Computational Geometry **Module 1-3** **NAME: Kevin Riggs**

PROBLEM SET 1

Scoring Rubric:

Points Earned on Problem	**Criteria**
4	Excellent. All work is correct. Neatly written and legible. Clearly presented and worked through without a fault.
3	Good. Only slight error in work. Neatly written and legible. Clearly presented and clear understanding of the concepts. Only possesses mistakes in calculations.
2	Fair. Errors in work. Illegible and not written neatly. Work is not presented in a way that shows understanding of the concepts. Calculation is not clear or correct.
1	Poor. Work is filled with errors, work is not clear and concepts are confused. Needs work to bring their work up to this courses expectations.
0	Immediate action needs to be taken to correct your understanding and/or effort in this class.

Directions: Each problem from the Problem Set will be worked out in its entirety. Your solution method may vary and will not be used against you if you get the right answer. You are expected to honor the following rules:

- Do not read any homework solutions or "sample solutions", including those produced by students (except yourself), instructors, or TAs, related to this or any other course on Computational Geometry or a closely related topic. Some of the homework questions I'll use this semester have been used before by me or other teachers at other institutions, and you're on your honor not to consult the solutions.
- Do not copy solutions or work from classmates or any work you've seen online. The purpose of the problems is to grow your understanding of the subject manner and refine your skills at writing and implementing algorithms. • Do not give solutions to each other. Learning is not done by copying solutions.

You may consult any books, papers, web pages, or other inanimate information sources you please, so long as they're not other people's homework solutions or sample solutions. If you use information from such a source, please cite it. I don't believe that the solutions to many of the problems I'll give are available in easy-to-find references, but if you do find one, it's fair game as long as you cite it. After all, a good literature search should be rewarded. You may not, however, let your classmates know about your discovery or sources until the deadline has passed.

You are welcome to ask me to clarify anything in the readings, the lectures, or this homework.

I prefer typeset solutions, on paper. I will accept handwritten solutions with the proviso that I reserve the right to give a zero to a solution if there is any word or variable I can't read clearly. (Sorry, I just don't have time to infer from context whether that vague squiggle is an i or a j.) Any such decision is final and irrevocable. By contrast, typeset solutions reserve the right to argue with my grading after the fact.

On the other hand, handwritten figures will be gratefully accepted. Anything to encourage you to draw figures! A simple figure can often mean the difference between my spending five minutes or two hours grading an answer. Thank you in advance for every figure you draw.

1. (Sorting Points) Let E be an unsorted set of n segments that are the edges of a convex polygon. Describe an O($n$log$n$) algorithm that computes from E a list containing all vertices of the polygon, in clockwise order.

I apologize for the low quality in my answers, it feels a little bit disrespectful turning it in. Future submissions will be more thoughtful as this is the last week where I'm working dusk til dawn. As I write this today, the assignment's due date, we finished test verification. Long story short, we went through a reorg, I was put on a new team in December and inherited a project that needed a lot of love in short time. Now the project is healthy and I'm back to normal hours.

```
An answer is:
Result list: res
Pop the first edge to get Ph and Pt

For edge in E
    For point in edge
        Calculate the angle made by Ph Pt point, call this constant time
        Insert point according to calculated angle (All will be less than pi), call this O(logn)
Insert Ph and Pt
```

Another algorithm uses a few data structures. First, the original unsorted set of n segments. Second, A, a doubly linked list of points which is the final output of points in a clockwise order. This initial list starts with the points from the first edge which are its head and tail. Finally, there is a binary sorted list of the heads and tails by x coordinate then y coordinate, BAL, of doubly linked lists, Li, that hold intermediate chains of points. Each of these lists have a head and tail. The final list is built up outside in, the intermediate lists are built inside out. Searching for a point in BAL is O(logn) updating BAL is O(logn).

```
Pop the first edge from E, set the first point as the head of A the second as the tail and both as lead points.
For edge in E, check point membership in A and BAL
    Case: neither point is a lead point in A or a head/tail point in one of the lists in AL
        Add the points as a new list in AL
    Case: neither a lead point but is lead point in one Li in AL
        Update Li in AL with points
    Case: neither is a lead point but is a lead point in two Li in AL
        Update and merge the two respective Li in AL
    Case: one is a lead point and one is a lead point in an Li from AL
        Update A and AL, merge respective Li into A
    Case: one is a lead point and the other is not a lead point in Li From AL
        Update A
    Case: each point in edge is respectively a lead point in A
        Link each point to leads and each other respectively
        Check directionality of A, flip head/tail as necessary
        Done
```

