Swinburne University of Technology

Faculty of Science, Engineering and Technology

MIDTERM COVER SHEET

Subject Code: COS30008
Subject Title: Data Structures and Patterns
Assignment number and title: Midterm, Convex Hull
Due date: May 2, 2021, 23:59
Lecturer: Dr. Markus Lumpe

Your name:______________________________ **Your student ID:**________________

Check Tutorial	Wed 08:30	Wed 10:30	Wed 16:30	Thurs 08:30	Thurs 10:30	Thurs 14:30	Thurs 16:30	Fri 08:30	Fri 10:30	Fri 14:30

Marker's comments:

Problem	Marks	Obtained
1	92	
2	138	
3	194 (buildConvexHull: 86)	
Total	424	

Midterm: Convex Hull

Preliminaries

The goal of this midterm is to implements Graham's scan to compute the convex hull of a set of points in 2D space. This problem requires three class types: `Vector2D` to represent quantities with magnitude and size, `Point2D` to encode points in 2D, and `Point2DSet` to provide a container for 2D points that can be used for storing, sorting, and systematic traversal of 2D point and the construction of the corresponding convex hull. In geometry, the convex hull of a set Q of points, written CH(Q), is the smallest convex polygon P for which each point of Q is either on the boundary of P or in its interior, and every line segment joining some points p_0 and p_1 of Q lies completely within polygon P:

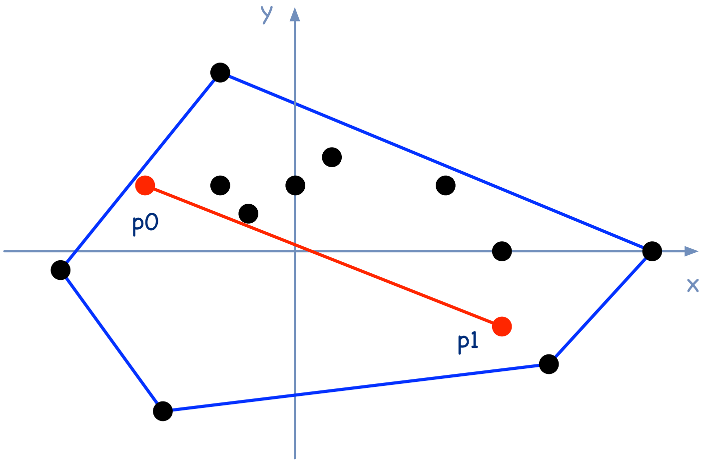

Figure 1: Convex Hull.

Computing the convex hull of a set of points is an interesting problem in its own right. Many algorithms for some other computational-geometry problems start by computing a convex hull. Graham's scan solves the convex hull problem by maintaining a stack S (or some other sequential data structure) of candidate points. Each point in the input set Q is pushed once onto the stack, and the points that are not vertices (i.e., boundary points) of CH(Q) are eventually popped from the stack. When the algorithm terminates, stack S contains exactly the vertices of CH(Q), in counterclockwise order of their appearance on the boundary:

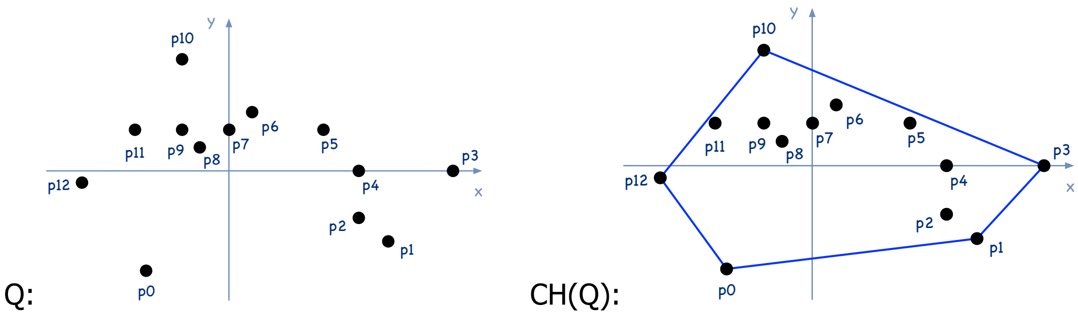

Figure 2: CH(Q) = {P12, P10, P3, P1, P0}.

