else {
    std::iter_swap(a2, b2);
  }
}
```

4.3 baseline::find_poles

Defined in file "baseline.h++"

```
template<typename R, typename I = cphmpl::const_iterator<R>>
std::pair<I, I> find_poles(R const& range);
```

Effect

Find the lexicographically smallest and largest points in the given `range` of points.

Parameters

`range:` the range of points under consideration

Return value

A pair of iterators pointing to the west pole and east pole, respectively.

Current implementation

```
template<typename R, typename I = cphmpl::const_iterator<R>>
requires cphmpl::specifies_range<R> and is_point<cphmpl::value<R>>
constexpr std::pair<I, I> find_poles(R const& range) {
  I first = std::cbegin(range);
  I past = std::cend(range);
  using P = cphmpl::value<I>;
  std::pair<I, I> pair = std::minmax_element(first, past,
    [](P const& a, P const& b) -> bool {
      return (a.x < b.x) or (a.x == b.x and a.y < b.y);
    });
  return pair;
}
```

4.4 `baseline::swap_ranges`

Defined in file "baseline.h++"

```
template<typename R, typename I>
constexpr I swap_ranges(R& range, I target);
```

Effect

Exchanges the elements between the given `range` and another range starting at `target`. The relative order of the elements in the first range is preserved, but the same is not necessary true for the other range. (`std::swap_ranges` preserves the order of the elements in both ranges!)

Parameters

`range:` the first range of elements to be swapped

`target:` beginning of the second range of elements to be swapped

Return value

Iterator to the element past the last element exchanged in the range beginning with `target`.

Current implementation

```
template<typename R, typename I>
requires cphmpl::specifies_range<R> and cphmpl::is_iterator<I>
constexpr I swap_ranges(R& range, I target) {
  using J = cphmpl::iterator<R>;
  J source = std::begin(range);
  J past = std::end(range);
  if (source == past) {
    return target;
  }
  if (source == target) {
    return past;
  }
  using V = cphmpl::value<I>;
  I hole = target;
  V const t = std::move(*target);
  I const last = std::prev(past);
  while (true) {
    *hole = std::move(*source);
    ++hole;
    if (source == last) {
      break;
    }
    *source = std::move(*hole);
    ++source;
  }
  *source = std::move(t);
  return hole;
}
```

4.5 std::partition

Defined in file <algorithm>, see [https://en.cppreference.com].

```
template<typename ForwardIt, typename UnaryPredicate>
constexpr ForwardIt partition(ForwardIt first, ForwardIt past,
    ↪ UnaryPredicate p);
```

Effect

Reorders the elements in the range ⟦`first`, `past`⟩ in such a way that all elements for which the predicate `p` returns **true** precede the elements for which predicate `p` returns **false**. Relative order of the elements is not preserved.

Parameters

first, past: The half-open range of elements to be reordered

p: Unary predicate which returns **true** if the element should be ordered before other elements.

Return value

Iterator to the first element of the second group.

Possible implementation

```
template<typename I, typename U>
requires cphmpl::is_forward_iterator<I> and
    ↪ cphmpl::is_unary_predicate<U>
constexpr I partition(I first, I past, U predicate) {
  first = std::find_if_not(first, past, p);
  if (first == last) {
    return first;
  }
  for (I i = std::next(first); i != past; ++i) {
    if (predicate(*i)) {
      std::iter_swap(i, first);
      ++first;
    }
  }
  return first;
}
```

4.6 baseline::partition_above_below

Defined in file "baseline.h++"; the optimized versions improve upon this.

```
template<typename R, typename I>
I partition_above_below(R& range, I pole, I antipole);
```

Effect

Reorders the points in the given `range` in such a way that all points that are above the line—specified by the points pointed to by `pole` and `antipole`—precede the points that are below or on that line. Relative order of the points is not retained in any of the groups. Precondition: the iterators must refer to outside the `range`.

Parameters

range: the range of points under consideration

pole, antipole: these iterators pointing to points specify the partitioning line between the two groups.

Return value

Iterator to the first point of the second group of points below the line.

Naive implementation

```
template<typename R, typename I>
requires cphmpl::specifies_range<R> and is_point<cphmpl::value<R>>
    ↪ and cphmpl::is_iterator<I> and is_point<cphmpl::value<I>>
I partition_above_below(R& range, I pole, I antipole) {
  I first = std::begin(range);
  I past = std::end(range);
  using P = cphmpl::value<I>;
  I mid = std::partition(first, past,
    [&](P const& r) → bool {
      return left_turn(*pole, *antipole, r);
    });
  return mid;
}
```

4.7 opt2::eliminate

Defined in file "opt2.h++"; the other versions handle this elimination in their own ways.

```
template<typename R, typename I, typename C>
std::tuple<I, I, I, I> eliminate(R& range, I p, I q, I r, C less);
```

Effect

Reorders the points in the given `range` into three groups: (1) All points that are above $\overline{\texttt{*p *q}}$ come first (region R_1). (2) All points that are above $\overline{\texttt{*q *r}}$ come next (region R_2). (3) All points that are inside the triangle determined by the points `*p`, `*q`, and `*r` come last. Relative order of the points in neither of the groups is preserved. During the partitioning process the farthest point from $\overline{\texttt{*p *q}}$ in region R_1 and that from $\overline{\texttt{*q *r}}$ in region R_2 are localized as well. Precondition: All points in the `range` are on the left of $\overrightarrow{\texttt{*p *r}}$.

Parameters

`range`: the range of points under consideration
`p`, `q`, `r`: positions of points outside the `range` specifying a triangle
`less`: a comparator used in tie breaking when the distance from a line segment is the same for two or more points.

Return value

The four-tuple of iterators specifying past the end of the region R_1, past the end of the region R_2, the farthest point above $\overline{\texttt{*p *q}}$ inside region R_1, and

the farthest point above $\overline{\texttt{*q *r}}$ inside region R_2.

Naive implementation

```
template<typename R, typename I, typename C, typename T = typename
    ↪ cphmpl::value<I>::coordinate>
requires
/* 1 */ cphmpl::specifies_range<R> and
/* 2 */ cphmpl::is_iterator<I> and
/* 3 */ is_point<cphmpl::value<I>> and
/* 4 */ std::is_same_v<I, cphmpl::iterator<R>> and
/* 5 */ cphmpl::is_binary_predicate<C, T, T>
std::tuple<I, I, I, I> eliminate(R& range, I p, I q, I r, C less) {
  I m;
  I q1;
  std::tie(m, q1) = partition_two_way(range, p, q, less);
  auto rests = cphstl::range(m, std::end(range));
  I t;
  I q2;
  std::tie(t, q2) = partition_two_way(rests, q, r, less);
  return std::make_tuple(m, t, q1, q2);
}
```

5. Highlights of the optimizations

In this package we release the source code of the programs used in the certification [13]. The package contains the following six versions of QUICKHULL:

baseline: This version was taken from the catalogue [7] (release 1.31). To unify the style with the other programs in this package, we have made some stylistic changes to the original.

move opt: This version is a modification of the baseline that is more efficient with respect to the number of coordinate moves made on an average. But if the output is large, it will actually make a bit more moves than BASELINE.

opt 1: This version implements the first optimization of Hoang and Linh [11]: the orientation predicates are partially evaluated outside the partitioning loops.

opt 2: This version implements the second optimization of Hoang and Linh [11]: the recursive calls assume that the farthest point from the partitioning line is already available. This removes some redundant orientation tests. This was a known optimization; it was already used by Eddy in his original Fortran implementation of QUICKHULL and later by Allison and Noga [2].

opt 3: This version implements the third optimization of Hoang and Linh [11]: Points inside or on a triangle are eliminated more efficiently so that only one orientation test is performed per point in the considered range.

opt 4: This version is almost identical to the THROW-AWAY elimination described in the catalogue [7]. Now OPT 3 is used to handle the points remaining after the elimination; not PLANE-SWEEP as in one of the early implementations [4] or in the catalogue.

In general, the validation and test frameworks are the same as those used in the catalogue [7, 8].

6. Use of the `makefile`

You can use `make` to redo the experiments done in the certification [13]. For each version, the `makefile` provides the following facilities. For the sake of concreteness, we fix the version to be BASELINE.

`baseline.check`**:** Run a unit-test to check that the code compiles.

`baseline.test`**:** Run the program through all our test cases.

`baseline.square`**:** Run the CPU benchmark for the square data set.

`baseline.parabola`**:** Run the CPU benchmark for a multiset of random points on a parabola.

`baseline.turn`**:** Measure how many times an orientation test is called.

`baseline.comp`**:** Measure how many coordinate comparisons are performed.

`baseline.move`**:** Measure how many coordinate moves are performed.

`baseline.arithmetic`**:** Measure how many different operations (constructions, comparisons, additions, subtractions, and multiplications) are carried out for wide integers. (These measurements are done in the classes `cphstl`::$\mathbb{Z}$<`b`>, for different widths `b`.)

The same `makefile` can also handle the micro-benchmarks since these are packaged in the same way as the convex-hull algorithms. Each micro-benchmark is defined its own namespace—the name of this namespace is the same as the name of the file without the trailing suffix `h++` —and the driver will run the function `solve` in this namespace.

7. CPU measurements

In the certification [13] we did not publish any CPU timings; we do it here to satisfy the curiosity of a skeptical reader. The test environment was my personal computer. The details of the hardware and software in use are given in Table 1.

These experiments were run for the same data sets as those used in the certification [13]:

Square data set: A multiset of points in a square; the coordinates of the points were random integers uniformly distributed over the interval $[\![-2^{31}\,..\,2^{31}[\![$ (i.e. random **int**s).

Killer data set: A multiset of points on a parabola; the x-coordinates of the points were integers selected randomly from the closed interval $[\![-2^{15}\,..\,2^{15}]\!]$ and the y-coordinates were computed from the formula $y = x^2$.

Table 1. Hardware and software in use.

Processor. Intel® Core™ i7-6600U CPU @ 2.6 GHz (turbo-boost up to 3.6 GHz) × 4
Word size. 64 bits
First-level data cache. 8-way set-associative, 64 sets, 4 × 32 KB
First-level instruction cache. 8-way set-associative, 64 sets, 4 × 32 KB
Second-level cache. 4-way set-associative, 1 024 sets, 256 KB
Third-level cache. 16-way set-associative, 4 096 sets, 4.096 MB
Main memory. 8.038 GB
Operating system. Ubuntu 18.04.1 LTS
Kernel. Linux 4.15.0-43-generic
Compiler. g++ 8.2.0
Compiler options. `-O3 -std=c++2a -Wall -Wextra -x c++ -fconcepts -DNDEBUG`

Table 2. The running time of the different versions of QUICKHULL [in nanoseconds per point] for the square data sets.

n	BASELINE	MOVE OPT	OPT 1	OPT 2	OPT 3	OPT 4	THROW-AWAY [9]
2^{10}	78.92	78.83	71.58	65.62	42.31	45.53	20.40
2^{15}	74.96	74.41	68.75	63.60	39.41	40.04	16.49
2^{20}	77.17	74.06	67.79	63.70	39.24	39.47	15.82
2^{25}	76.24	74.79	68.91	64.09	41.02	38.05	15.38
2^{30}	157.2	138.5	131.2	130.6	96.22	71.61	46.79

The obtained results are given in Table 2 (square data sets) and Table 3 (killer data sets). Looking at these results and those reported in the certification [13], we conclude that some further tuning is still possible. At least we have observed the following possibilities:

- In OPT 4 the correct edge on the approximate convex hull can be localized using binary search instead of linear search. This will save some coordinate comparisons.
- By using a floating-point filter, both BASELINE and OPT 4 could be accelerated by about a factor of two. For the other versions this did not work, since currently we can only accelerate the `left_turn` predicate, but not the `signed_area` calculations (for more information on floating-point acceleration, see [7, 9]).
- For the CPH STL integers, multiplication turned out to be slower than that for the extension `__int128` supported by the `g++` compiler (for more details on the efficiency of the CPH STL integers, see [14]). So it is probable that the arithmetic operations can be sped up further.

Table 3. The running time of the different versions of QUICKHULL [in nanoseconds per point] for the killer data sets.

n	BASELINE	MOVE OPT	OPT 1	OPT 2	OPT 3	OPT 4	THROW-AWAY [9]
2^{10}	395	389	370	299	194	231	60.19
2^{15}	593	585	563	452	284	323	73.18
2^{20}	643	633	577	491	296	333	90.63
2^{25}	643	630	583	492	294	331	91.83

8. Conclusion

To conclude, QUICKERHULL (OPT 4) is faster than QUICKHULL (BASELINE). With floating-point acceleration, we could speed up THROW-AWAY even more than the factor of three announced by Hoang and Linh [11]. However, the truth is more complicated than the phrase "a factor of three faster" suggests.

References

[1] S. G. Akl and G. T. Toussaint, A fast convex hull algorithm, *Inf. Process. Lett.* **7**, 5 (1978), 219–222. https://doi.org/10.1016/0020-0190(78)90003-0

[2] D. C. S. Allison and M. T. Noga, Some performance tests of convex hull algorithms, *BIT* **24**, 1 (1984), 2–13. https://doi.org/10.1007/BF01934510

[3] A. M. Andrew, Another efficient algorithm for convex hulls in two dimensions, *Inf. Process. Lett.* **9**, 5 (1979), 216–219. https://doi.org/10.1016/0020-0190(79)90072-3

[4] B. K. Bhattacharya and G. T. Toussaint, Time- and storage-efficient implementation of an optimal planar convex hull algorithm, *Image Vision Comput.* **1**, 3 (1983), 140–144. https://doi.org/10.1016/0262-8856(83)90065-3

[5] L. Devroye and G. T. Toussaint, A note on linear expected time algorithms for finding convex hulls, *Computing* **26**, 4 (1981), 361–366. https://doi.org/10.1007/BF02237955

[6] W. F. Eddy, A new convex hull algorithm for planar sets, *ACM Trans. Math. Software* **3**, 4 (1977), 398–403. https://doi.org/10.1145/355759.355766

[7] A. N. Gamby and J. Katajainen, Convex-hull algorithms: Implementation, testing, and experimentation, *Algorithms* **11**, 12 (2018). https://doi.org/10.3390/a11120195

[8] A. N. Gamby and J. Katajainen, Convex-hull algorithms in C++, CPH STL report **2018-1**, Department of Computer Science, University of Copenhagen (2018–2019). http://www.diku.dk/~jyrki/Myris/GK2018S.html

[9] A. N. Gamby and J. Katajainen, *A faster convex-hull algorithm via bucketing*, Submitted for publication (2019).

[10] A. J. P. Gomes, A total order heuristic-based convex hull algorithm for points in the plane, *Comput.-Aided Design* **70** (2016), 153–160. https://doi.org/10.1016/j.cad.2015.07.013

[11] N.-D. Hoang and N. K. Linh, Quicker than Quickhull, *Vietnam J. Math.* **43**, 1 (2015), 57–70. https://doi.org/10.1007/s10013-014-0067-1

[12] J. Katajainen, Pure compile-time functions and classes in the CPH MPL, CPH STL report **2017-2**, Department of Computer Science, University of Copenhagen (2017). http://hjemmesider.diku.dk/~jyrki/Myris/Kat2017R.html

[13] J. Katajainen, *Certification of* QUICKERHULL*: An algorithm for finding convex hulls in the plane*, Submitted for publication (2019).
[14] J. Katajainen, *Hacker's multiple-precision integer-division program in close scrutiny*, Submitted for publication (2019).
[15] F. P. Preparata and S. J. Hong, Convex hulls of finite sets of points in two and three dimensions, *Commun. ACM* **20**, 2 (1977), 87–93. https://doi.org/10.1145/359423.359430

A. Source code

A.1 Copyright notice

A.2 Release date

March 10, 2019

B. Different variants of QUICKHULL

B.1 baseline.h++