Analysis:

Computational Geometry **Module 1-3** **NAME: Kevin Riggs**

PROBLEM SET 1

We need to iterate through each edge, O(n).
The max size of BAL is between 0 and n/2.
Checking if a point is a lead, head or tail is constant time O(1)
Checking if a point is a lead in BAL is O(logn)
Updating BAL is O(logn)
Updating and merging the lists are constant time O(1)

Iterate through each edge O(n) *(Look ups in BAL O(logn) + updates in BAL O(logn))
O(nlogn)

2. (Point Location Test) For a convex hull algorithm, we have to be able to test whether a point *r* lies to the left or right of the directed line through two points *p* and *q*. Let *p* = (p_x,p_y), *q* = (q_x,q_y), and *r* = (r_x,r_y).

   (a) Show that the sign of the determinant

$$D = \begin{vmatrix} 1 & p_x & p_y \\ 1 & q_x & q_y \\ 1 & r_x & r_y \end{vmatrix}$$

   determines whether the point *r* lies to the left or the right of the line.

Visual, hands-on proof by the right hand rule. Lay the back of your hand on the table. Point your index finger and thumb to make an L. Your fingertip to your hand crease represents p to q, the crease to your thumb represents q to r. Point your other fingers toward the ceiling, they represent the sign of the determinant. As you twist your hand around, the determinant always stays positive and r always stays to the directional right of pq. If you flip your hand palm down, as you turn your hand, r stays to the left and the determinant stays negative.

   (b) Show that |*D*| in fact is twice the surface of the triangle determined by *p*,*q* and *r*.

Given two sides of triangle, a and b, and the angle between them, C, the area of the triangle is a*b*sin(C)/2

Another way to calculate the determinant, given two vectors, V1 and V2, and the angle between them theta, is |V1|*|V2|*sin(theta). Swapping a, b, and C for V1, V2 and theta we get the absolute value of the determinant is twice the area of a triangle.

   (c) Why is this an attractive way to implement the basic test in ConvexHull? Give an argument for both floating point and integer coordinates.

It can also check for collinearity. We only need to do 6 multiplications.

The main differences I can think of between floating point and integers is precision and time complexity for arithmetic operations. So an argument for using this with floating point is we'd only need a couple, constant amount, of FLOPs. For integers, Maybe an argument for both is depending on the order of operations we choose we can improve our chances of avoiding overflow or underflow.

Computational Geometry **Module 1-3** **NAME: Kevin Riggs**
PROBLEM SET 1

3. (Convex Hull) Write your own algorithm for finding the convex hull CH(*P*) of a set of *n* points *P*. You should have code written that does the following:

   (a) Create a random set of *n* points in the plane

   (b) Compute the convex hull and return the list CH(*P*) in clockwise order

Your submission of the code should include a description of how to run the code and any credits for code that you borrowed. Not including this description will result in losing credit for this problem.

Execution instructions:
Double clicking the .exe or running from the command line. It will open up three windows, one showing the random points, one showing the convex hull from OpenCV's built in function and one showing the convex hull from my implementation. The command line output shows the points ordered in "clockwise" order. That is in quotations because visually in most computer coordinates, my implementation is showing clockwise. However, everywhere else it is counterclockwise as normally the positive y direction goes up not down.

Apologies, I didn't test on a second machine that doesn't have OpenCV installed. It's also only built for Windows. I'm 95% sure it will not run unless you have the exact opencv version installed. Being strapped for time I didn't update the build script to either put everything in the executable or include the required libs. But there's a chance it will work. If cmake, opencv and a c++ compiler is installed it can just be built and run with the cmakelists.txt and source file.

Implementation and Algorithm resources:
https://www.geeksforgeeks.org/convex-hull-set-2-graham-scan/

4. (Doubly-Connected Edge List) Suppose that a doubly-connected edge list of a connected subdivision is given. Give pseudocode for an algorithm that lists all faces with vertices that appear on the boundary.