Graham's scan takes as input a set Q of points, where the cardinality of Q is greater or equal to 3, $|Q| \geq 3$. If Q has less than 3 elements, then all those elements are in CH(Q). The position of points is defined relative to the origin point (0,0) of a Cartesian coordinate system. In a first step, Graham's scan identifies the point with the smallest y-coordinate. In case of a tie, the point with the smallest x-coordinate is chosen. In Figure 2, p_0 is that point. This point becomes the new origin of the set Q of points (see Figure 3 below). Using this setup, we can compute the polar coordinates of every point (i.e., the distance and the direction from the pole) in which the new origin points serves as the pole:

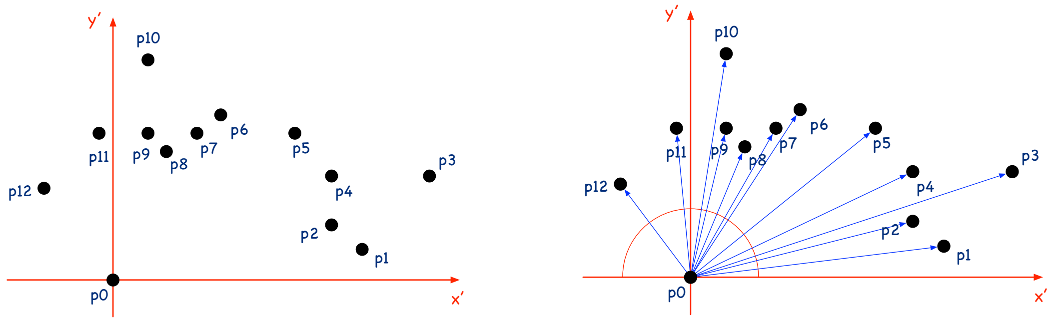

Figure 3: New Origin P0 and Polar Coordinates.

Using the polar coordinate system, we sort the set Q of points based on direction (i.e., polar angle in increasing order) and start the scan process to determine CH(Q) beginning with the pole and its next two closest points. Please note that points can be collinear, that is, they can occur on the same line segment. In such a case, the point with the greater distance wins. (Point p_2 is added initially only if it is not collinear to p_1. Point p1 would come before p2.)

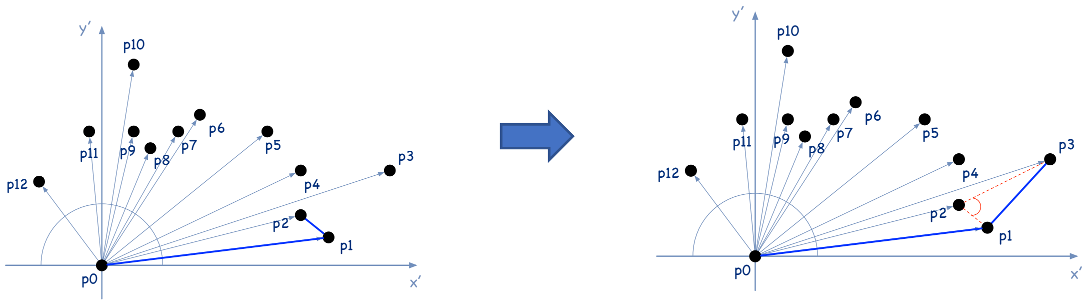

Figure 4: {P2, P1, P0} to {P3, P1, P0}.

Figure 4 illustrates how the algorithm determines that p_3 rather than p_2 is in CH(Q). The angle formed by points p_1, p_2, p_3 does not make a left turn. Consequently, point p_2 is not in CH(Q) and point p_3 is a new candidate in CH(Q). Whether or not a candidate point is in CH(Q) depends of the remaining points. For example, points p_5, p_6, and p_9 are in CH(Q) until we reach point p_{10}:

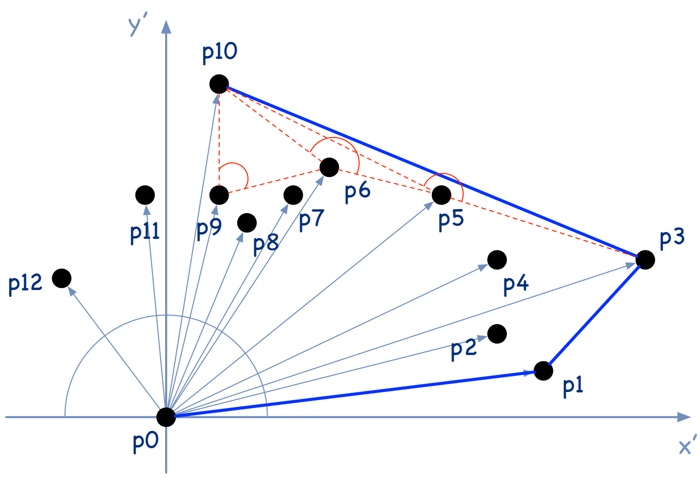

Figure 5: Point P10 is in CH(Q).

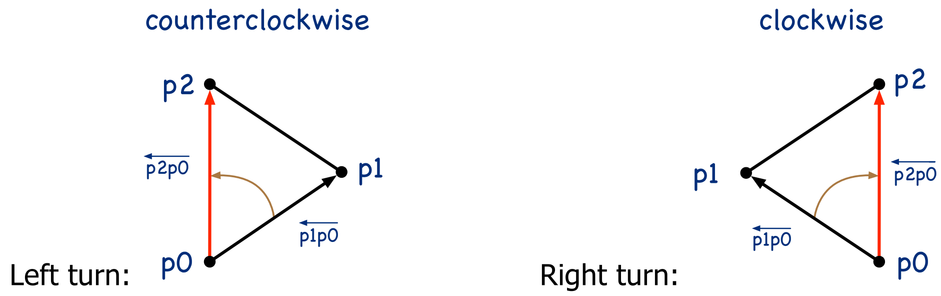

Figure 6: Two consecutive line segments turn as P1.

In order to implement Graham's scan, we need to determine whether two consecutive line segments $\overline{p_0p_1}$ and $\overline{p_1p_2}$ turn left or right at point p_1. Cross products of 2D vectors allow us to answer this question without computing the angle. We first have to build the vectors $\overleftarrow{p_1p_0}$ and $\overleftarrow{p_2p_0}$ and compute the cross product:

$$(p_2 - p_0) \times (p_1 - p_0) = (x_2 - x_0)(y_1 - y_0) - (x_1 - x_0)(y_2 - y_0).$$

If the sign of this cross product is negative, then $\overleftarrow{p_2p_0}$ is counterclockwise with respect to $\overleftarrow{p_1p_0}$, and thus we make a left turn at p_1. A positive cross product indicates a clockwise orientation and a right turn. A cross product of 0 means that points p_0, p_1, and p_2 are collinear.

Please note, the cross product is actually a three-dimensional concept. For 2D vectors it is given as the determinate of a matrix:

$$\begin{aligned} p_1 \times p_2 &= \det \begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix} \\ &= x_1y_2 - x_2y_1 \\ &= -p_2 \times p_1 \end{aligned}$$

Graham's scan

1. Let p_0 be the point in Q with the minimum y-coordinate, or the leftmost such point in case of a tie.
2. Let $\{p_1, p_2, \ldots, p_n\}$ be the remaining points in Q, sorted by polar angle in counterclockwise order around p_0.
3. Let S be a stack-like data structure and push p_0, p_1, and p_2 onto S. (If $|Q| < 3$, stop.)
4. **for** i = 3 **to** n
5.   **while** the angle formed by points `Next-To-Top(S)`, `Top(S)`, and p_i makes a non-left turn **do**
6.   `Pop(S)`
7.   `Push(S, pi)`
8. **return** S

The operation `Top(S)` yields the top element on the stack without removing it, whereas `Next-To-Top(S)` provides access to the element beneath `Top(S)` without removing it. `Pop` and `Push` have the usual stack semantics.

Please note that the midterm utilizes the C++ library type `std::vector`, a class template for resizable arrays. This type can be used to store the set Q of points, order points based on the minimum y-component and polar angle, and defines methods that allow us to use it like a stack. A stack can be viewed as an array (or `vector`) of elements that is changed at one end only.