```
/*
  Performance Engineering Laboratory © 2017–2019
*/

#ifndef __QUICKHULL_BASELINE__
#define __QUICKHULL_BASELINE__

#define MULTIPLE_PRECISION

#include "point.h++" // signed_area left_turn no_turn

#include <algorithm> // std::minmax_element std::partitioning ...
#include <cassert> // assert macro
#include "cphmpl/functions.h++" // cphmpl type functions
#include "cphstl/ranges.h++" // cphstl ranges
#include <cstdlib> // std::size_t
#include <functional> // std::less std::greater
#include <iterator> // std::begin std::end std::size std::empty ...
#include <type_traits> // std::is_same_v
#include <utility> // std::pair std::make_pair std::move ...
#include "validation.h++" // same_multiset convex_polygon all_inside
#include <vector> // std::vector

namespace baseline {

  template<typename T, typename I = typename
    ↪ std::tuple_element<0, T>::type>
  requires cphmpl::is_pair<T, I, I> and cphmpl::is_iterator<I>
  constexpr void parallel_iter_swap(T a, T b) {
    I a1;
    I a2;
    std::tie(a1, a2) = a;
    I b1;
    I b2;
    std::tie(b1, b2) = b;
    assert(a1 != a2 and b1 != b2);
    std::iter_swap(a1, b1);
    if (a1 == b2) {
      std::iter_swap(a2, b1);
    }
    else {
      std::iter_swap(a2, b2);
    }
  }

  template<typename R, typename I = cphmpl::const_iterator<R>>
  requires cphmpl::specifies_range<R> and
    ↪ is_point<cphmpl::value<R>>
```

```
47  constexpr std::pair<I, I> find_poles(R const& range) {
48    I first = std::cbegin(range);
49    I past = std::cend(range);
50    using P = cphmpl::value<I>;
51    std::pair<I, I> pair = std::minmax_element(first, past,
52      [](P const& a, P const& b) → bool {
53        return (a.x < b.x) or (a.x == b.x and a.y < b.y);
54      });
55    return pair;
56  }
57
58  template<typename R, typename I, typename C, typename T =
      ↪ typename cphmpl::value<I>::coordinate>
59  requires
60  /* 1 */ cphmpl::specifies_range<R> and
61  /* 2 */ is_point<cphmpl::value<R>> and
62  /* 3 */ cphmpl::is_iterator<I> and
63  /* 4 */ is_point<cphmpl::value<I>> and
64  /* 5 */ cphmpl::is_binary_predicate<C, T, T>
65  I find_furthest(R const& range, I pole, I antipole, C less) {
66    assert(not std::empty(range));
67    assert(*pole ≠ *antipole);
68    I first = std::cbegin(range);
69    I past = std::cend(range);
70    I answer = pole;
71    using Z = decltype(signed_area(*pole, *pole, *pole));
72    Z best = Z();
73    is_lexicographically_smaller<C, T> tiebreak(less);
74    for (I i = first; i ≠ past; ++i) {
75      Z Φ = signed_area(*pole, *antipole, *i);
76      if (Φ > best or (Φ == best and tiebreak(*i, *answer))) {
77        answer = i;
78        best = Φ;
79      }
80    }
81    return answer;
82  }
83
84  template<typename R, typename I>
85  requires cphmpl::specifies_range<R> and
      ↪ is_point<cphmpl::value<R>> and cphmpl::is_iterator<I>
      ↪ and is_point<cphmpl::value<I>>
86  I partition_above_below(R& range, I pole, I antipole) {
87    I first = std::begin(range);
88    I past = std::end(range);
89    using P = cphmpl::value<I>;
90    I mid = std::partition(first, past,
91      [&](P const& r) → bool {
92        return left_turn(*pole, *antipole, r);
93      });
94    return mid;
95  }
```

```
96
97  template<typename R, typename I>
98  requires cphmpl::specifies_range<R> and cphmpl::is_iterator<I>
99  constexpr I swap_ranges(R& range, I target) {
100   using J = cphmpl::iterator<R>;
101   J source = std::begin(range);
102   J past = std::end(range);
103   if (source == past) {
104     return target;
105   }
106   if (source == target) {
107     return past;
108   }
109   using V = cphmpl::value<I>;
110   I hole = target;
111   V const t = std::move(*target);
112   I const last = std::prev(past);
113   while (true) {
114     *hole = std::move(*source);
115     ++hole;
116     if (source == last) {
117       break;
118     }
119     *source = std::move(*hole);
120     ++source;
121   }
122   *source = std::move(t);
123   return hole;
124 }
125
126 template<typename I>
127 requires cphmpl::is_iterator<I>
128 void move_away(I here, I rest, I past) {
129   if (here == rest or rest == past) {
130     return;
131   }
132   if (rest - here < past - rest) {
133     auto range = cphstl::range(here, rest);
134     (void) swap_ranges(range, past - (rest - here));
135   }
136   else {
137     auto range = cphstl::range(rest, past);
138     (void) swap_ranges(range, here);
139   }
140 }
141
142 template<typename I, typename C, typename T = typename
    ↪ cphmpl::value<I>::coordinate>
143 requires
144 /* 1 */ cphmpl::is_iterator<I> and
145 /* 2 */ is_point<cphmpl::value<I>> and
146 /* 3 */ cphmpl::is_binary_predicate<C, T, T>
```

```
147   I recurse(I pole, I past, I antipole, C less) {
148     std::size_t n = std::distance(pole, past);
149     if (n == 1) {
150       return past;
151     }
152     if (n == 2) {
153       if (no_turn(*std::next(pole), *pole, *antipole)) {
154         return std::next(pole);
155       }
156       else {
157         return past;
158       }
159     }
160     auto whole = cphstl::range(std::next(pole), past);
161     I pivot = find_furthest(whole, pole, antipole, less);
162     if (pivot == pole) {
163       return std::next(pole);
164     }
165     I last = std::prev(past);
166     std::iter_swap(pivot, last); // pivot at the end
167     auto range = cphstl::range(std::next(pole), last);
168     I mid = partition_above_below(range, pole, last);
169     I eliminated = recurse(pole, mid, last, less);
170     std::iter_swap(mid, last);
171     std::iter_swap(eliminated, mid); // pivot at its final place
172     pivot = eliminated;
173     std::size_t m = past - mid;
174     ++mid;
175     ++eliminated;
176     move_away(eliminated, mid, past);
177     auto rests = cphstl::range(std::next(pivot), pivot + m);
178     I interior = partition_above_below(rests, pivot, antipole);
179     eliminated = recurse(pivot, interior, antipole, less);
180     return eliminated;
181   }
182
183   template<typename I>
184   requires cphmpl::is_iterator<I> and is_point<cphmpl::value<I>>
185   I solve(I first, I past) {
186     std::size_t n = past - first;
187     if (n < 2 or (n == 2 and *first != *(first + 1))) {
188       return past;
189     }
190     auto sequence = cphstl::range(first, past);
191     std::pair<I, I> poles = find_poles(sequence);
192     I last = std::prev(past);
193     parallel_iter_swap(std::make_pair(first, last), poles);
194     if (*first == *last) {
195       return std::next(first);
196     }
197     auto range = cphstl::range(std::next(first), last);
198     I mid = partition_above_below(range, first, last);
```

```
199       std::size_t m = past - mid;
200       using T = decltype((*first).x);
201       I eliminated = recurse(first, mid, last, std::less<T>());
202       std::iter_swap(mid, last);
203       std::iter_swap(eliminated, mid);
204       I east = eliminated;
205       ++mid;
206       ++eliminated;
207       move_away(eliminated, mid, past);
208       eliminated = recurse(east, east + m, first,
          ↪ std::greater<T>());
209       return eliminated;
210     }
211
212     template<typename R>
213     requires cphmpl::specifies_range<R> and
          ↪ is_point<cphmpl::value<R>>
214     cphmpl::iterator<R> solve(R& sequence) {
215       return solve(std::begin(sequence), std::end(sequence));
216     }
217
218     template<typename I>
219     requires cphmpl::is_iterator<I> and is_point<cphmpl::value<I>>
220     bool check(I first, I past) {
221       using P = cphmpl::value<I>;
222       using S = std::vector<P>;
223       using J = typename S::iterator;
224       S data;
225       std::size_t n = std::distance(first, past);
226       data.resize(n);
227       std::copy(first, past, data.begin());
228       J rest = solve(data.begin(), data.end());
229       bool ok = validation::same_multiset(data.begin(), data.end(),
          ↪ first, past) and validation::convex_polygon(data.begin(),
          ↪ rest) and validation::all_inside(rest, data.end(),
          ↪ data.begin(), rest);
230       return ok;
231     }
232
233     template<typename R>
234     requires cphmpl::specifies_range<R> and
          ↪ is_point<cphmpl::value<R>>
235     bool check(R const& sequence) {
236       return check(std::cbegin(sequence), std::cend(sequence));
237     }
238   }
239
240   #endif
```

B.2 move_opt.h++

```
1  /*
2    Performance Engineering Laboratory © 2017–2019
3  */
4
5  #ifndef __MOVE_OPTIMIZED_QUICKHULL__
6  #define __MOVE_OPTIMIZED_QUICKHULL__
7
8  #define MULTIPLE_PRECISION
9
10 #include "point.h++" // signed_area left_turn no_turn
11
12 #include <algorithm> // std::minmax_element std::partitioning ...
13 #include "baseline.h++" // baseline::parallel_iter_swap ...
14 #include <cassert> // assert macro
15 #include "cphmpl/functions.h++" // cphmpl type functions
16 #include "cphstl/ranges.h++" // cphstl ranges
17 #include <cstdlib> // std::size_t
18 #include <functional> // std::less std::greater
19 #include <iterator> // std::begin std::end std::size std::empty ...
20 #include <type_traits> // std::is_same_v
21 #include <utility> // std::pair std::make_pair ...
22 #include "validation.h++" // same_multiset convex_polygon all_inside
23 #include <vector> // std::vector
24
25 namespace move_opt {
26
27   template<typename I, typename C, typename T = typename
       ↪ cphmpl::value<I>::coordinate>
28   requires
29   /* 1 */ cphmpl::is_iterator<I> and
30   /* 2 */ is_point<cphmpl::value<I>> and
31   /* 3 */ cphmpl::is_binary_predicate<C, T, T>
32   I recurse(I pole, I past, I antipole, C less) {
33     std::size_t n = std::distance(pole, past);
34     if (n ≤ 1) {
35       return past;
36     }
37     if (n == 2) {
38       if (no_turn(*std::next(pole), *pole, *antipole)) {
39         return std::next(pole);
40       }
41       else {
42         return past;
43       }
44     }
45     auto whole = cphstl::range(std::next(pole), past);
46     I pivot = baseline::find_furthest(whole, pole, antipole, less);
47     if (pivot == pole) {
48       return std::next(pole);
49     }
50     I last = std::prev(past);
51     std::iter_swap(pivot, last); // pivot at the end
```

```
52      auto range = cphstl::range(std::next(pole), last);
53      I mid = baseline::partition_above_below(range, pole, last);
54      I eliminated1 = recurse(pole, mid, last, less);
55      std::iter_swap(mid, last); // pivot in the middle
56      auto rests = cphstl::range(std::next(mid), past);
57      I interior = baseline::partition_above_below(rests, mid,
        ↪ antipole);
58      I eliminated2 = recurse(mid, interior, antipole, less);
59      auto second_chain = cphstl::range(mid, eliminated2);
60      I eliminated = baseline::swap_ranges(second_chain, eliminated1);
61      return eliminated;
62    }
63
64    template<typename I>
65    requires cphmpl::is_iterator<I> and is_point<cphmpl::value<I>>
66    I solve(I first, I past) {
67      std::size_t n = std::distance(first, past);
68      if (n < 2 or (n == 2 and *first ≠ *std::next(first))) {
69        return past;
70      }
71      auto whole = cphstl::range(first, past);
72      std::pair<I, I> poles = baseline::find_poles(whole);
73      I last = std::prev(past);
74      baseline::parallel_iter_swap(std::make_pair(first, last), poles);
75      if (*first == *last) {
76        return std::next(first);
77      }
78      auto range = cphstl::range(std::next(first), last);
79      I mid = baseline::partition_above_below(range, first, last);
80      using T = decltype((*first).x);
81      I eliminated1 = recurse(first, mid, last, std::less<T>());
82      std::iter_swap(mid, last);
83      I eliminated2 = recurse(mid, past, first, std::greater<T>());
        ↪ // downunder
84      auto second_chain = cphstl::range(mid, eliminated2);
85      I eliminated = baseline::swap_ranges(second_chain, eliminated1);
86      return eliminated;
87    }
88
89    template<typename R>
90    requires cphmpl::specifies_range<R> and
        ↪ is_point<cphmpl::value<R>>
91    cphmpl::iterator<R> solve(R& sequence) {
92      return solve(std::begin(sequence), std::end(sequence));
93    }
94
95    template<typename I>
96    requires cphmpl::is_iterator<I> and is_point<cphmpl::value<I>>
97    bool check(I first, I past) {
98      using P = cphmpl::value<I>;
99      using S = std::vector<P>;
100     using J = typename S::iterator;
```

```
101       S data;
102       std::size_t n = std::distance(first, past);
103       data.resize(n);
104       std::copy(first, past, data.begin());
105       J rest = solve(data.begin(), data.end());
106       bool ok = validation::same_multiset(data.begin(), data.end(),
          ↪ first, past) and validation::convex_polygon(data.begin(),
          ↪ rest) and validation::all_inside(rest, data.end(),
          ↪ data.begin(), rest);
107       return ok;
108     }
109
110     template<typename R>
111     requires cphmpl::specifies_range<R> and
          ↪ is_point<cphmpl::value<R>>
112     bool check(R const& sequence) {
113       return check(std::cbegin(sequence), std::cend(sequence));
114     }
115   }
116
117   #endif
```

B.3 opt1.h++

```
1   /*
2     Performance Engineering Laboratory © 2017–2019
3   */
4
5   #ifndef __QUICKHULL_OPT1__
6   #define __QUICKHULL_OPT1__
7
8   #define PARTIALLY_EVALUATED
9
10  #include "point.h++" // signed_area left_turn no_turn
11
12  #include <algorithm> // std::minmax_element std::partitioning ...
13  #include "baseline.h++" // baseline::parallel_iter_swap ...
14  #include <cassert> // assert macro
15  #include "cphmpl/functions.h++" // cphmpl type functions
16  #include "cphstl/ranges.h++" // cphstl ranges
17  #include <cstdlib> // std::size_t
18  #include <iterator> // std::begin std::end std::size std::empty ...
19  #include <type_traits> // std::is_same_v
20  #include <utility> // std::pair std::make_pair ...
21  #include "validation.h++" // same_multiset convex_polygon all_inside
22  #include <vector> // std::vector
23
24  namespace opt1 {
25
26    template<typename R, typename I>
27    requires cphmpl::specifies_range<R> and
```

```
    ↪ is_point<cphmpl::value<R>> and cphmpl::is_iterator<I>
    ↪ and is_point<cphmpl::value<I>>
I partition_above_below(R& range, I pole, I antipole) {
  I first = std::begin(range);
  I past = std::end(range);
  using P = cphmpl::value<I>;
  partially_evaluated::left_turn on_the_left(*pole, *antipole);
  I mid = std::partition(first, past,
    [&](P const& r) -> bool {
      return on_the_left(r);
    });
  return mid;
}

template<typename I, typename C, typename T = typename
  ↪ cphmpl::value<I>::coordinate>
requires
/* 1 */ cphmpl::is_iterator<I> and
/* 2 */ is_point<cphmpl::value<I>> and
/* 3 */ cphmpl::is_binary_predicate<C, T, T>
I recurse(I pole, I past, I antipole, C less) {
  std::size_t n = std::distance(pole, past);
  if (n ≤ 1) {
    return past;
  }
  if (n == 2) {
    if (no_turn(*std::next(pole), *pole, *antipole)) {
      return std::next(pole);
    }
    else {
      return past;
    }
  }
  auto whole = cphstl::range(std::next(pole), past);
  I pivot = baseline::find_furthest(whole, pole, antipole, less);
  if (pivot == pole) {
    return std::next(pole);
  }
  I last = std::prev(past);
  std::iter_swap(pivot, last); // pivot at the end
  auto range = cphstl::range(std::next(pole), last);
  I mid = opt1::partition_above_below(range, pole, last);
  I eliminated1 = recurse(pole, mid, last, less);
  std::iter_swap(mid, last); // pivot in the middle
  auto rests = cphstl::range(std::next(mid), past);
  I interior = opt1::partition_above_below(rests, mid, antipole);
  I eliminated2 = recurse(mid, interior, antipole, less);
  auto second_chain = cphstl::range(mid, eliminated2);
  I eliminated = baseline::swap_ranges(second_chain, eliminated1);
  return eliminated;
}

```