My thought is for each face, if the outercomponent is null or the outercomponent of the incident face of the twin is null, the face has a vertex on the boundary.
So we will iterate through each face, if the outercomponent is null we will added to the set of faces that will be turned into the list of faces to return. Otherwise we check to see if the incident face of the twin of the outercomponent is null(I'm going to error on the side of caution and check each of those edges.)

We could improve this by cutting short when we come across a face we have already seen. Possibly even preemptively remove faces of inner components.

```
BoundaryVertexFaces(Input: double-connected edge list L, Output: List of faces with vertices on boundary F):
    For face in L.Faces:
        If outercomponent(face) == null
            Add face to F
        Else
            For edge in edges around twin(outercomponent(face)):
                If outercomponent(incidentFace(edge)) == null:
                    Add face to F
    ConvertSetToList(F)
    Return F
```

Computational Geometry **Module 1-3** **NAME: Kevin Riggs**
PROBLEM SET 1

5. (Triangulating Polygons) Prove that any polygon admits a triangulation, even if it has holes. Can you say anything about the number of triangles in the triangulation?

Theorem from lecture 2: Every simple polygon admits a triangulation and any triangulation of a simple polygon with n vertices consists of exactly n-2 triangles.

I'm not sure what to add so I just put some thoughts down about the proof.

Proof by induction works by showing it is true for a simple case, n=3, assuming it is true for an arbitrary case, n>3 and m<n, and showing it is true when changing the arbitrary case, M1 < n and M2 < n because both M1 and M2 are less than m. Those subpolygons can be triangulated because M1 and M2 are less than n, or they are triangles, which means the original polygon can be triangulated.

The interesting part of this proof is that the induction is climbing down a ladder(Splitting polygons into smaller ones until we get to a simple triangle) instead of up a ladder(n=1 is true, show n=m+1 is true assuming n=m is true).

So when reading over the proof in the lecture two slides. I might be misreading or misunderstanding but I think what we want to say is that two vertices of P occur in each of the two subpolygons. In that way we get M1 + M2 = N – 2. Which is true for the simple case of a square with a diagonal through it. As I'm reading it right now it's saying the two subpolygons do not share any vertices but I may be misreading.

6. (Delaunay Triangulations) Let A and B be two disjoint point sets in the plane. (Think of them as a red point set and a black point set.) Let $a \in A$ and $b \in B$ be two points from these sets that minimize the Euclidean distance $|ab| = \sqrt{(a_x - b_x)^2 + (a_y - b_y)^2}$ where $a = (a_x, a_y)$ and $b = (b_x, b_y)$. Prove that $\overline{ab}$ is an edge of DT($A \cup B$).

My goal is to prove that to not have this edge a part of the triangulation would result in an illegal edge. Consider if we raise Pj and lower Pi, in the screenshot below, to where it is clear that the Euclidean distance is minimized through Pl and Pk. Doing this still wouldn't change the validity of the example. A legal triangulation maximizes the minimal angles of the resulting triangles. In the simplest case, not including the edge that minimizes the Euclidean distance in the triangulation would result in an illegal edge. Additionally, it does not matter if we add points above or below the respective point sets of [Pl,Pj] and [Pi,Pk].

Computational Geometry **Module 1-3** **NAME: Kevin Riggs**
PROBLEM SET 1

Edge Flips

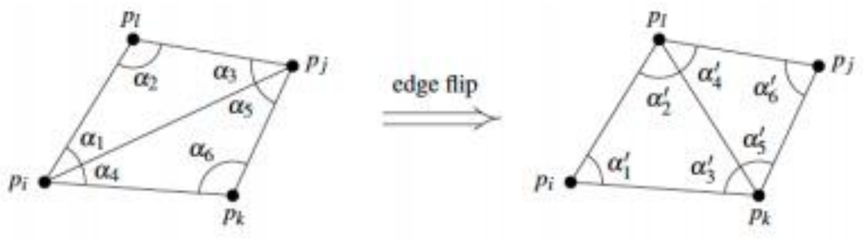

Figure 10: Flipping an edge.

Note We call the edge e an *illegal edge* if $\min\limits_{1 \le i \le 6} \alpha_i < \min\limits_{1 \le i \le 6} \alpha_{i'}$.