Test Driver

```
#include <iostream>
#include "Point2DSet.h"

using namespace std;

void main()
{
  Point2DSet lSet;

  lSet.populate( "points.txt" );

  cout << "Points:" << endl;

  for ( const Point2D& p : lSet )
  {
    cout << p << endl;
  }

  Point2DSet lConvexHull;

  lSet.buildConvexHull( lConvexHull );

  cout << "Convex hull:" << endl;

  size_t lSize = lConvexHull.size();

  for ( size_t i = 0; i < lSize; i++ )
  {
    cout
        << lConvexHull[i].getId()
        << " - "
        << lConvexHull[(i + 1) % lSize].getId() << endl;
  }

  return 0;
}
```

Output (for `points.txt`):

```
Points:
p00: (-5,-6)
p01: (6,-4)
p02: (5.5,-3)
p03: (8,0)
p04: (5,0)
p05: (4,2)
p06: (1,3)
p07: (0,2)
p08: (-1,1)
p09: (-1.5,2)
p10: (-1.5,6)
p11: (-5.5,1.5)
p12: (-8,-1)
Convex hull:
p00 - p01
p01 - p03
p03 - p10
p10 - p12
p12 - p00
```

Problem 1

Class `Vector2D` defines the infrastructure to represent quantities with magnitude and size:

```
#pragma once

#include <iostream>

class Vector2D
{
private:
  double fX;      // x coordinate
  double fY;      // y coordinate

public:
  // 2D vector constructor; default is unit vector î
  Vector2D( double aX = 1.0, double aY = 0.0 );

  // getters & setters for x and y coordinates
  void setX( double aX );
  double getX() const;
  void setY( double aY );
  double getY() const;

  // 2D vector addition: this + aRHS; returns a fresh 2D vector
  Vector2D operator+( const Vector2D& aRHS ) const;

  // 2D vector subtraction: this - aRHS; returns a fresh 2D vector
  Vector2D operator-( const Vector2D& aRHS ) const;

  // Length (or magnitude) of a 2D vector
  double magnitude() const;

  // Direction (angle) of 2D vector
  // The angle is the tangent of y divided by x (hint: atan2)
  double direction() const;

  // Inner product (scalar product) of two 2D vectors
  // Does not require angle between vectors
  double dot( const Vector2D& aRHS ) const;

  // In 2D, the cross product of two 2D vectors is
  // the determinate of the matrix
  //
  //        | x1 x2 |
  //    det |       | = x1*y2 - x2*y1
  //        | y1 y2 |
  //
  double cross( const Vector2D& aRHS ) const;

  // Angle between two 2D vectors
  // The function must properly handle null vectors = [0,0]
  // and return an angle consistent with the dot product.
  double angleBetween( const Vector2D& aRHS ) const;

  // I/O for 2D vectors
  friend std::ostream& operator<<( std::ostream& aOutStream,
                                   const Vector2D& aObject );
  friend std::istream& operator>>( std::istream& aInStream,
                                   Vector2D& aObject );
};
```

The methods of `Vector2D` define the standard features of 2D vectors. Objects of `Vector2D` can be null vectors, that is, `[0,0]`. The methods `dot()` and `angleBetween()` must properly handle null vectors. The dot product of two vectors is zero if the vectors are orthogonal.

The input operator for `Vector2D` just has to read the x- and y-coordinates.

The output operator has to produce the usual vector notation. That is a `Vector2D` object with `fX = 2` and `fY = 3` has to send to the output steam a string “`[2,3]`”.

There is no test driver for `Vector2D`. You may define one yourself to guarantee the correctness of your implementation.