```
77  template<typename I>
78  requires cphmpl::is_iterator<I> and is_point<cphmpl::value<I>>
79  I solve(I first, I past) {
80    std::size_t n = std::distance(first, past);
81    if (n < 2 or (n == 2 and *first != *std::next(first))) {
82      return past;
83    }
84    auto whole = cphstl::range(first, past);
85    std::pair<I, I> poles = baseline::find_poles(whole);
86    I last = std::prev(past);
87    baseline::parallel_iter_swap(std::make_pair(first, last), poles);
88    if (*first == *last) {
89      return std::next(first);
90    }
91    auto range = cphstl::range(std::next(first), last);
92    I mid = opt1::partition_above_below(range, first, last);
93    using T = decltype((*first).x);
94    I eliminated_1 = recurse(first, mid, last, std::less<T>());
95    std::iter_swap(mid, last);
96    I eliminated_2 = recurse(mid, past, first, std::greater<T>());
97    auto second_chain = cphstl::range(mid, eliminated_2);
98    I eliminated = baseline::swap_ranges(second_chain, eliminated_1);
99    return eliminated;
100 }
101
102 template<typename R>
103 requires cphmpl::specifies_range<R> and
      ↪ is_point<cphmpl::value<R>>
104 cphmpl::iterator<R> solve(R& sequence) {
105   return solve(std::begin(sequence), std::end(sequence));
106 }
107
108 template<typename I>
109 requires cphmpl::is_iterator<I> and is_point<cphmpl::value<I>>
110 bool check(I first, I past) {
111   using P = cphmpl::value<I>;
112   using S = std::vector<P>;
113   using J = cphmpl::iterator<S>;
114   S data;
115   std::size_t n = std::distance(first, past);
116   data.resize(n);
117   std::copy(first, past, data.begin());
118   J rest = solve(data.begin(), data.end());
119   bool ok = validation::same_multiset(data.begin(), data.end(),
      ↪ first, past) and validation::convex_polygon(data.begin(),
      ↪ rest) and validation::all_inside(rest, data.end(),
      ↪ data.begin(), rest);
120   return ok;
121 }
122
123 template<typename R>
124 requires cphmpl::specifies_range<R> and
```

```
    ↪ is_point<cphmpl::value<R>>
  bool check(R const& sequence) {
    return check(std::cbegin(sequence), std::cend(sequence));
  }
}

#endif
```

B.4 opt2.h++

```
/*
  Author: Jyrki Katajainen © 2019

  Only optimization 2: Redundancy removal
*/

#ifndef __QUICKHULL_OPT2__
#define __QUICKHULL_OPT2__

#define MULTIPLE_PRECISION

#include "point.h++"

#include "cphmpl/functions.h++" // cphmpl::width
#include "cphstl/ranges.h++" // cphstl ranges

#include <algorithm> // std::minmax_element ...
#include "baseline.h++" // baseline::parallel_iter_swap ...
#include <cassert> // assert macro
#include <cstdlib> // std::size_t
#include <functional> // std::less std::greater
#include <iterator> // std::begin std::end std::cbegin std::cend
#include <tuple> // std::tuple std::make_tuple std::tie
#include "validation.h++" // same_multiset convex_polygon all_inside
#include <vector> // std::vector
#include <type_traits> // std::is:same_v

namespace opt2 {

  template<typename Z, typename I, typename F>
  void update_max_if(Z Φ, Z& max, I& above, I here, F key) {
    if (Φ > max) {
      max = Φ;
      above = here;
    }
    else if (Φ == max and key(*here) < key(*above)) {
      max = Φ;
      above = here;
    }
  }

```

```
42  template<typename Z, typename I, typename F>
43  void update_min_if(Z Φ, Z& min, I& below, I here, F key) {
44    if (Φ < min) {
45      min = Φ;
46      below = here;
47    }
48    else if (Φ == min and key(*here) > key(*below)) {
49      min = Φ;
50      below = here;
51    }
52  }
53
54  template<typename R, typename I, typename F>
55  requires
56  /* 1 */ cphmpl::specifies_range<R> and
57  /* 2 */ cphmpl::is_iterator<I> and
58  /* 3 */ is_point<cphmpl::value<I>> and
59  /* 4 */ std::is_same_v<I, cphmpl::iterator<R>> and
60  /* 5 */ cphmpl::is_unary_function<F, cphmpl::value<I>>
61  std::tuple<I, I, I> partition_above_below(R& range, I p, I q, F
      ↪ key) {
62    I first = std::begin(range);
63    I last = std::end(range);
64    using Z = decltype(signed_area(*p, *p, *p));
65    Z min = Z();
66    Z max = Z();
67    I above = p;
68    I below = q;
69    while (true) {
70      while (true) {
71        if (first == last) {
72          return std::make_tuple(first, above, below);
73        }
74        else {
75          Z Φ = signed_area(*p, *q, *first);
76          if (Φ > Z()) {
77            update_max_if(Φ, max, above, first, key);
78            ++first;
79          }
80          else {
81            update_min_if(Φ, min, below, first, key);
82            break;
83          }
84        }
85      }
86      --last;
87      while (true) {
88        if (first == last) {
89          return std::make_tuple(first, above, below);
90        }
91        else {
```

```
92            Z Φ = signed_area(*p, *q, *last);
93            if (Φ ≤ Z()) {
94              update_min_if(Φ, min, below, last, key);
95              --last;
96            }
97            else {
98              update_max_if(Φ, max, above, last, key);
99              break;
100           }
101         }
102       }
103       std::iter_swap(first, last);
104       if (first == below) { // referential integrity
105         below = last;
106       }
107       if (last == above) {
108         above = first;
109       }
110       ++first;
111     }
112   }
113
114   template<typename I>
115   requires cphmpl::is_iterator<I> and is_point<cphmpl::value<I>>
116   std::tuple<I, I, I> partition_above_below(I p, I rest, I r) {
117     assert(p ≠ rest);
118     using P = decltype(*p);
119     using T = decltype((*p).x);
120     auto range = cphstl::range(std::next(p), rest);
121     std::tuple<I, I, I> output;
122     if ((*p).x == (*r).x) { // vertical
123       output = partition_above_below(range, p, r,
124         [](P const& a) → T {
125           return a.y;
126         });
127       return output;
128     }
129     output = partition_above_below(range, p, r,
130       [](P const& a) → T {
131         return a.x;
132       });
133     return output;
134   }
135
136   template<typename R, typename I, typename C, typename T =
        ↪ typename cphmpl::value<I>::coordinate>
137   requires
138   /* 1 */ cphmpl::specifies_range<R> and
139   /* 2 */ cphmpl::is_iterator<I> and
140   /* 3 */ is_point<cphmpl::value<I>> and
141   /* 4 */ std::is_same_v<I, cphmpl::iterator<R>> and
```

```
142   /* 5 */ cphmpl::is_binary_predicate<C, T, T>
143   std::tuple<I, I> partition_two_way(R& range, I p, I r, C less) {
144     using Z = decltype(signed_area(*p, *p, *p));
145     Z max = Z();
146     I above = p;
147     I i = std::begin(range); // end of region R1
148     I j = std::end(range); // query position
149     is_lexicographically_smaller<C, T> tiebreaker(less);
150     while (i != j) {
151       --j;
152       Z Φ = signed_area(*p, *r, *j);
153       if (Φ > Z()) {
154         if (Φ > max or (Φ == max and tiebreaker(*j, *above))) {
155           max = Φ;
156           above = i;
157         }
158         std::iter_swap(i, j);
159         ++i;
160         ++j;
161       }
162     }
163     return std::make_tuple(i, above);
164   }
165
166   template<typename R, typename I, typename C, typename T =
        ↪ typename cphmpl::value<I>::coordinate>
167   requires
168   /* 1 */ cphmpl::specifies_range<R> and
169   /* 2 */ cphmpl::is_iterator<I> and
170   /* 3 */ is_point<cphmpl::value<I>> and
171   /* 4 */ std::is_same_v<I, cphmpl::iterator<R>> and
172   /* 5 */ cphmpl::is_binary_predicate<C, T, T>
173   std::tuple<I, I, I, I> eliminate(R& range, I p, I q, I r, C
        ↪ less) {
174     // range contains a collection; p, q, r specify points outside
175     I m;
176     I q1;
177     std::tie(m, q1) = partition_two_way(range, p, q, less);
178     auto rests = cphstl::range(m, std::end(range));
179     I t;
180     I q2;
181     std::tie(t, q2) = partition_two_way(rests, q, r, less);
182     return std::make_tuple(m, t, q1, q2);
183   }
184
185   template<typename I, typename C, typename T = typename
        ↪ cphmpl::value<I>::coordinate>
186   requires
187   /* 1 */ cphmpl::is_iterator<I> and
188   /* 2 */ is_point<cphmpl::value<I>> and
189   /* 3 */ cphmpl::is_binary_predicate<C, T, T>
190   I recurse(I pole, I past, I pivot, I antipole, C less) {
```

```
191     // pole first; pivot one of [pole, past); antipole outside
192     if (pole == pivot) {
193       return std::next(pole);
194     }
195     I last = std::prev(past);
196     std::iter_swap(pivot, last);
197     I r_1;
198     I r_2;
199     I q_1;
200     I q_2;
201     auto range = cphstl::range(std::next(pole), last);
202     std::tie(r_1, r_2, q_1, q_2) = eliminate(range, pole, last,
        ↪ antipole, less);
203     I rest_1 = recurse(pole, r_1, q_1, last, less);
204     std::iter_swap(r_2, last);
205     std::iter_swap(r_1, r_2);
206     if (q_2 == last) {
207       q_2 = r_1;
208     }
209     else if (q_2 == r_1) {
210       q_2 = r_2;
211     }
212     ++r_2;
213     I rest_2 = recurse(r_1, r_2, q_2, antipole, less);
214     auto second_chain = cphstl::range(r_1, rest_2);
215     I eliminated = baseline::swap_ranges(second_chain, rest_1);
216     return eliminated;
217   }
218
219   template<typename I>
220   requires cphmpl::is_iterator<I> and is_point<cphmpl::value<I>>
221   I solve(I first, I past) {
222     std::size_t n = std::distance(first, past);
223     if (n < 2 or (n == 2 and *first != *std::next(first))) {
224       return past;
225     }
226     auto whole = cphstl::range(first, past);
227     std::pair<I, I> poles = baseline::find_poles(whole);
228     I last = std::prev(past);
229     baseline::parallel_iter_swap(std::make_pair(first, last), poles);
230     if (*first == *last) {
231       return std::next(first);
232     }
233     I mid;
234     I above;
235     I below;
236     std::tie(mid, above, below) = partition_above_below(first,
        ↪ last, last);
237
238     using T = decltype((*first).x);
239     I rest_1 = recurse(first, mid, above, last, std::less<T>());
240     std::iter_swap(mid, last);
```

```
241       if (below == last) { // referential integrity
242         below = mid;
243       }
244       else if (below == mid) {
245         below = last;
246       }
247       I rest_2 = recurse(mid, past, below, first, std::greater<T>());
248       auto second_chain = cphstl::range(mid, rest_2);
249       I eliminated = baseline::swap_ranges(second_chain, rest_1);
250       return eliminated;
251     }
252
253     template<typename R>
254     requires cphmpl::specifies_range<R> and
            ↪ is_point<cphmpl::value<R>>
255     cphmpl::iterator<R> solve(R& sequence) {
256       return solve(std::begin(sequence), std::end(sequence));
257     }
258
259     template<typename I>
260     requires cphmpl::is_iterator<I> and is_point<cphmpl::value<I>>
261     bool check(I first, I past) {
262       using P = cphmpl::value<I>;
263       using S = std::vector<P>;
264       using J = cphmpl::iterator<S>;
265       S data;
266       std::size_t n = std::distance(first, past);
267       data.resize(n);
268       std::copy(first, past, data.begin());
269       J rest = solve(data.begin(), data.end());
270       bool ok = validation::same_multiset(data.begin(), data.end(),
            ↪ first, past) and validation::convex_polygon(data.begin(),
            ↪ rest) and validation::all_inside(rest, data.end(),
            ↪ data.begin(), rest);
271       return ok;
272     }
273
274     template<typename R>
275     requires cphmpl::specifies_range<R> and
            ↪ is_point<cphmpl::value<R>>
276     bool check(R const& sequence) {
277       return check(std::cbegin(sequence), std::cend(sequence));
278     }
279   }
280
281   #endif
```

B.5 opt3.h++

```
1   /*
2     Author: Jyrki Katajainen © 2019
```

```
3
4     All optimizations 1, 2, and 3
5   */
6
7   #ifndef __QUICKHULL_OPT3__
8   #define __QUICKHULL_OPT3__
9
10  #define PARTIALLY_EVALUATED
11
12  #include "point.h++"
13
14  #include "cphmpl/functions.h++" // cphmpl::width
15  #include "cphstl/ranges.h++" // cphstl ranges
16
17  #include <algorithm> // std::minmax_element ...
18  #include "baseline.h++" // baseline::parallel_iter_swap ...
19  #include <cassert> // assert macro
20  #include <cstdlib> // std::size_t
21  #include <functional> // std::less std::greater
22  #include <iterator> // std::begin std::end std::cbegin std::cend ..
        ↪ .
23  #include <tuple> // std::tuple std::make_tuple std::tie
24  #include "validation.h++" // same_multiset convex_polygon all_inside
25  #include <vector> // std::vector
26  #include <type_traits> // std::is:same_v
27
28  namespace opt3 {
29
30    // move *st → *nd, *nd → *rd, and *rd → *st in parallel
31
32    template<typename I>
33    requires cphmpl::is_iterator<I>
34    void cyclic_iter_roll(I st, I nd, I rd) {
35      auto t = *rd;
36      *rd = *nd;
37      *nd = *st;
38      *st = t;
39    }
40
41    template<typename Z, typename I, typename F>
42    void update_max_if(Z Φ, Z& max, I& above, I here, F key) {
43      if (Φ > max) {
44        max = Φ;
45        above = here;
46      }
47      else if (Φ == max and key(*here) < key(*above)) {
48        max = Φ;
49        above = here;
50      }
51    }
52
53    template<typename Z, typename I, typename F>
```

```
54    void update_min_if(Z Φ, Z& min, I& below, I here, F key) {
55      if (Φ < min) {
56        min = Φ;
57        below = here;
58      }
59      else if (Φ == min and key(*here) > key(*below)) {
60        min = Φ;
61        below = here;
62      }
63    }
64
65    template<typename R, typename I, typename F>
66    requires
67    /* 1 */ cphmpl::specifies_range<R> and
68    /* 2 */ cphmpl::is_iterator<I> and
69    /* 3 */ is_point<cphmpl::value<I>> and
70    /* 4 */ std::is_same_v<I, cphmpl::iterator<R>> and
71    /* 5 */ cphmpl::is_unary_function<F, cphmpl::value<I>>
72    std::tuple<I, I, I> partition_above_below(R& range, I p, I q, F
        ↪ key) {
73      I first = std::begin(range);
74      I last = std::end(range);
75      partially_evaluated::signed_area formula(*p, *q);
76      using Z = decltype(formula(*p));
77      Z min = Z();
78      Z max = Z();
79      I above = p;
80      I below = q;
81      while (true) {
82        while (true) {
83          if (first == last) {
84            return std::make_tuple(first, above, below);
85          }
86          else {
87            Z Φ = formula(*first);
88            if (Φ > Z()) {
89              update_max_if(Φ, max, above, first, key);
90              ++first;
91            }
92            else {
93              update_min_if(Φ, min, below, first, key);
94              break;
95            }
96          }
97        }
98        --last;
99        while (true) {
100         if (first == last) {
101           return std::make_tuple(first, above, below);
102         }
103         else {
104           Z Φ = formula(*last);
```

```
105           if (Φ ≤ Z()) {
106             update_min_if(Φ, min, below, last, key);
107             --last;
108           }
109           else {
110             update_max_if(Φ, max, above, last, key);
111             break;
112           }
113         }
114       }
115       std::iter_swap(first, last);
116       if (first == below) { // referential integrity
117         below = last;
118       }
119       if (last == above) {
120         above = first;
121       }
122       ++first;
123     }
124   }
125
126   template<typename I>
127   requires cphmpl::is_iterator<I> and is_point<cphmpl::value<I>>
128   std::tuple<I, I, I> partition_above_below(I p, I rest, I r) {
129     assert(p ≠ rest);
130     using P = cphmpl::value<I>;
131     using T = decltype((*p).x);
132     auto range = cphstl::range(std::next(p), rest);
133     std::tuple<I, I, I> output;
134     if ((*p).x == (*r).x) { // vertical
135       output = partition_above_below(range, p, r,
136         [](P const& a) → T {
137           return a.y;
138         });
139       return output;
140     }
141     output = partition_above_below(range, p, r,
142       [](P const& a) → T {
143         return a.x;
144       });
145     return output;
146   }
147
148   template<typename R, typename I, typename C, typename T =
      ↪ typename cphmpl::value<I>::coordinate>
149   requires
150   /* 1 */ cphmpl::specifies_range<R> and
151   /* 2 */ cphmpl::is_iterator<I> and
152   /* 3 */ is_point<cphmpl::value<I>> and
153   /* 4 */ std::is_same_v<I, cphmpl::iterator<R>> and
154   /* 5 */ cphmpl::is_binary_predicate<C, T, T>
155   std::tuple<I, I, I, I> eliminate(R& range, I p, I q, I r, C
```

```
    ↪ less) {
156   // range contains a collection; p, q, r specify points outside
157   assert(not std::empty(range));
158   partially_evaluated::signed_area formula1(*p, *q);
159   partially_evaluated::signed_area formula2(*q, *r);
160   using P = decltype(*p);
161   auto on_the_left = [&](P const& a) → bool {return less(a.x,
      ↪ (*q).x);};
162   is_lexicographically_smaller<C, T> tiebreak(less);
163   using Z = decltype(formula1(*p));
164   Z max1 = Z();
165   Z max2 = Z();
166   I q1 = p;
167   I q2 = q;
168   I i = std::begin(range); // end of region R1
169   I j = std::begin(range); // end of region R2
170   I k = std::end(range); // query position; front of the interior
171   while (k ≠ j) {
172     --k;
173     if (on_the_left(*k)) {
174       Z Φ1 = formula1(*k);
175       if (Φ1 > Z()) {
176         if (Φ1 > max1 or (Φ1 == max1 and tiebreak(*k, *q1))) {
177           max1 = Φ1;
178           q1 = i;
179         }
180         cyclic_iter_roll(i, j, k);
181         if (q2 == i) { // referential integrity
182           q2 = j;
183         }
184         ++i;
185         ++j;
186         ++k;
187       }
188     }
189     else {
190       Z Φ2 = formula2(*k);
191       if (Φ2 > Z()) {
192         if (Φ2 > max2 or (Φ2 == max2 and tiebreak(*k, *q2))) {
193           max2 = Φ2;
194           q2 = j;
195         }
196         std::iter_swap(j, k);
197         ++j;
198         ++k;
199       }
200     }
201   }
202   return std::make_tuple(i, j, q1, q2);
203 }
204
205 template<typename I, typename C, typename T = typename
```

```
    ↪ cphmpl::value<I>::coordinate>
206 requires
207 /* 1 */ cphmpl::is_iterator<I> and
208 /* 2 */ is_point<cphmpl::value<I>> and
209 /* 3 */ cphmpl::is_binary_predicate<C, T, T>
210 I recurse(I pole, I past, I pivot, I antipole, C less) {
211   // pole first; pivot one of [pole, past); antipole outside
212   if (pole == pivot or std::next(pole) == past) {
213     return std::next(pole);
214   }
215   if (std::next(std::next(pole)) == past) {
216     if (no_turn(*std::next(pole), *pole, *antipole)) {
217       return std::next(pole);
218     }
219     else {
220       return past;
221     }
222   }
223   I last = std::prev(past);
224   std::iter_swap(pivot, last);
225   I r1;
226   I r2;
227   I q1;
228   I q2;
229   auto range = cphstl::range(std::next(pole), last);
230   std::tie(r1, r2, q1, q2) = eliminate(range, pole, last,
    ↪ antipole, less);
231   I rest1 = recurse(pole, r1, q1, last, less);
232   std::iter_swap(r2, last);
233   std::iter_swap(r1, r2);
234   if (q2 == last) {
235     q2 = r1;
236   }
237   else if (q2 == r1) {
238     q2 = r2;
239   }
240   ++r2;
241   I rest2 = recurse(r1, r2, q2, antipole, less);
242   auto second_chain = cphstl::range(r1, rest2);
243   I eliminated = baseline::swap_ranges(second_chain, rest1);
244   return eliminated;
245 }
246
247 template<typename I>
248 requires cphmpl::is_iterator<I> and is_point<cphmpl::value<I>>
249 I solve(I first, I past) {
250   std::size_t n = std::distance(first, past);
251   if (n < 2 or (n == 2 and *first ≠ *std::next(first))) {
252     return past;
253   }
254   I last = std::prev(past);
255   auto whole = cphstl::range(first, past);
```

```
256     std::pair<I, I> poles = baseline::find_poles(whole);
257     baseline::parallel_iter_swap(std::make_pair(first, last), poles);
258     if (*first == *last) {
259       return std::next(first);
260     }
261     I mid;
262     I above;
263     I below;
264     std::tie(mid, above, below) = partition_above_below(first,
        ↪ last, last);
265     using T = decltype((*first).x);
266     I rest_1 = recurse(first, mid, above, last, std::less<T>());
267     std::iter_swap(mid, last);
268     if (below == last) { // referential integrity
269       below = mid;
270     }
271     else if (below == mid) {
272       below = last;
273     }
274     I rest_2 = recurse(mid, past, below, first, std::greater<T>());
275     auto second_chain = cphstl::range(mid, rest_2);
276     I eliminated = baseline::swap_ranges(second_chain, rest_1);
277     return eliminated;
278   }
279
280   template<typename R>
281   requires cphmpl::specifies_range<R> and
        ↪ is_point<cphmpl::value<R>>
282   cphmpl::iterator<R> solve(R& sequence) {
283     return solve(std::begin(sequence), std::end(sequence));
284   }
285
286   template<typename I>
287   requires cphmpl::is_iterator<I> and is_point<cphmpl::value<I>>
288   bool check(I first, I past) {
289     using P = cphmpl::value<I>;
290     using S = std::vector<P>;
291     using J = cphmpl::iterator<S>;
292     S data;
293     std::size_t n = std::distance(first, past);
294     data.resize(n);
295     std::copy(first, past, data.begin());
296     J rest = solve(data.begin(), data.end());
297     bool ok = validation::same_multiset(data.begin(), data.end(),
        ↪ first, past);
298     bool polygon = validation::convex_polygon(data.begin(), rest);
299     bool all_in = validation::all_inside(rest, data.end(),
        ↪ data.begin(), rest);
300     return ok and polygon and all_in;
301   }
302
303   template<typename R>
```

```
304     requires cphmpl::specifies_range<R> and
          ↪ is_point<cphmpl::value<R>>
305     bool check(R const& sequence) {
306       return check(std::cbegin(sequence), std::cend(sequence));
307     }
308   }
309
310   #endif
```

B.6 opt4.h++

```
1   /*
2     Performance Engineering Laboratory © 2017–2019
3
4     Find the extrema in eight directions and eliminate all points
5     inside the convex hull of these points.
6   */
7
8   #ifndef __QUICKHULL_OPT4__
9   #define __QUICKHULL_OPT4__
10
11  #include <algorithm> // std::sort std::min std::minmax_element
12  #include <cassert> // assert macro
13  #include <cmath> // std::sqrt
14  #include "cphmpl/functions.h++" // cphmpl::width
15  #include "cphstl/integers.h++" // cphstl::ℤ
16  #include <cstddef> // std::size_t
17  #include <iterator> // std::begin std::end std::cbegin std::cend
18
19  #define MULTIPLE_PRECISION
20  #define PARTIALLY_EVALUATED
21
22  #include "point.h++"
23
24  #include "opt3.h++"
25
26  #include <utility> // std::pair std::get
27  #include "validation.h++" // same_multiset convex_polygon all_inside
28  #include <vector> // std::vector
29
30  namespace opt4 {
31
32    template<typename I>
33    requires cphmpl::is_iterator<I> and is_point<cphmpl::value<I>>
34    std::vector<I> find_extrema(I first, I past) {
35      assert(first ≠ past);
36      using T = decltype((*first).x);
37      constexpr std::size_t w = cphmpl::width<T>;
38      using Z = cphstl::ℤ<w + 2>;
39      std::vector<I> max_position(8, first);
40      T xmin = (*first).x;
```

```
41      T xmax = (*first).x;
42      T ymin = (*first).y;
43      T ymax = (*first).y;
44      Z ne = Z(xmin) + Z(ymin);
45      Z se = Z(xmin) - Z(ymin);
46      Z sw = -(Z(xmin) + Z(ymin));
47      Z nw = Z(ymin) - Z(xmin);
48      for (I i = first; i != past; ++i) {
49        if ((*i).x < xmin) {
50          xmin = (*i).x;
51          max_position[0] = i;
52        }
53        if ((*i).x > xmax) {
54          xmax = (*i).x;
55          max_position[1] = i;
56        }
57        if ((*i).y < ymin) {
58          ymin = (*i).y;
59          max_position[2] = i;
60        }
61        if ((*i).y > ymax) {
62          ymax = (*i).y;
63          max_position[3] = i;
64        }
65        Z wide_x = Z((*i).x);
66        Z wide_y = Z((*i).y);
67        Z dot = wide_x + wide_y;
68        if (dot > ne) {
69          ne = dot;
70          max_position[4] = i;
71        }
72        dot = wide_x - wide_y;
73        if (dot > se) {
74          se = dot;
75          max_position[5] = i;
76        }
77        dot = -(wide_x + wide_y);
78        if (dot > sw) {
79          sw = dot;
80          max_position[6] = i;
81        }
82        dot = wide_x - wide_y;
83        if (dot > nw) {
84          nw = dot;
85          max_position[7] = i;
86        }
87      }
88      return max_position;
89    }
90
91    template<typename S>
92    requires cphmpl::specifies_range<S>
```

```
93    void remove_duplicates(S& sequence) {
94      assert(not std::empty(sequence));
95      assert(std::is_sorted(std::cbegin(sequence),
        ↪ std::cend(sequence)));
96      std::size_t i = 0;
97      std::size_t j = 1;
98      auto v = sequence[0];
99      while (j != std::size(sequence)) {
100       if (not (sequence[j] == v)) {
101         std::swap(sequence[i], v);
102         ++i;
103         v = sequence[j];
104       }
105       ++j;
106     }
107     std::swap(sequence[i], v);
108     ++i;
109     std::size_t leftovers = std::size(sequence) - i;
110     while (leftovers != 0) {
111       sequence.pop_back();
112       --leftovers;
113     }
114   }
115
116   template<typename P, typename I>
117   requires is_point<P> and cphmpl::is_iterator<I> and
        ↪ is_point<cphmpl::value<I>>
118   bool inside_convex_polygon(P const& p, I upper, I lower, I past) {
119     assert(past - upper > 2);
120     assert(*upper == *(past - 1));
121     assert(*upper < *(upper + 1));
122     assert(*upper < *(past - 2));
123     I i = upper;
124     do {
125       ++i;
126     } while (i != lower and *i < p);
127     if (left_turn(*(i - 1), *i, p)) {
128       return false;
129     }
130     I last = past - 1;
131     I j = lower;
132     do {
133       ++j;
134     } while (j != last and *j > p);
135     if (left_turn(*(j - 1), *j, p)) {
136       return false;
137     }
138     return true;
139   }
140
141   template<typename I>
142   requires cphmpl::is_iterator<I> and is_point<cphmpl::value<I>>
```

```
143  I eliminate_inner_points(I first, I past, I polygon, I rest) {
144    assert(polygon != rest);
145    using P = cphmpl::value<I>;
146    if (rest - polygon == 1) {
147      I mid = std::partition(first, past,
148        [&](P const& r) -> bool {
149          return not (r == *polygon);
150        });
151      return mid;
152    }
153    if (rest - polygon == 2) {
154      I mid = std::partition(first, past,
155        [&](P const& r) -> bool {
156          return not on_line_segment(*polygon, *(polygon + 1), r);
157        });
158      return mid;
159    }
160    // rest - polygon > 2
161    std::vector<P> approximate_hull;
162    for (I k = polygon; k != rest; ++k) {
163      approximate_hull.push_back(*k);
164    }
165    approximate_hull.push_back(*polygon);
166    auto upper = approximate_hull.begin();
167    auto lower = upper;
168    auto end = approximate_hull.end();
169    for (auto k = upper + 1; k != end - 1; ++k) {
170      if (*lower < *k) {
171        lower = k;
172      }
173    }
174    I mid = std::partition(first, past,
175      [&](P const& r) -> bool {
176        return not inside_convex_polygon(r, upper, lower, end);
177      });
178    return mid;
179  }
180
181  template<typename I>
182  requires cphmpl::is_iterator<I> and is_point<cphmpl::value<I>>
183  I prune(I first, I past) {
184    if (past - first < 2) {
185      return past;
186    }
187    // Step 1
188    std::vector<I> extrema = find_extrema(first, past);
189    std::sort(extrema.begin(), extrema.end());
190    remove_duplicates(extrema);
191    for (std::size_t i = 0; i != extrema.size(); ++i) {
192      std::iter_swap(first + i, extrema[i]);
193    }
194    // Step 2
```

```
195       I rest = opt3::solve(first, first + extrema.size());
196       // Step 3
197       I interior = eliminate_inner_points(rest, past, first, rest);
198       return interior;
199     }
200
201     template<typename I>
202     requires cphmpl::is_iterator<I> and is_point<cphmpl::value<I>>
203     I solve(I first, I past) {
204       I rest = prune(first, past);
205       return opt3::solve(first, rest);
206     }
207
208     template<typename R>
209     requires cphmpl::specifies_range<R> and
          ↪ is_point<cphmpl::value<R>>
210     cphmpl::iterator<R> solve(R& sequence) {
211       return solve(std::begin(sequence), std::end(sequence));
212     }
213
214     template<typename I>
215     requires cphmpl::is_iterator<I> and is_point<cphmpl::value<I>>
216     bool check(I first, I past) {
217       using P = cphmpl::value<I>;
218       using S = std::vector<P>;
219       using J = cphmpl::iterator<S>;
220       S data;
221       std::size_t n = std::distance(first, past);
222       data.resize(n);
223       std::copy(first, past, data.begin());
224       J rest = solve(data.begin(), data.end());
225       bool ok = validation::same_multiset(data.begin(), data.end(),
          ↪ first, past) and validation::convex_polygon(data.begin(),
          ↪ rest) and validation::all_inside(rest, data.end(),
          ↪ data.begin(), rest);
226       return ok;
227     }
228
229     template<typename R>
230     requires cphmpl::specifies_range<R> and
          ↪ is_point<cphmpl::value<R>>
231     bool check(R const& sequence) {
232       return check(std::cbegin(sequence), std::cend(sequence));
233     }
234   }
235
236   #endif
```

C. Micro-benchmarks

C.1 partition_baseline.h++

```
/*
  This dummy partitions the sequence into two by the line which
  passes through two points
*/

#include <iterator>

#define MULTIPLE_PRECISION
#include "point.h++"

namespace partition_baseline {

  template<typename I>
  I solve(I first, I past) {
    using P = typename std::iterator_traits<I>::value_type;
    P const p = *first;
    I last = std::prev(past);
    P const q = *last;
    I mid = std::partition(std::next(first), last,
      [&](P const& r) -> bool {
        return left_turn(p, q, r);
      });
    return mid;
  }
}
```

C.2 partition_opt1.h++

```
/*
  This dummy partitions the sequence into two by the line which
  passes through two points
*/

#include <iterator> // std::iterator_traits

#define PARTIALLY_EVALUATED
#include "point.h++"

namespace partition_opt1 {

  template<typename I>
  I solve(I first, I past) {
    using P = typename std::iterator_traits<I>::value_type;
    I last = std::prev(past);
    partially_evaluated::left_turn on_the_left(*first, *last);
    I mid = std::partition(std::next(first), last,
      [&](P const& r) -> bool {
```

```
20          return on_the_left(r);
21        });
22      return mid;
23    }
24  }
```

D. Helpers

D.1 point.h++

```
#ifndef __POINT__
#define __POINT__

#include <algorithm> // std::min std::max
#include <cassert> // assert macro
#include "cphmpl/functions.h++" // cphmpl::width
#include "cphstl/integers.h++" // cphstl::ℤ
#include <cstddef> // std::size_t
#include <functional> // std::less
#include <limits> // std::numeric_limits
#include <string> // std::string std::to_string
#include <tuple> // std::tuple