Problem 2

Points in 2D are represented by objects of type `Point2D`:

```
#pragma once

#include "Vector2D.h"

#include <iostream>
#include <string>

class Point2D
{
private:
  std::string fId;                // point id
  Vector2D fPosition;             // position in 2D
  const Point2D* fOrigin;         // coordinate reference point, initially (0,0)

  // Direction (angle) of point w.r.t. aOther
  double directionTo( const Point2D& aOther ) const;

  // Length (or magnitude) of point w.r.t. aOther
  double magnitudeTo( const Point2D& aOther ) const;

public:
  // constructors
  Point2D();
  Point2D( const std::string& aId, double aX, double aY );
  explicit Point2D( std::istream& aIStream );

  // getters & setters
  const std::string& getId() const;
  void setX( const double& aX );
  const double getX() const;
  void setY( const double& aY );
  const double getY() const;
  void setOrigin( const Point2D& aPoint );
  const Point2D& getOrigin() const;

  // 2D vector this - aRHS
  Vector2D operator-( const Point2D& aRHS ) const;

  // Direction (angle) of point w.r.t. origin
  double direction() const;

  // Length (or magnitude) of point w.r.t. origin
  double magnitude() const;

  // Are this point and aOther on the same line segment?
  bool isCollinear( const Point2D& aOther ) const;

  // Does line segment this-aP2 make a right turn at this point?
  bool isClockwise( const Point2D& aP0, const Point2D& aP2 ) const;

  // Is this' y-coordinate less than aRHS's y-coordinate?
  // If there is a tie, this' x-coordinate less than aRHS's x-coordinate?
  bool operator<( const Point2D& aRHS ) const;

  // I/O for 2D points
  friend std::ostream& operator<<( std::ostream& aOStream,
                                   const Point2D& aObject );
  friend std::istream& operator>>( std::istream& aIStream,
                                   Point2D& aObject );
};
```

Class `Point2D` encodes points in 2D. Objects of class `Point2D` have a name or id, a position in 2D space expressed as a 2D vector, and an origin that provides a coordinate reference. The latter can change depending on the way we look at 2D points. When we create points initially, the origin is set to (0,0), the origin of a Cartesian coordinate system. To facilitate this approach, the application has to define a constant global `Point2D` object `gCoordinateOrigin` that represents the 2D point (0,0). Ideally, this constant global object should be defined in `Point2D.cpp`, like:

```
static const Point2D gCoordinateOrigin;
```

where **`static`** means, `gCoordinateOrigin` it is only visible within compilation unit `Point2D.cpp`. The declaration uses the default constructor for Point2D.

Class `Point2D` defines three overloaded constructors:

- `Point2D()`: the default constructor that sets coordinates to (0,0) and name to the empty string; the origin is `gCoordinateOrigin`,
- `Point2D( const std::string& aId, double aX, double aY)`: sets coordinates and name to corresponding values; the origin is `gCoordinateOrigin`,
- `Point2D( std::istream& aIStream )`: sets coordinates and name to corresponding values read from `aIStream`; the origin is `gCoordinateOrigin`.

The getters and setters provide the usual semantics.

Class `Point2D` also defines two private member functions: `directionTo()` and `magnitudeTo()`. The former returns the direction (i.e. angle) from the target point to `aOther`, whereas the latter returns the length of the vector between `aOther` and the target object. These member functions allow for the calculation/implementation of `direction()` and `magnitude()` with respect to the origin of a 2D point.

The `operator-()` yields 2D vector between the target object and `aRHS`.

The predicate `isCollinear()` returns true if the target object and `aOther` are on the same line segment. To guarantee numerical stability, deviations of less than 10^{-4} are considered equal.

The predicate `isClockwise()` returns true, if a line segment to `aP2` makes a right turn at the target point (see Figure 6).

The `operator<()` returns true if the target point has a smaller y-coordinate that `aRHS`. If there is a tie, then the one with the smallest x-coordinate wins. To guarantee numerical stability, deviations of less than 10^{-4} are considered equal.

The input operator for Point2D just has to read the name/id and x- and y-coordinates.

The output operator has to produce the usual point notation. That is a `Point2D` object with `fX = 2` and `fY = 3` has to send to the output steam a string "`(2,3)`". In addition, the output is prefixed with the name of the 2D point. For example, a 2D point with `fId = "p00"` and `fX = 2` and `fY = 3` has produce the output "`p00: (2,3)`".

There is no test driver for `Point2D`. You may define one yourself to guarantee the correctness of your implementation.