// is_point

template<typename P>
concept bool is_point =
  requires (P p, P q) {
  typename P::coordinate;
  {p.x} -> typename P::coordinate;
  {p.y} -> typename P::coordinate;
  P();
  P(p.x, p.y);
  {p == q} -> bool;
  {p != q} -> bool;
};

// is_line_segment

template<typename L>
concept bool is_line_segment =
  cphmpl::is_tuple<L> and std::tuple_size<L>::value == 2 and
  is_point<typename std::tuple_element<0, L>::type> and
  is_point<typename std::tuple_element<1, L>::type>;

// is_triangle

template<typename T>
concept bool is_triangle =
  cphmpl::is_tuple<T> and std::tuple_size<T>::value == 3 and
  is_point<typename std::tuple_element<0, T>::type> and
  is_point<typename std::tuple_element<1, T>::type> and
  is_point<typename std::tuple_element<2, T>::type>;

// is_polygon

template<typename G>
concept bool is_polygon =
```

```
49    cphmpl::specifies_range<G> and is_point<cphmpl::value<G>>;
50
51  // point
52
53  template<typename T>
54  requires cphmpl::is_integer<T>
55  class point {
56  public:
57
58    using self = point<T>;
59    using coordinate = T;
60
61    T x;
62    T y;
63
64    explicit point()
65      : x(T()), y(T()) {
66    }
67
68    point(T x_coordinate, T y_coordinate)
69      : x(x_coordinate), y(y_coordinate) {
70    }
71
72    std::string to_string() const {
73      return "(" + std::to_string(x) + ", " + std::to_string(y) + ")";
74    }
75
76    bool operator==(point<T> const& other) const {
77      return x == other.x and y == other.y;
78    }
79
80    bool operator!=(point<T> const& other) const {
81      return not (*this == other);
82    }
83
84    bool operator<(point<T> const& other) const {
85      return (x < other.x) or (x == other.x and y < other.y);
86    }
87
88    bool operator>(point<T> const& other) const {
89      return other < *this;
90    }
91
92    friend std::ostream& operator<<(std::ostream& os, point<T>
        ↪ const& p) {
93      return os << p.to_string();
94    }
95  };
96
97  #if defined(MULTIPLE_PRECISION) or defined(FLOATING_POINT_FILTER) or
        ↪ defined(PARTIALLY_EVALUATED)
98
```

```
99  namespace multiple_precision {
100
101   template<typename P>
102   requires is_point<P>
103   bool left_turn(P const& p, P const& q, P const& r) {
104
105 #ifdef MEASURE_TURNS
106
107     ++turns;
108
109 #endif
110
111     using N = std::size_t;
112     using T = decltype(p.x);
113     constexpr N w = cphmpl::width<T>;
114     using Z = cphstl::ℤ<2 * w + 4>;
115     Z p_x(p.x);
116     Z p_y(p.y);
117     Z q_x(q.x);
118     Z q_y(q.y);
119     Z r_x(r.x);
120     Z r_y(r.y);
121     Z lhs = (q_x - p_x) * (r_y - p_y);
122     Z rhs = (r_x - p_x) * (q_y - p_y);
123     return lhs > rhs;
124   }
125
126   template<typename P>
127   requires is_point<P>
128   bool no_turn(P const& p, P const& q, P const& r) {
129
130 #ifdef MEASURE_TURNS
131
132     ++turns;
133
134 #endif
135
136     using N = std::size_t;
137     using T = decltype(p.x);
138     constexpr N w = cphmpl::width<T>;
139     using Z = cphstl::ℤ<2 * w + 4>;
140     Z p_x(p.x);
141     Z p_y(p.y);
142     Z q_x(q.x);
143     Z q_y(q.y);
144     Z r_x(r.x);
145     Z r_y(r.y);
146     Z lhs = (q_x - p_x) * (r_y - p_y);
147     Z rhs = (r_x - p_x) * (q_y - p_y);
148     return lhs == rhs;
149   }
150
```

```
151   // compute the signed area of the parallelogram spanned by
152   // $\overrightarrow{pq}$ and $\overrightarrow{pr}$.
153
154   template<typename P>
155   requires is_point<P>
156   auto signed_area(P const& p, P const& q, P const& r) {
157
158 #ifdef MEASURE_TURNS
159
160     ++turns;
161
162 #endif
163
164     using N = std::size_t;
165     using T = decltype(p.x);
166     constexpr N w = cphmpl::width<T>;
167     using Z = cphstl::ℤ<2 * w + 5>;
168     Z p_x(p.x);
169     Z p_y(p.y);
170     Z q_x(q.x);
171     Z q_y(q.y);
172     Z r_x(r.x);
173     Z r_y(r.y);
174     Z lhs = (q_x - p_x) * (r_y - p_y);
175     Z rhs = (r_x - p_x) * (q_y - p_y);
176     Z result = lhs - rhs;
177     return result;
178   }
179
180   // return the square of the distance between p and q
181
182   template<typename P>
183   requires is_point<P>
184   auto distance_squared(P const& p, P const& q) {
185     using N = std::size_t;
186     using T = decltype(p.x);
187     constexpr N w = cphmpl::width<T>;
188     using Z = cphstl::ℤ<2 * w + 5>;
189     Z p_x = Z(p.x);
190     Z p_y = Z(p.y);
191     Z q_x = Z(q.x);
192     Z q_y = Z(q.y);
193     return (p_x - q_x) * (p_x - q_x) + (p_y - q_y) * (p_y - q_y);
194   }
195
196   template<typename P>
197   requires is_point<P>
198   auto L_∞distance(P const& p, P const& q) {
199     using N = std::size_t;
200     using T = decltype(p.x);
201     constexpr N w = cphmpl::width<T>;
202     using Z = cphstl::ℤ<w + 2>;
```

```
203       Z p_x = Z(p.x);
204       Z p_y = Z(p.y);
205       Z q_x = Z(q.x);
206       Z q_y = Z(q.y);
207       Z dx = (q_x > p_x) ? q_x - p_x : p_x - q_x;
208       Z dy = (q_y > p_y) ? q_y - p_y : p_y - q_y;
209       return std::max(dx, dy);
210     }
211
212     template<typename P>
213     requires is_point<P>
214     auto L_1_distance(P const& p, P const& q) {
215       using N = std::size_t;
216       using T = decltype(p.x);
217       constexpr N w = cphmpl::width<T>;
218       using Z = cphstl::ℤ<w + 3>;
219       Z p_x = Z(p.x);
220       Z p_y = Z(p.y);
221       Z q_x = Z(q.x);
222       Z q_y = Z(q.y);
223       Z dx = (q_x > p_x) ? q_x - p_x : p_x - q_x;
224       Z dy = (q_y > p_y) ? q_y - p_y : p_y - q_y;
225       return dx + dy;
226     }
227
228     // return the orientation when moving from p to r via q
229     // +1: left turn
230     //  0: p, q, and r are collinear
231     // -1: right turn
232
233     template<typename P>
234     requires is_point<P>
235     int orientation(P const& p, P const& q, P const& r) {
236       using N = std::size_t;
237       using T = decltype(p.x);
238       constexpr N w = cphmpl::width<T>;
239       using Z = cphstl::ℤ<2 * w + 4>;
240       Z p_x = Z(p.x);
241       Z p_y = Z(p.y);
242       Z q_x = Z(q.x);
243       Z q_y = Z(q.y);
244       Z r_x = Z(r.x);
245       Z r_y = Z(r.y);
246       Z lhs = (q_x - p_x) * (r_y - p_y);
247       Z rhs = (r_x - p_x) * (q_y - p_y);
248       if (lhs == rhs) {
249         return 0;
250       }
251       return (lhs > rhs) ? +1 : -1;
252     }
253   }
254
```

```
255 #endif
256
257 #if defined(PARTIALLY_EVALUATED)
258
259 namespace partially_evaluated {
260
261   template<typename P>
262   requires is_point<P>
263   class left_turn {
264   public:
265
266     using T = typename P::coordinate;
267     static constexpr std::size_t w = cphmpl::width<T>;
268     using Z = cphstl::ℤ<2 * w + 4>;
269
270     // lhs = (q_x - p_x) * (r_y - p_y);
271     // rhs = (r_x - p_x) * (q_y - p_y);
272
273     left_turn(P const& p, P const& q)
274       : p_x(p.x), p_y(p.y), precomputed_dx(Z(q.x) - Z(p.x)),
275         precomputed_dy(Z(q.y) - Z(p.y)) {
276     }
277
278     bool operator()(P const& r) {
279
280 #ifdef MEASURE_TURNS
281
282     ++turns;
283
284 #endif
285
286       Z r_x(r.x);
287       Z r_y(r.y);
288       Z dx = r_x - p_x;
289       Z dy = r_y - p_y;
290       Z lhs = precomputed_dx * dy;
291       Z rhs = dx * precomputed_dy;
292       return lhs > rhs;
293     }
294
295   private:
296
297     Z p_x;
298     Z p_y;
299     Z precomputed_dx;
300     Z precomputed_dy;
301   };
302
303   template<typename P>
304   requires is_point<P>
305   class signed_area {
306   public:
```

```
307
308     using T = typename P::coordinate;
309     static constexpr std::size_t w = cphmpl::width<T>;
310     using Z = cphstl::ℤ<2 * w + 5>;
311
312     // Z lhs = (q_x - p_x) * (r_y - p_y);
313     // Z rhs = (r_x - p_x) * (q_y - p_y);
314
315     signed_area(P const& p, P const& q)
316       : p_x(p.x), p_y(p.y), precomputed_dx(Z(q.x) - Z(p.x)),
317         precomputed_dy(Z(q.y) - Z(p.y)) {
318     }
319
320     Z operator()(P const& r) {
321
322 #ifdef MEASURE_TURNS
323
324     ++turns;
325
326 #endif
327
328       Z r_x(r.x);
329       Z r_y(r.y);
330       Z dx = r_x - p_x;
331       Z dy = r_y - p_y;
332       Z lhs = precomputed_dx * dy;
333       Z rhs = dx * precomputed_dy;
334       return lhs - rhs;
335     }
336
337   private:
338
339     Z p_x;
340     Z p_y;
341     Z precomputed_dx;
342     Z precomputed_dy;
343   };
344 }
345
346 #endif
347
348 #if defined(DOUBLE_PRECISION)
349
350 namespace double_precision {
351
352   template<typename P>
353   requires is_point<P>
354   bool left_turn(P const& p, P const& q, P const& r) {
355
356 #ifdef MEASURE_TURNS
357
358     ++turns;
```

```
359
360 #endif
361
362     using Z = long long int;
363     using U = unsigned long long int;
364     Z p_x = Z(p.x);
365     Z p_y = Z(p.y);
366     Z q_x = Z(q.x);
367     Z q_y = Z(q.y);
368     Z r_x = Z(r.x);
369     Z r_y = Z(r.y);
370     Z dx1 = q_x - p_x;
371     Z dx2 = r_x - q_x;
372     Z dy1 = q_y - p_y;
373     Z dy2 = r_y - q_y;
374
375     enum {non_negative = 0, negative = 1};
376     bool dx1_sign = dx1 < Z();
377     bool dy2_sign = dy2 < Z();
378     bool dx2_sign = dx2 < Z();
379     bool dy1_sign = dy1 < Z();
380
381     bool lhs_sign = dx1_sign != dy2_sign;
382     bool rhs_sign = dx2_sign != dy1_sign;
383
384     if (dx1 == Z() or dy2 == Z()) {
385       if (dx2 == Z() or dy1 == Z()) {
386         return false;
387       }
388       return rhs_sign;
389     }
390     if (lhs_sign != rhs_sign) {
391       return rhs_sign;
392     }
393     else {
394       U ux1 = dx1_sign == negative ? U(-dx1) : U(dx1);
395       U uy2 = dy2_sign == negative ? U(-dy2) : U(dy2);
396       U ux2 = dx2_sign == negative ? U(-dx2) : U(dx2);
397       U uy1 = dy1_sign == negative ? U(-dy1) : U(dy1);
398       U lhs = ux1 * uy2;
399       U rhs = ux2 * uy1;
400       if (lhs_sign == negative) {
401         return lhs < rhs;
402       }
403       else {
404         return lhs > rhs;
405       }
406     }
407   }
408
409   template<typename P>
410   requires is_point<P>
```

```
411   bool no_turn(P const& p, P const& q, P const& r) {
412
413 #ifdef MEASURE_TURNS
414
415     ++turns;
416
417 #endif
418
419     using T = decltype(p.x);
420     using Z = long long;
421     Z p_x = Z(p.x);
422     Z p_y = Z(p.y);
423     Z q_x = Z(q.x);
424     Z q_y = Z(q.y);
425     Z r_x = Z(r.x);
426     Z r_y = Z(r.y);
427     Z dx_1 = q_x - p_x;
428     Z dx_2 = r_x - q_x;
429     Z dy_1 = q_y - p_y;
430     Z dy_2 = r_y - q_y;
431
432     bool check1 = Z(T(dx_1)) != dx_1 and Z(T(dy_2)) != dy_2;
433     bool check2 = Z(T(dx_2)) != dx_2 and Z(T(dy_1)) != dy_1;
434     if (check1 or check2) {
435       // if they do not both overflow, they must be different
436       if (check1 and check2) {
437         // if overflow direction not the same, they are different
438         if (((dx_1 > 0) == (dy_2 > 0)) == ((dx_2 > 0) == (dy_1 >
    ↪ 0))) {
439           // the non-overflow part must also be the same
440           return dx_1 * dy_2 == dx_2 * dy_1;
441         }
442       }
443       return false;
444     }
445     else {
446       return dx_1 * dy_2 == dx_2 * dy_1;
447     }
448   }
449
450   // may do controlled overflow on very large values
451
452   template<typename P>
453   requires is_point<P>
454   auto signed_area(P const& p, P const& q, P const& r) {
455     assert(not right_turn(p, q, r));
456
457 #ifdef MEASURE_TURNS
458
459     ++turns;
460
461 #endif
```

```
462
463       using Z = long long;
464       using U = unsigned long long;
465       Z p_x = Z(p.x);
466       Z p_y = Z(p.y);
467       Z q_x = Z(q.x);
468       Z q_y = Z(q.y);
469       Z r_x = Z(r.x);
470       Z r_y = Z(r.y);
471       Z dx_1 = q_x - p_x;
472       Z dx_2 = r_x - q_x;
473       Z dy_1 = q_y - p_y;
474       Z dy_2 = r_y - q_y;
475       return (U) (dx_1 * dy_2 - dx_2 * dy_1);
476     }
477   }
478
479   #endif
480
481   #if defined(FLOATING_POINT_FILTER)
482
483   namespace floating_point_filter {
484
485     template<typename P>
486     requires is_point<P>
487     bool left_turn(P const& p, P const& q, P const& r) {
488
489   #ifdef MEASURE_TURNS
490
491       ++turns;
492
493   #endif
494
495       using R = double;
496       constexpr R u = std::numeric_limits<R>::epsilon();
497       R p_x = R(p.x);
498       R p_y = R(p.y);
499       R q_x = R(q.x);
500       R q_y = R(q.y);
501       R r_x = R(r.x);
502       R r_y = R(r.y);
503       R lhs = (q_x - p_x) * (r_y - p_y);
504       R rhs = (r_x - p_x) * (q_y - p_y);
505       R det = lhs - rhs;
506       R detsum = 0.0;
507       if (lhs > 0.0) {
508         if (rhs <= 0.0) {
509           return lhs > rhs;
510         }
511         else {
512           detsum = lhs + rhs;
513         }
```