Problem 3

We use class `Point2DSet` to define a container for 2D points that can be used for storing, sorting, and systematic traversal of 2D points and the construction of the convex hull thereof.

```
#pragma once

#include "Point2D.h"

#include <vector>
#include <iostream>

class Point2DSet
{
private:

  using PointVector = std::vector<Point2D>;

  PointVector fPoints;

public:

  using Iterator = std::vector<Point2D>::const_iterator;
  using Comparator = bool (*)( const Point2D&, const Point2D& );

  // Add a 2D point to set
  void add( const Point2D& aPoint );
  void add( Point2D&& aPoint );

  // Remove the last point in the set
  void removeLast();

  // Tests aPoint, returns true if aPoint makes no left turn
  bool doesNotTurnLeft( const Point2D& aPoint ) const;

  // Load 2D points from file
  void populate( const std::string& aFileName );

  // Run Graham's scan using out parameter aConvexHull
  void buildConvexHull( Point2DSet& aConvexHull );

  // Returns numner of elements in set
  size_t size() const;

  // Clears set
  void clear();

  // Sort set using stable_sort on vectors
  void sort( Comparator aComparator );

  // Indexer for set
  const Point2D& operator[]( size_t aIndex ) const;

  // Iterator methods
  Iterator begin() const;
  Iterator end() const;
};
```

Class `Point2DSet` defines a container for 2D points. It uses `std::vector` for storage. The type `std::vector` is a C++ standard class template and provides a sequence container representing arrays that can change in size. We instantiate `std::vector` with `Point2D`, which yields a vector of 2D points. We will manipulate this vector at one and only. That is, when we add an element, then the element is inserted at the end. If we need to remove an element, then that element should be the last element in the vector. Using vectors this way allows us to view vectors as stacks. The class template `std::vector` provides the necessary operations.

Class `Point2DSet` does not require any user defined constructors. C++ synthesizes the required default constructor, which initializes the instance variable `fPoints` to an empty vector of 2D points.

We can use one of the two `add()` methods to insert 2D points into the set. The first uses a l-value parameter to copy the argument into the set. The second takes an r-value parameter to move the argument into the set.

The method `removeLast()` removes the last element in the set.

The predicate `doesNotTurnLeft()` returns true if `aPoint` does not make a left turn with respect to the neighboring points in the set. This is a crucial helper method to run Graham's scan. There must be at least 3 elements. Otherwise, there is a domain error.

The method `populate()` reads point data from a text input file. The file contains the number of points followed by the respective point data.

The method `buildConvexHull()` implements Graham's scan. We are not using a stack, as suggested by the algorithm, but `Point2DSet` object `aConvexHull`. Instances of Point2DPoint provide the necessary stack-like abstractions to map the operations `Top(S)`, `Next-To-Top(S)`, `Push(S, p`$_i$`)`, and `Pop(S)`. When method `buildConvexHull()` finishes, the reference argument `aConvexHull` contains the 2D points that constitute the convex hull.

The methods `size()` and `clear()` return the number of elements in the set and empty the set, respectively.

The method `sort()` takes a comparator, a Boolean function with two 2D point arguments, and performs in-place sorting of the underlying set, that is, the vector `fPoints`. The C++ template class `std::vector` does not directly support sorting. You need to use `stable_sort` defined in `algorithm`. The procedure `stable_sort` requires three parameters: an iterator positioned at the first element, an iterator to mark the end, and a binary function that returns true if its first argument is considered to go before the second. To complete Graham's scan, you need two comparators:

- ```
  bool orderByCoordinates( const Point2D& aLeft,
                           const Point2D& aRight )
  ```
- ```
  bool orderByPolarAngle( const Point2D& aLHS,
                          const Point2D& aRHS )
  ```

Again, to guarantee numerical stability, deviations of less than 10^{-4} are considered equal.

Finally, the indexer **`operator[]`**`()` and the iterator methods `begin()` and `end()` have the expected semantics. The C++ template class `std::vector` defines the necessary abstractions to implement the behavior. (Actually, class `Point2DSet` defines an object adapter for `std::vector` to represent 2D points and the construct their respective convex hull.)

You can use the test driver define in `Main.cpp` to test your implementation.

Submission deadline: Sunday, May 2, 2021, 23:59.

Submission procedure: PDF of printed code for code of **`Vector2D`**, **`Point2D`**, and **`Point2DSet`**.