```
514       }
515       else if (lhs < 0.0) {
516         if (rhs ≥ 0.0) {
517           return lhs > rhs;
518         }
519         else {
520           detsum = -lhs - rhs;
521         }
522       }
523       else {
524         return lhs > rhs;
525       }
526       R errbound = (3 * u + 16 * u * u) * detsum;
527       if (det ≥ errbound or -det ≥ errbound) {
528         return lhs > rhs;
529       }
530       return multiple_precision::left_turn(p, q, r);
531     }
532
533     template<typename P>
534     requires is_point<P>
535     bool no_turn(P const& p, P const& q, P const& r) {
536
537   #ifdef MEASURE_TURNS
538
539       ++turns;
540
541   #endif
542
543       using R = double;
544       constexpr R u = std::numeric_limits<R>::epsilon();
545       R p_x = R(p.x);
546       R p_y = R(p.y);
547       R q_x = R(q.x);
548       R q_y = R(q.y);
549       R r_x = R(r.x);
550       R r_y = R(r.y);
551       R lhs = (q_x - p_x) * (r_y - p_y);
552       R rhs = (r_x - p_x) * (q_y - p_y);
553       R det = lhs - rhs;
554       R detsum = 0.0;
555       if (lhs > 0.0) {
556         if (rhs ≤ 0.0) {
557           return false;
558         }
559         else {
560           detsum = lhs + rhs;
561         }
562       }
563       else if (lhs < 0.0) {
564         if (rhs ≥ 0.0) {
565           return false;
```

```
566         }
567         else {
568           detsum = -lhs - rhs;
569         }
570       }
571       else {
572         return lhs == rhs;
573       }
574       R errbound = (3 * u + 16 * u * u) * detsum;
575       if (det >= errbound or -det >= errbound) {
576         return lhs == rhs;
577       }
578       return multiple_precision::no_turn(p, q, r);
579     }
580
581     template<typename P>
582     requires is_point<P>
583     auto signed_area(P const& p, P const& q, P const& r) {
584       return multiple_precision::signed_area(p, q, r);
585     }
586   }
587
588   #endif
589
590   #if defined(MULTIPLE_PRECISION)
591
592   using namespace multiple_precision;
593
594   #elif defined(DOUBLE_PECISION)
595
596   using namespace double_precision;
597
598   #elif defined(FLOATING_POINT_FILTER)
599
600   using namespace floating_point_filter;
601
602   #elif defined(PARTIALLY_EVALUATED)
603
604   using namespace multiple_precision;
605
606   #endif
607
608   template<typename P>
609   requires is_point<P>
610   bool right_turn(P const& p, P const& q, P const& r) {
611     return left_turn(r, q, p);
612   }
613
614   // is p on the line segment {q, r}?
615
616   template<typename P>
617   requires is_point<P>
```

```
618 bool on_line_segment(P const& p, P const& q, P const& r) {
619   P left = std::min(q, r);
620   P right = std::max(q, r);
621   if (p < left) {
622     return false;
623   }
624   if (p > right) {
625     return false;
626   }
627   return no_turn(p, q, r);
628 }
629
630 // is p lexicographically smaller than q with respect to the given
      ↪ ordering?
631
632 template<typename C, typename T>
633 requires cphmpl::is_binary_predicate<C, T, T>
634 class is_lexicographically_smaller {
635 public:
636
637   is_lexicographically_smaller(C comparator)
638     : less(comparator) {
639   }
640
641   template<typename P>
642   requires is_point<P>
643   bool operator()(P const& p, P const& q) {
644     return less(p.x, q.x) or (p.x == q.x and less(p.y, q.y));
645   }
646
647 private:
648
649   C less;
650 };
651
652 #endif
```

D.2 validation.h++

```
1  #ifndef __VALIDATION__
2  #define __VALIDATION__
3
4  #include <algorithm> // std::copy std::sort std::equal std::rotate
5  #include <cassert> // assert macro
6  #include <cstdlib> // std::size_t
7  #include <iostream> // std streams
8  #include <iterator> // std::iterator_traits
9  #include "point.h++" // left_turn right_turn
10 #include <vector> // std::vector
11
12 namespace validation {
```

```
13
14  template<typename I, typename J>
15  bool same_multiset(I p, I q, J r, J s) {
16    using P = typename std::iterator_traits<I>::value_type;
17    using S = std::vector<P>;
18    std::size_t n = q - p;
19    std::size_t m = s - r;
20    if (m != n) {
21      return false;
22    }
23    S backup;
24    backup.resize(n);
25    std::copy(p, q, backup.begin());
26    std::sort(backup.begin(), backup.end(),
27      [](P const& a, P const& b) -> bool {
28        return (a.x < b.x) or (a.x == b.x and a.y < b.y);
29      });
30    S other;
31    other.resize(n);
32    std::copy(r, s, other.begin());
33    std::sort(other.begin(), other.end(),
34      [](P const& a, P const& b) -> bool {
35        return (a.x < b.x) or (a.x == b.x and a.y < b.y);
36      });
37    return std::equal(backup.begin(), backup.end(), other.begin(),
38      [](P const& a, P const& b) -> bool {
39        return (a.x == b.x) and (a.y == b.y);
40      });
41  }
42
43  // [p, r) circular chain of vertices
44
45  template<typename I>
46  bool left_turns_only(I p, I r) {
47    std::size_t h = r - p;
48    if (h < 3) {
49      return true;
50    }
51    for (I q = p; q != r; ++q) {
52      I next = q + 1;
53      I next_after = next + 1;
54      if (q == r - 2) {
55        next_after = p;
56      }
57      else if (q == r - 1) {
58        next = p;
59        next_after = p + 1;
60      }
61      if (not left_turn(*q, *next, *next_after)) {
62        std::cerr << "Left-turn: " << *q << " " << *next << " "
63                  << *next_after << " failed\n";
64        return false;
```

```
65        }
66      }
67      return true;
68    }
69
70    // [p, r) circular chain of vertices
71
72    template<typename I>
73    bool right_turns_only(I p, I r) {
74      std::size_t h = r - p;
75      if (h < 3) {
76        return true;
77      }
78      for (I q = p; q != r; ++q) {
79        I next = q + 1;
80        I next_after = next + 1;
81        if (q == r - 2) {
82          next_after = p;
83        }
84        else if (q == r - 1) {
85          next = p;
86          next_after = p + 1;
87        }
88        if (not right_turn(*q, *next, *next_after)) {
89          std::cerr << "Right-turn: " << *q << " " << *next << " "
90                    << *next_after << " failed\n";
91          return false;
92        }
93      }
94      return true;
95    }
96
97    // [p, r) half-circular chain of vertices
98
99    template<typename I>
100   bool monotone(I p, I r) {
101     assert(p != r);
102     using P = typename std::iterator_traits<I>::value_type;
103     std::size_t h = r - p;
104     if (h == 0 or h == 1) {
105       return true;
106     }
107     I q = p;
108     while (q != r - 1) {
109       P a = *q;
110       P b = *(q + 1);
111       if (not (a.x < b.x or (a.x == b.x and a.y < b.y))) {
112         std::cerr << "Convex: " << a << " " << b
113                   << ": not monotone\n";
114         return false;
115       }
116       ++q;
```

```
117       }
118       return true;
119     }
120
121     // [p, r) circular chain of vertices
122
123     template<typename I>
124     bool convex_polygon(I p, I r) {
125       using P = typename std::iterator_traits<I>::value_type;
126       using S = std::vector<P>;
127       using J = typename S::iterator;
128       std::size_t h = r - p;
129       if (h == 0 or h == 1) {
130         return true;
131       }
132       if (h == 2) {
133         if (*p != *(p + 1)) {
134           return true;
135         }
136         else {
137           std::cerr << "Convex: h = 2: two equal points\n";
138           return false;
139         }
140       }
141       // h ≥ 3
142       bool clockwise = right_turn(*p, *(p + 1), *(p + 2));
143       if (clockwise and (not right_turns_only(p, r))) {
144         std::cerr << "Convex: h = " << h
145                   << ": not a spiral (cw)\n";
146         return false;
147       }
148       if (not clockwise and (not left_turns_only(p, r))) {
149         std::cerr << "Convex: h = " << h
150                   << ": not a spiral (ccw)\n";
151         return false;
152       }
153
154       using P = typename std::iterator_traits<I>::value_type;
155       using S = std::vector<P>;
156       using J = typename S::iterator;
157       S hull;
158       hull.resize(h);
159       std::copy(p, r, hull.begin());
160       J west = std::min_element(hull.begin(), hull.end(),
161         [](P const& a, P const& b) -> bool {
162           return (a.x < b.x) or (a.x == b.x and a.y < b.y);
163         });
164       (void) std::rotate(hull.begin(), west, hull.end());
165       hull.push_back(*west);
166       west = hull.begin();
167       J east = std::max_element(hull.begin(), hull.end() - 1,
168         [](P const& a, P const& b) -> bool {
```

```
169         return (a.x < b.x) or (a.x == b.x and a.y < b.y);
170       });
171     assert(west != east);
172     if (clockwise) {
173       if (not monotone(west, east + 1)) {
174         std::cerr << "Convex: h = " << h
175                   << ": upper hull not monotone (cw)\n";
176         return false;
177       }
178       std::reverse(east, hull.end());
179       if (not monotone(east, hull.end())) {
180         std::cerr << "Convex: h = " << h
181                   << ": lower hull not monotone (cw)\n";
182         return false;
183       }
184       return true;
185     }
186     // counterclockwise
187     if (not monotone(west, east + 1)) {
188       std::cerr << "Convex: h = " << h
189                 << ": lower hull not monotone (ccw)\n";
190       return false;
191     }
192     std::reverse(east, hull.end());
193     if (not monotone(east, hull.end())) {
194       std::cerr << "Convex: h = " << h
195                 << ": upper hull not monotone (ccw)\n";
196       return false;
197     }
198     return true;
199   }
200
201   // [p, r) upper hull from left to right
202
203   template<typename I, typename P>
204   bool above(I p, I r, P const& u) {
205     std::size_t h = r - p;
206     assert(h > 1);
207     I last = r - 1;
208     assert(u.x ≥ (*p).x and u.x ≤ (*last).x);
209     if ((*p).x == (*(p + 1)).x) {
210       ++p;
211     }
212     if (h != 1 and (*last).x == (*(last - 1)).x) {
213       --r;
214     }
215     I q = std::lower_bound(p, r, u,
216       [](P const& a, P const& b) → bool {
217         return (a.x < b.x);
218       });
219     if (q == p) {
220       return u.y > (*p).y;
```

```
221     }
222     return left_turn(*(q - 1), *q, u);
223   }
224
225   // [p, r) lower hull from left to right
226
227   template<typename I, typename P>
228   bool below(I p, I r, P const& u) {
229     std::size_t h = r - p;
230     assert(h > 1);
231     I last = r - 1;
232     assert(u.x ≥ (*p).x and u.x ≤ (*last).x);
233     if ((*p).x == (*(p + 1)).x) {
234       ++p;
235     }
236     if (h ≠ 1 and (*last).x == (*(last - 1)).x) {
237       --r;
238     }
239     I q = std::lower_bound(p, r, u,
240       [](P const& a, P const& b) → bool {
241         return (a.x < b.x);
242       });
243     if (q == p) {
244       return u.y < (*p).y;
245     }
246     return right_turn(*(q - 1), *q, u);
247   }
248
249   // [r, s) multiset of points; [p, q) convex polygon
250
251   template<typename I, typename J>
252   bool all_inside(I r, I s, J p, J q) {
253     std::size_t h = q - p;
254     if (h == 0) {
255       if (r ≠ s) {
256         std::cerr << "Inside: h = 0: no points can be inside\n";
257         return false;
258       }
259       return true;
260     }
261     if (h == 1) {
262       for (J i = r; i ≠ s; ++i) {
263         if (*i ≠ *p) {
264           std::cerr << "Inside: h = 1: all points not equal\n";
265           return false;
266         }
267       }
268       return true;
269     }
270     if (h == 2) {
271       for (J i = r; i ≠ s; ++i) {
272         if (not on_line_segment(*i, *p, *(p + 1))) {
```

```
273           std::cerr << "Inside: h = 2: all points not collinear\n";
274           return false;
275         }
276       }
277       return true;
278     }
279     using P = typename std::iterator_traits<I>::value_type;
280     using S = std::vector<P>;
281     using K = typename S::iterator;
282     S hull;
283     hull.resize(h);
284     std::copy(p, q, hull.begin());
285     K west = std::min_element(hull.begin(), hull.end(),
286       [](P const& a, P const& b) -> bool {
287         return (a.x < b.x) or (a.x == b.x and a.y < b.y);
288       });
289     (void) std::rotate(hull.begin(), west, hull.end());
290     hull.push_back(*west);
291     west = hull.begin();
292     K east = std::max_element(hull.begin(), hull.end() - 1,
293       [](P const& a, P const& b) -> bool {
294         return (a.x < b.x) or (a.x == b.x and a.y < b.y);
295       });
296     assert(west != east);
297     for (I i = r; i != s; ++i) {
298       if ((*i).x < (*west).x) {
299         std::cerr << "Inside: " << *i
300                   << " on left of the output\n";
301         return false;
302       }
303       if ((*i).x > (*east).x) {
304         std::cerr << "Inside: " << *i
305                   << " on right of the output\n";
306         return false;
307       }
308     }
309     bool clockwise = right_turn(*west, *(west + 1), *(west + 2));
310     if (clockwise) {
311       for (I i = r; i != s; ++i) {
312         if (above(west, east + 1, *i)) {
313           std::cerr << "Inside: " << *i
314                     << " above the upper hull (cw)\n";
315           return false;
316         }
317       }
318       std::reverse(east, hull.end());
319       for (I i = r; i != s; ++i) {
320         if (below(east, hull.end(), *i)) {
321           std::cerr << "Inside: " << *i
322                     << " below the lower hull (cw)\n";
323           return false;
324         }
```

```
325         }
326       }
327       else { // counterclockwise
328         for (I i = r; i != s; ++i) {
329           if (below(west, east + 1, *i)) {
330             std::cerr << "Inside: " << *i
331                       << " below the lower hull (ccw)\n";
332             return false;
333           }
334         }
335         std::reverse(east, hull.end());
336         for (I i = r; i != s; ++i) {
337           if (above(east, hull.end(), *i)) {
338             std::cerr << "Inside: " << *i
339                       << " above the upper hull (ccw)\n";
340             return false;
341           }
342         }
343       }
344       return true;
345     }
346   }
347
348   #endif
```

E. Drivers

E.1 check-driver.c++

```
#include "algorithm.h++" // NAME::solve NAME::check
#include <cassert> // assert macro
#include "cphstl/ranges.h++" // cphstl ranges
#include <iterator> // std::distance
#include "point.h++" // point
#include <vector> // std::vector

int main() {
  using P = point<int>;
  using S = std::vector<P>;
  using I = typename S::iterator;

  // iterator interface

  S bag{P(0, 0), P(0, 1), P(0, 2), P(0, 1)};
  I rest = NAME::solve(bag.begin(), bag.end());
  auto h = std::distance(bag.begin(), rest);
  assert(h == 2);
  assert(NAME::check(bag.begin(), bag.end()));

  // range interface

  auto whole = cphstl::range(std::begin(bag), std::end(bag));
  I eliminated = NAME::solve(whole);
  h = std::distance(bag.begin(), eliminated);
  assert(h == 2);
  assert(NAME::check(whole));
}
```

E.2 test-driver.c++

```
/*
  Performance Engineering Laboratory © 2017–2018

  Brownie points:
  11 Sep 2017: The test cases for overflow detection by
    ↪ Marius–Florin Cristian
*/

#include <algorithm> // std::copy std::equal
#include <cassert> // assert macro
#include <cstdlib> // std::rand std::size_t
#include <exception> // std::exception
#include <iostream> // std::cout std::cerr
#include <iterator> // std::iterator_traits
#include <limits> // std::numeric_limits
#include <random> // std::linear_congruential_engine
```

```
16 #include <vector> // std::vector
17
18 #define DOUBLE_PRECISION
19 #include "algorithm.h++" // NAME::check
20 #include "point.h++" // point
21
22 using linear_congruential_engine =
       ↪ std::linear_congruential_engine<unsigned long long,
       ↪ 6364136223846793005U, 1442695040888963407U, 0U>;
23 unsigned long long const seed = 10164167376618180197U;
24 linear_congruential_engine random_number_generator{seed};
25
26 template<typename I>
27 using checker = bool (*)(I, I);
28
29 template<typename A>
30 using testcase = void (*)(A);
31
32 template<typename I>
33 void generate(I p, I r) {
34   using P = typename std::iterator_traits<I>::value_type;
35   using T = typename P::coordinate;
36   T t = 100;
37   std::uniform_int_distribution<T> distribution(-t, t);
38   for (I q = p; q != r; ++q) {
39     T x = distribution(random_number_generator);
40     T y = distribution(random_number_generator);
41     *q = P(x, y);
42   }
43 }
44
45 template<typename checker>
46 void empty_set(checker f) {
47   std::cout << "empty set\n";
48   using P = point<int>;
49   using S = std::vector<P>;
50   S input;
51   assert(f(input.begin(), input.end()));
52 }
53
54 template<typename checker>
55 void one_point(checker f) {
56   std::cout << "one point\n";
57   using P = point<int>;
58   using S = std::vector<P>;
59   S input;
60   P origo;
61   input.push_back(origo);
62   assert(f(input.begin(), input.end()));
63 }
64
65 template<typename checker>
```

```
66  void two_points(checker f) {
67    std::cout << "two points\n";
68    using P = point<int>;
69    using S = std::vector<P>;
70    S input;
71    P origo;
72    P another(1, 1);
73    input.push_back(origo);
74    input.push_back(another);
75    assert(f(input.begin(), input.end()));
76  }
77
78  template<typename checker>
79  void three_points_on_a_vertical_line(checker f) {
80    std::cout << "three points on a vertical line\n";
81    using P = point<int>;
82    P origo;
83    P st(0, 1);
84    P nd(0, 2);
85    using S = std::vector<P>;
86    S input;
87    input.push_back(origo);
88    input.push_back(st);
89    input.push_back(nd);
90    assert(f(input.begin(), input.end()));
91  }
92
93  template<typename checker>
94  void three_points_on_a_horizontal_line(checker f) {
95    std::cout << "three points on a horizontal line\n";
96    using P = point<int>;
97    P origo;
98    P up(1, 0);
99    P top(2, 0);
100   using S = std::vector<P>;
101   S input;
102   input.push_back(origo);
103   input.push_back(up);
104   input.push_back(top);
105   assert(f(input.begin(), input.end()));
106 }
107
108 template<typename checker>
109 void ten_equal_points(checker f) {
110   std::cout << "ten equal points\n";
111   using P = point<int>;
112   P p(1, 1);
113   using S = std::vector<P>;
114   S input;
115   for (std::size_t i = 0; i != 10; ++i) {
116     input.push_back(p);
117   }
```

```
118   assert(f(input.begin(), input.end()));
119 }
120
121 template<typename checker>
122 void four_poles_and_many_duplicates_inside(checker f) {
123   std::cout << "four poles and many duplicates inside\n";
124   using P = point<int>;
125   using S = std::vector<P>;
126   std::size_t n = 10;
127   P origin(0, 0);
128   P west(-1, 0);
129   P north(0, 1);
130   P east(1, 0);
131   P south(0, -1);
132   S input;
133   input.push_back(west);
134   input.push_back(north);
135   input.push_back(east);
136   input.push_back(south);
137   for (std::size_t i = 0; i != n; ++i) {
138     input.push_back(origin);
139   }
140   assert(f(input.begin(), input.end()));
141 }
142
143 template<typename checker>
144 void four_poles_with_duplicates(checker f) {
145   std::cout << "four poles with duplicates\n";
146   using P = point<int>;
147   P west(-1, 0);
148   P north(0, 1);
149   P east(1, 0);
150   P south(0, -1);
151   using S = std::vector<P>;
152   S input;
153   for (std::size_t i = 0; i < 5; ++i) {
154     input.push_back(south);
155     input.push_back(north);
156     input.push_back(west);
157     input.push_back(east);
158   }
159   assert(f(input.begin(), input.end()));
160 }
161
162 template<typename checker>
163 void quadrilateral_with_duplicates(checker f) {
164   std::cout << "quadrilateral with duplicates\n";
165   using P = point<int>;
166   P bottom_left(0, 0);
167   P top_left(1, 1);
168   P bottom_right(4, 0);
169   P top_right(3, 1);
```

```
170     using S = std::vector<P>;
171     S input;
172     for (std::size_t i = 0; i < 5; ++i) {
173       input.push_back(bottom_left);
174       input.push_back(top_left);
175       input.push_back(bottom_right);
176       input.push_back(top_right);
177     }
178     assert(f(input.begin(), input.end()));
179   }
180
181   template<typename checker>
182   void duplicates_on_the_periphery(checker f) {
183     std::cout << "duplicates on the periphery\n";
184     using P = point<int>;
185     using S = std::vector<P>;
186     S input;
187     for (std::size_t i = 0; i < 2; ++i) {
188       input.push_back(P(0, 0));
189       input.push_back(P(1, 1));
190       input.push_back(P(2, 2));
191       input.push_back(P(3, 2));
192       input.push_back(P(3, 0));
193       input.push_back(P(4, 2));
194       input.push_back(P(4, 0));
195       input.push_back(P(5, 2));
196       input.push_back(P(5, 0));
197       input.push_back(P(6, 2));
198       input.push_back(P(6, 0));
199       input.push_back(P(7, 1));
200       input.push_back(P(8, 0));
201     }
202     assert(f(input.begin(), input.end()));
203   }
204
205   template<typename checker>
206   void many_east_pole_candidates(checker f) {
207     std::cout << "many east-pole candidates\n";
208     using P = point<int>;
209     P west(0, 0);
210     P east(1, 4);
211     P three(1, 3);
212     P two(1, 2);
213     P one(1, 1);
214     P zero(1, 0);
215     using S = std::vector<P>;
216     S input;
217     for (std::size_t i = 0; i < 5; ++i) {
218       input.push_back(west);
219       input.push_back(east);
220       input.push_back(three);
221       input.push_back(two);
```

```
222       input.push_back(zero);
223       input.push_back(one);
224     }
225     assert(f(input.begin(), input.end()));
226   }
227
228   template<typename checker>
229   void line_quadrilateral_line(checker f) {
230     std::cout << "line quadrilateral line\n";
231     using P = point<int>;
232     using S = std::vector<P>;
233     S input;
234     input.push_back(P(0, 0));
235     input.push_back(P(1, 0));
236     input.push_back(P(2, 0));
237     input.push_back(P(2, 0));
238     input.push_back(P(3, 0));
239
240     input.push_back(P(4, 1));
241     input.push_back(P(5, 2));
242     input.push_back(P(6, 3));
243     input.push_back(P(6, 3));
244     input.push_back(P(7, 2));
245     input.push_back(P(8, 1));
246
247     input.push_back(P(4, -1));
248     input.push_back(P(5, -2));
249     input.push_back(P(6, -3));
250     input.push_back(P(6, -3));
251     input.push_back(P(7, -2));
252     input.push_back(P(8, -1));
253
254     input.push_back(P(9, 0));
255     input.push_back(P(10, 0));
256     input.push_back(P(11, 0));
257     input.push_back(P(12, 0));
258     assert(f(input.begin(), input.end()));
259   }
260
261   template<typename checker>
262   void many_big_numbers(checker f) {
263     std::cout << "many big numbers\n";
264     using T = int;
265     using P = point<int>;
266     T max = std::numeric_limits<T>::max();
267     P origo(0, 0);
268     P left_bottom(-max, -max);
269     P just_above(-max, -max + 1);
270     P right_top(max, max);
271     P left_top(-max, max);
272     P neighbour_below(-max, max - 1);
273     P neighbour_beside(-max - 1, max);
```

```
274   using S = std::vector<P>;
275   S input;
276   input.push_back(left_bottom);
277   input.push_back(just_above);
278   input.push_back(origo);
279   input.push_back(left_top);
280   input.push_back(right_top);
281   input.push_back(neighbour_below);
282   input.push_back(neighbour_beside);
283   assert(f(input.begin(), input.end()));
284 }
285
286 template<typename checker>
287 void noisy_box(checker f) {
288   std::cout << "noisy box\n";
289   using T = int;
290   using P = point<T>;
291   T max = std::numeric_limits<T>::max();
292   using S = std::vector<P>;
293   using I = typename S::iterator;
294   constexpr std::size_t n = 10;
295   S input(n);
296   std::uniform_int_distribution<T> distribution(-max + 1, max - 1);
297   for (I q = input.begin(); q != input.end(); ++q) {
298     T x = distribution(random_number_generator);
299     T y = distribution(random_number_generator);
300     *q = P(x, y);
301   }
302   input.push_back(P(max, max));
303   input.push_back(P(max, -max));
304   input.push_back(P(-max, -max));
305   input.push_back(P(-max, max));
306   assert(f(input.begin(), input.end()));
307 }
308
309 template<typename checker>
310 void random_points_on_a_parabola(checker f) {
311   std::cout << "random points on a parabola\n";
312   using P = point<int>;
313   using S = std::vector<P>;
314   using I = typename S::iterator;
315   std::size_t const n = 4;
316   S input(n);
317   int j = 0;
318   for (I q = input.begin(); q != input.end(); ++q) {
319     *q = P(j, j * j);
320     ++j;
321   }
322   std::shuffle(input.begin(), input.end(), random_number_generator);
323   assert(f(input.begin(), input.end()));
324 }
325
```

```
326 template<typename checker>
327 void random_points_in_a_square(checker f) {
328   std::cout << "random points in a square\n";
329   std::size_t const bign = 10000;
330   for (std::size_t n = 0; n ≤ bign; ++n) {
331     if (n % 100 == 0) {
332       std::cout << "." << std::flush;
333     }
334     using P = point<int>;
335     using S = std::vector<P>;
336     S input(n);
337     generate(input.begin(), input.end());
338     assert(f(input.begin(), input.end()));
339   }
340   std::cout << "\n" << std::flush;
341 }
342
343 template<typename checker>
344 void positive_overflow(checker f) {
345   std::cout << "positive overflow\n";
346   using T = int;
347   using P = point<T>;
348   T max = std::numeric_limits<T>::max();
349   P origo(100, 100);
350   P left_top(100, max);
351   P right_top(max, max);
352   P middle(100000, 100000);
353   P right_bottom(max, 100);
354   using S = std::vector<P>;
355   S input;
356   input.push_back(left_top);
357   input.push_back(right_bottom);
358   input.push_back(origo);
359   input.push_back(middle);
360   input.push_back(right_top);
361   assert(f(input.begin(), input.end()));
362 }
363
364 template<typename checker>
365 void negative_overflow(checker f) {
366   std::cout << "negative overflow\n";
367   using T = int;
368   using P = point<T>;
369   T min = std::numeric_limits<T>::min();
370   P origo;
371   P left_top(min, 0);
372   P left_bottom(min, min);
373   P right_bottom(0, min);
374   P middle(–10000, –10000);
375   using S = std::vector<P>;
376   S input;
377   input.push_back(left_top);
```

```
378   input.push_back(right_bottom);
379   input.push_back(origo);
380   input.push_back(middle);
381   input.push_back(left_bottom);
382   assert(f(input.begin(), input.end()));
383 }
384
385 template<typename checker>
386 void corners_of_the_universe(checker f) {
387   std::cout << "corners of the universe\n";
388   using T = int;
389   using P = point<T>;
390   T min = std::numeric_limits<T>::min();
391   T max = std::numeric_limits<T>::max();
392   P origo;
393   P left_top(min, max);
394   P left_bottom(min, min);
395   P right_top(max, max);
396   P right_bottom(max, min);
397   using S = std::vector<P>;
398   S input;
399   input.push_back(left_top);
400   input.push_back(right_bottom);
401   input.push_back(origo);
402   input.push_back(right_top);
403   input.push_back(left_bottom);
404   assert(f(input.begin(), input.end()));
405 }
406
407 template<typename checker>
408 void negative_coordinates(checker f) {
409   std::cout << "negative coordinates\n";
410   using P = point<int>;
411   P p1(-3, -3);
412   P p2(-3, -2);
413   P p3(0, 0);
414   P p4(1, -1);
415   using S = std::vector<P>;
416   S input;
417   input.push_back(p1);
418   input.push_back(p2);
419   input.push_back(p3);
420   input.push_back(p4);
421   assert(f(input.begin(), input.end()));
422 }
423
424 template<typename checker>
425 void degenerate_coordinates(checker f) {
426   std::cout << "degenerate coordinates\n";
427   using P = point<int>;
428   P p1(0, 0);
429   P p2(1, 0);
```

```
430   P p3(2, 0);
431   P p4(3, 0);
432   P p5(3, 1);
433   P p6(3, 2);
434   P p7(3, 3);
435   P p8(2, 3);
436   P p9(1, 3);
437   P p10(0, 3);
438   using S = std::vector<P>;
439   S input;
440   input.push_back(p1);
441   input.push_back(p2);
442   input.push_back(p3);
443   input.push_back(p4);
444   input.push_back(p5);
445   input.push_back(p6);
446   input.push_back(p7);
447   input.push_back(p8);
448   input.push_back(p9);
449   input.push_back(p10);
450   assert(f(input.begin(), input.end()));
451 }
452
453 int main() {
454   using P = point<int>;
455   using S = std::vector<P>;
456   using I = typename S::iterator;
457   using R = checker<I>;
458   using T = testcase<R>;
459
460   T suite[] = {
461     empty_set,
462     one_point,
463     two_points,
464     three_points_on_a_vertical_line,
465     three_points_on_a_horizontal_line,
466     ten_equal_points,
467     four_poles_and_many_duplicates_inside,
468     four_poles_with_duplicates,
469     quadrilateral_with_duplicates,
470     duplicates_on_the_periphery,
471     many_east_pole_candidates,
472     line_quadrilateral_line,
473     many_big_numbers,
474     noisy_box,
475     random_points_on_a_parabola,
476     random_points_in_a_square,
477     positive_overflow,
478     negative_overflow,
479     corners_of_the_universe,
480     negative_coordinates,
481     degenerate_coordinates
```

```
482     };
483
484     R program = NAME::check;
485     for (T test: suite) {
486       try {
487         test(program);
488       }
489       catch(std::exception& e) {
490         std::cerr << e.what() << std::endl;
491       }
492     }
493     return 0;
494   }
```

E.3 driver.c++

```
1   /*
2     Performance Engineering Laboratory © 2017–2019
3   */
4
5   unsigned long maxsize = 128 * 1024 * 1024; // 32 for PARABOLA
6   // 1 << 30 = 1073741824 there is space, but things become very slow
7
8   #if defined(MEASURE_ARITHMETIC)
9
10  unsigned long long constructions = 0;
11  unsigned long long comparisons = 0;
12  unsigned long long additions = 0;
13  unsigned long long subtractions = 0;
14  unsigned long long multiplications = 0;
15
16  #endif
17
18  #if defined(MEASURE_TURNS)
19
20  long long turns = 0;
21
22  #endif
23
24  #if defined(MEASURE_COMPS) or defined(MEASURE_MOVES)
25
26  long long comps = 0;
27  long long moves = 0;
28
29  #endif
30
31  #include <iostream> // std::cout std::cerr
32
33  #include <cassert> // assert macro
34  #include <cstdlib> // std::atoi std::size_t
35  #include <cmath> // std::sqrt
```

```cpp
#include "cphstl/integers.h++" // cphstl integers
#include <ctime> // std::clock_t std::clock CLOCKS_PER_SEC
#include <iterator> // std::iterator_traits std::distance
#include <limits> // std::numeric_limits
#include <random> // std::linear_congruential_engine

#include "algorithm.h++" // NAME::solve

using linear_congruential_engine = std::linear_congruential_engine<
  unsigned long long, 6364136223846793005U, 1442695040888963407U,
    ↪ 0U>;
constexpr unsigned long long seed = 10164167376618180197U;
linear_congruential_engine random_number_generator{seed};

#if defined(DISC)

template<typename I>
void generate(I p, I r) {
  using P = typename std::iterator_traits<I>::value_type;
  using T = typename P::coordinate;
  T t = std::numeric_limits<T>::max();
  std::uniform_int_distribution<T> distribution(-t, t);
  for (I q = p; q != r; ++q) {
    T x = 0;
    T y = 0;
    using R = double;
    R radius = R(t);
    R root = 0.0;
    do {
      x = distribution(random_number_generator);
      y = distribution(random_number_generator);
      R dx = R(x) * R(x);
      R dy = R(y) * R(y);
      root = std::sqrt(dx + dy);
    } while (root > radius);
    *q = P(x, y);
  }
}

#elif defined(UNIVERSE)

template<typename I>
void generate(I p, I r) {
  std::size_t n = r - p;
  using P = typename std::iterator_traits<I>::value_type;
  using T = typename P::coordinate;
  T t = std::sqrt(n);
  std::uniform_int_distribution<T> distribution(-t, +t);
  for (I q = p; q != r; ++q) {
    T x = distribution(random_number_generator);
    T y = distribution(random_number_generator);
    *q = P(x, y);
```

```
87    }
88  }
89
90  #elif defined(SPECIAL)
91
92  template<typename I>
93  void generate(I p, I r) {
94    assert(std::distance(p, r) ≥ 4);
95    using P = typename std::iterator_traits<I>::value_type;
96    P origin(0, 0);
97    P west(−1, 0);
98    P north(0, 1);
99    P east(1, 0);
100   P south(0, −1);
101   *p = west;
102   ++p;
103   *p = north;
104   ++p;
105   *p = east;
106   ++p;
107   *p = south;
108   ++p;
109   for (I q = p; q ≠ r; ++q) {
110     *q = origin;
111   }
112 }
113
114 #elif defined(LINE)
115
116 template<typename I>
117 void generate(I p, I r) {
118   using P = typename std::iterator_traits<I>::value_type;
119   using T = typename P::coordinate;
120   T t = std::numeric_limits<T>::max();
121   std::uniform_int_distribution<T> distribution(−t, t);
122   for (I q = p; q ≠ r; ++q) {
123     T x = distribution(random_number_generator);
124     *q = P(x, x);
125   }
126 }
127
128 #elif defined(PARABOLA)
129
130 template<typename I>
131 void generate(I p, I r) {
132   using P = typename std::iterator_traits<I>::value_type;
133   using T = int; // typename P::coordinate;
134   T t = 1 << 15;
135   std::uniform_int_distribution<T> distribution(−t, t);
136   for (I q = p; q ≠ r; ++q) {
137     T x = distribution(random_number_generator);
138     *q = P(x, x * x);
```

```
139   }
140 }
141
142 #elif defined(SORTED)
143
144 template<typename I>
145 void generate(I p, I r) {
146   using P = typename std::iterator_traits<I>::value_type;
147   using T = int;
148   T min = std::numeric_limits<T>::min();
149   T max = std::numeric_limits<T>::max();
150   std::uniform_int_distribution<T> distribution(min, max);
151   for (I q = p; q != r; ++q) {
152     T x = distribution(random_number_generator);
153     T y = distribution(random_number_generator);
154     *q = P(x, y);
155   }
156   std::sort(p, r,
157     [](P const& p, P const& q) -> bool {
158       return p.x < q.x;
159     });
160 }
161
162 #elif defined(BELL)
163
164 template<typename I>
165 void generate(I p, I r) {
166   using P = typename std::iterator_traits<I>::value_type;
167   using T = typename P::coordinate;
168   using D = typename std::iterator_traits<I>::difference_type;
169   D n = std::distance(p, r);
170   using R = double;
171   R radius = R(std::numeric_limits<T>::max());
172   R mean = 0.0;
173   R stddev = radius / (2 + std::log(n));
174   std::normal_distribution<R> distribution(mean, stddev);
175   for (I q = p; q != r; ++q) {
176     T x = std::round(distribution(random_number_generator));
177     T y = std::round(distribution(random_number_generator));
178     *q = P(x, y);
179   }
180 }
181
182 #else // default SQUARE
183
184 template<typename I>
185 void generate(I p, I r) {
186   using P = typename std::iterator_traits<I>::value_type;
187   using T = int; // typename P::coordinate;
188   T min = std::numeric_limits<T>::min();
189   T max = std::numeric_limits<T>::max();
190   std::uniform_int_distribution<T> distribution(min, max);
```

```
191     for (I q = p; q != r; ++q) {
192       T x = distribution(random_number_generator);
193       T y = distribution(random_number_generator);
194       *q = P(x, y);
195     }
196   }
197
198   #endif
199
200   void usage(char const* program) {
201     std::cerr << "Usage: " << program << " <n>\n";
202     exit(1);
203   }
204
205   #if defined(MEASURE_COMPS) or defined(MEASURE_MOVES)
206
207   template<typename T>
208   class counter {
209   private:
210
211     T datum;
212
213   public:
214
215     static constexpr bool is_signed = true;
216     static constexpr bool is_integer = true;
217     static constexpr bool is_exact = true;
218     static constexpr bool is_bounded = true;
219     static constexpr bool is_modulo = not is_signed;
220     static constexpr std::size_t radix = 2;
221     static constexpr std::size_t length = 1;
222     static constexpr std::size_t width = 8 * sizeof(T);
223     static constexpr T min = std::numeric_limits<T>::min();
224     static constexpr T max = std::numeric_limits<T>::max();
225
226     explicit counter()
227       : datum(0) {
228       moves += 1;
229     }
230
231     counter(int x)
232       : datum(x) {
233       moves += 1;
234     }
235
236     counter(counter const& other) :
237       datum(other.datum) {
238       moves += 1;
239     }
240
241     template<typename number>
242     explicit counter(number x = 0)
```

```
243       : datum(x) {
244       moves += 1;
245     }
246
247     counter(counter&& other) {
248       datum = std::move(other.datum);
249       moves += 1;
250     }
251
252     counter& operator=(counter const& other) {
253       datum = other.datum;
254       moves += 1;
255       return *this;
256     }
257
258     counter& operator=(counter&& other) {
259       datum = std::move(other.datum);
260       moves += 1;
261       return *this;
262     }
263
264     operator T() const {
265       return datum;
266     }
267
268     template<unsigned long int b>
269     requires b ≥ width
270     operator cphstl::ℤ<b>() const {
271       return cphstl::ℤ<b>(datum);
272     }
273
274     template<typename U>
275     friend bool operator==(counter<U> const&, counter<U>
          ↪ const&);
276
277     template<typename U>
278     friend bool operator≠(counter<U> const&, counter<U>
          ↪ const&);
279
280     template<typename U>
281     friend bool operator<(counter<U> const&, counter<U> const&);
282
283     template<typename U>
284     friend bool operator>(counter<U> const&, counter<U> const&);
285
286     template<typename U>
287     friend bool operator≤(counter<U> const&, counter<U> const&);
288
289     template<typename U>
290     friend bool operator≥(counter<U> const&, counter<U> const&);
291   };
292
```

```
293 template<typename T>
294 bool operator==(counter<T> const& x, counter<T> const& y) {
295   ++comps;
296   return x.datum == y.datum;
297 }
298
299 template<typename T>
300 bool operator≠(counter<T> const& x, counter<T> const& y) {
301   ++comps;
302   return x.datum ≠ y.datum;
303 }
304
305 template<typename T>
306 bool operator<(counter<T> const& x, counter<T> const& y) {
307   ++comps;
308   return x.datum < y.datum;
309 }
310
311 template<typename T>
312 bool operator>(counter<T> const& x, counter<T> const& y) {
313   ++comps;
314   return x.datum > y.datum;
315 }
316
317 template<typename T>
318 bool operator≤(counter<T> const& x, counter<T> const& y) {
319   ++comps;
320   return x.datum ≤ y.datum;
321 }
322
323 template<typename T>
324 bool operator≥(counter<T> const& x, counter<T> const& y) {
325   ++comps;
326   return x.datum ≥ y.datum;
327 }
328
329 #endif
330
331 int main(int argc, char** argv) {
332   unsigned long n = 15;
333   if (argc == 2) {
334     n = std::atoi(argv[1]);
335   }
336   else {
337     usage(argv[0]);
338   }
339
340   unsigned long repetitions = maxsize / n;
341   if (n > maxsize) {
342     repetitions = 1;
343     maxsize = repetitions * n;
344   }
```

```
345
346 #if defined(MEASURE_MOVES) or defined(MEASURE_COMPS)
347
348   using P = point<counter<int>>;
349
350 #else
351
352   using P = point<int>;
353
354 #endif
355
356   using I = P*;
357   I a = new P[maxsize];
358   I b = a;
359   for (volatile unsigned long t = 0; t != repetitions; ++t) {
360     generate(b, b + n);
361     b = b + n;
362   }
363
364   b = a;
365
366 #if defined(MEASURE_ARITHMETIC)
367
368   constructions = 0;
369   comparisons = 0;
370   additions = 0;
371   subtractions = 0;
372   multiplications = 0;
373
374 #elif defined(MEASURE_TURNS)
375
376   turns = 0;
377
378 #elif defined(MEASURE_MOVES)
379
380   moves = 0;
381
382 #elif defined(MEASURE_COMPS)
383
384   comps = 0;
385
386 #elif defined(PRUNING)
387
388   pruning_efficiency = 0;
389
390 #else
391
392   std::clock_t start = std::clock();
393
394 #endif
395
396   for (volatile unsigned long t = 0; t != repetitions; ++t)  {
```

```
397       (void) NAME::solve(&b[0], &b[n]);
398       b = b + n;
399     }
400
401   #if defined(MEASURE_ARITHMETIC)
402
403     double t = double(repetitions) * double(n);
404     std::cout.precision(3);
405     std::cout << n << "\n";
406     std::cout << " structors " << double(constructions) / t << "\n";
407     std::cout << " <         " << double(comparisons) / t << "\n";
408     std::cout << " +         " << double(additions) / t << "\n";
409     std::cout << " -         " << double(subtractions) / t << "\n";
410     std::cout << " *         " << double(multiplications) / t <<
        ↪ "\n";
411
412   #elif defined(MEASURE_TURNS)
413
414     double t = double(repetitions) * double(n);
415     std::cout.precision(3);
416     std::cout << n << "\t" << double(turns) / t << "\n";
417
418   #elif defined(MEASURE_MOVES)
419
420     double t = double(repetitions) * double(n);
421     std::cout.precision(3);
422     std::cout << n << "\t" << double(moves) / t << "\n";
423
424   #elif defined(MEASURE_COMPS)
425
426     double t = double(repetitions) * double(n);
427     std::cout.precision(3);
428     std::cout << n << "\t" << double(comps) / t << "\n";
429
430   #elif defined(PRUNING)
431
432     double t = double(repetitions) * double(n);
433     std::cout.precision(3);
434     std::cout << n << "\tpruning: " << double(pruning_efficiency) *
        ↪ 100.0 / t << "\n";
435
436   #else
437
438     std::clock_t stop = std::clock();
439
440     double t = double(repetitions) * double(n);
441     double ns = 1000000000.0 * double(stop - start) /
        ↪ double(CLOCKS_PER_SEC);
442     std::cout.precision(4);
443     std::cout << n << "\t" << ns / t << "\n";
444
445   #endif
```

```
446
447     delete[] a;
448     return 0;
449 }
```

F. Makefile

F.1 makefile

```
CXX=g++-8
CXXFLAGS=-O3 -std=c++2a -Wall -Wextra -x c++ -fconcepts -DNDEBUG
IFLAGS = -I../..

header-files:= $(wildcard *.h++)
implementations:= $(basename $(header-files))
square-tests:= $(addsuffix .square, $(implementations))
disc-tests:= $(addsuffix .disc, $(implementations))
universe-tests:= $(addsuffix .universe, $(implementations))
bell-tests:= $(addsuffix .bell, $(implementations))
special-tests:= $(addsuffix .special, $(implementations))
line-tests:= $(addsuffix .line, $(implementations))
sorted-tests:= $(addsuffix .sorted, $(implementations))
parabola-tests:= $(addsuffix .parabola, $(implementations))
turn-tests:= $(addsuffix .turn, $(implementations))
arithmetic-tests:= $(addsuffix .arithmetic, $(implementations))
comp-tests:= $(addsuffix .comp, $(implementations))
move-tests:= $(addsuffix .move, $(implementations))
elimsquare-tests:= $(addsuffix .elimsquare, $(implementations))
elimdisc-tests:= $(addsuffix .elimdisc, $(implementations))
sanitychecks:= $(addsuffix .check, $(implementations))
unittests:= $(addsuffix .test, $(implementations))
benchmarks:= $(addsuffix .benchmark, $(implementations))

.PHONY: all clean find pilot veryclean

N = 1024 32768 1048576 33554432 # 1073741824 # the last one took 10 minutes

$(square-tests): %.square : %.h++
	@cp $*.h++ algorithm.h++
	$(CXX) $(CXXFLAGS) $(IFLAGS) -DNAME=$* driver.c++
	@for n in $(N) ; do \
	  ./a.out $$n ; \
	done; \
	rm -f algorithm.h++ ./a.out

$(parabola-tests): %.parabola : %.h++
	@cp $*.h++ algorithm.h++
	$(CXX) $(CXXFLAGS) $(IFLAGS) -DPARABOLA -DNAME=$* driver.c++
	@for n in $(N) ; do \
	  ./a.out $$n ; \
	done; \
	rm -f algorithm.h++ ./a.out

$(turn-tests): %.turn : %.h++
	@cp $*.h++ algorithm.h++
	$(CXX) $(CXXFLAGS) $(IFLAGS) -DNAME=$* -DSQUARE -DMEASURE_TURNS
    ↪ driver.c++
	@for n in $(N) ; do \
	  ./a.out $$n ; \
	done; \
	rm -f algorithm.h++ ./a.out

$(arithmetic-tests): %.arithmetic : %.h++
	@cp $*.h++ algorithm.h++
	$(CXX) $(CXXFLAGS) $(IFLAGS) -DNAME=$* -DSQUARE -DMEASURE_ARITHMETIC
    ↪ driver.c++
	@for n in $(N) ; do \
	  ./a.out $$n ; \
	done; \
```

```
59          rm -f algorithm.h++ ./a.out
60
61  $(comp-tests): %.comp : %.h++
62          @cp $*.h++ algorithm.h++
63          $(CXX) $(CXXFLAGS) $(IFLAGS) -DNAME=$* -DSQUARE -DMEASURE_COMPS
        ↪ driver.c++
64          @for n in $(N) ; do \
65            ./a.out $$n ; \
66          done; \
67          rm -f algorithm.h++ ./a.out
68
69  $(move-tests): %.move : %.h++
70          @cp $*.h++ algorithm.h++
71          $(CXX) $(CXXFLAGS) $(IFLAGS) -DNAME=$* -DSQUARE -DMEASURE_MOVES
        ↪ driver.c++
72          @for n in $(N) ; do \
73            ./a.out $$n ; \
74          done; \
75          rm -f algorithm.h++ ./a.out
76
77  $(benchmarks): %.benchmark : %.h++
78          @date 2>&1 | tee -a $*.log
79  #       @make -i --no-print-directory $*.turn 2>&1 | tee -a $*.log
80  #       @make -i --no-print-directory $*.comp 2>&1 | tee -a $*.log
81  #       @make -i --no-print-directory $*.move 2>&1 | tee -a $*.log
82  #       @make -i --no-print-directory $*.arithmetic 2>&1 | tee -a $*.log
83          @make -i --no-print-directory $*.square 2>&1 | tee -a $*.log
84  #       @make -i --no-print-directory $*.parabola 2>&1 | tee -a $*.log
85          @date 2>&1 | tee -a $*.log
86
87  $(eliminations): %.elimination : %.h++
88          @date 2>&1 | tee -a $*.log
89          @make -i --no-print-directory $*.elimsquare 2>&1 | tee -a $*.log
90          @make -i --no-print-directory $*.elimdisc 2>&1 | tee -a $*.log
91          @date 2>&1 | tee -a $*.log
92
93  tr:
94          @date
95          @make -i --no-print-directory baseline.benchmark
96          @make -i --no-print-directory move_opt.benchmark
97          @make -i --no-print-directory opt1.benchmark
98          @make -i --no-print-directory opt2.benchmark
99          @make -i --no-print-directory opt3.benchmark
100         @make -i --no-print-directory opt4.benchmark
101         @date
102
103 TESTFLAGS=-O3 -std=c++2a -Wall -Wextra -x c++ -fconcepts -g -DDEBUG
104
105 $(sanitychecks): %.check : %.h++
106         @cp $*.h++ algorithm.h++
107         $(CXX) $(TESTFLAGS) $(IFLAGS) -DNAME=$* check-driver.c++
108         ./a.out
109         rm -f ./a.out
110
111 $(unittests): %.test : %.h++
112         @cp $*.h++ algorithm.h++
113         $(CXX) $(TESTFLAGS) $(IFLAGS) -DNAME=$* test-driver.c++
114         ./a.out
115         rm -f algorithm.h++ ./a.out
116
117 # Other tools
118
119 clean:
120         - rm  -f a.out temp algorithm.h++ 2>/dev/null
121
```

```
122  veryclean: clean
123          - rm -f *~ .*~ */*~ 2>/dev/null
124
125  find:
126          find . -type f -print -exec grep $(word) {} \; | less
```