An Overview of Computational Geometry

Alex Scheffelin

Contents

1 Abstract

Computational geometry is an applied math field which can be described as the intersection of geometry and computer science. It has many real world applications, including in 3D modelling programs, graphics processing, etc. We will provide an overview of the methodology of computational geometry in the limited case of 2D objects, culminating in the discussion of the construction of convex hulls, a type of polygon in the plane which is determined by a set of points. Along the way we will try to explain some unique difficulties one faces when using a computer to compute things, and attempt to highlight the methods of proof used.

2 Introduction

2.1 Basic Definitions

I will define a few geometric objects, and give example about how one might represent them in a computer. The reader is free to skip this section and proceed to section 3 on floating point error if they find this section boring, or unneeded.

Definition 1. Point
A point is a point in the 2D plane represented uniquely by its x and y coordinates. One can represent a point as an ordered pair (x, y) where x is its x-coordinate, and y is its y-coordinate.

While it may seem pointless to define what a point is, nearly everything builds off of points, so it was worth defining. One can simply make a data structure called "point" which has two variables, two floating point numbers called "x" and "y" that simply represent its coordinates. From points, we build up to line segments.

Definition 2. Line Segment
A line segment is the geometric construct which spans between two points, and contains within it every point between them. It matters not what one considers the start and ending point as the line which goes from the point x to the point y, the line called xy is the same as the line which goes from y to x, the line called yx.

A precise definition of a line segment is very hard, as one must have some point to start from. Euclid took lines to be the start of his geometry. The reason for defining a line segment is to provide a way to discuss how one might store them in a computer. The easiest way is to simply create two points which are variables in it, say p_1 and p_2. The issue with this definition is that when it comes to evaluate equality of two lines, say xy and uv, one might seek to simply compare the two first points, and the two latter points, say $xy = uv$ if and only if $x = u$, and $y = v$. However as anyone who has taken geometry would know, the line $xy = yx$, thus one must check it two ways, $xy = uv$ if and only if $x = u$ and $y = v$, or $x = v$ and $y = u$. Note that we may have some degenerate cases, say what if $p_1 = p_2$? Then in reality we have only defined a point, and when performing geometric calculations with this line, one might possibly run into issues if one does not consider this case. We thus define the word degenerate,

Definition 3. Degenerate
Something which requires exceptional care to handle.

In computational geometry one example of common degenerate cases are when two lines are almost parallel, or when two things intersect only tangentially. In the first case, when two things are almost parallel, their intersection is very far away, and as we will later see in section 3 on floating point error, things being very large (in this case the x and y values of the intersection point) can cause issues with precision. We now will define a polygon, and simple polygon, and we will then proceed onto a discussion on floating point error.

Definition 4. Polygon
A polygon is a type of shape which is comprised of a series of line segments.

A polygon in general is a bit more general than what you might imagine to be a polygon. For example, this definition does not rule out the possibility that the polygon has self-intersection points. In order to store a polygon, one will usually store it as an ordered list of points. Thus a polygon can be stored as $\{p_1, p_2, \ldots, p_n\}$ which will describe its shape. If one were to try to draw the polygon, you would have to draw every edge, and if you store the data this way you can simply draw the lines $p_1p_2, p_2p_3, \ldots, p_np_1$. Note, as the last line we choose to draw is p_np_1, this polygon is not "closed", in the sense that it is understood that the final point is not the first point, and in order to draw it you must manually draw the last line p_np_1. It is common to store polygons this way as one can retrieve all the same data about the polygon, but for a polygon with n sides you need only store n points rather than $n+1$ points. The type of polygon you are more familiar with is called a simple polygon.

Definition 5. Simple Polygon
A polygon is a simple polygon where no line segments intersect except for those which are adjacent in the ordering.

What "adjacent in the ordering means" is simply that they are lines which would have to be next to each other. Take the polygon generated by the points $\{p_1, p_2, \ldots p_n\}$, the line segments of this are $\{p_1p_2, p_2p_3, \ldots p_np_1\}$. It is clear that line segments which are next to each other in this ordering (we also take p_np_1 and p_1p_2 to be next to each other) do intersect, namely at their endpoints. The lines p_kp_{k+1} and $p_{k+1}p_{k+2}$ will intersect at p_{k+1}, but this does not contradict the polygon being a simple polygon. Simple polygons are in general easier to work with, self intersections bring a whole slew of issues which make some operations more tedious, inefficient, etc. but polygons as a whole are a fundamental geometric object. They can be used to model floors, the boundary of regions on a map, etc. The 3D corollary of a polygon, a polyhedron is a fundamental object in 3D graphics, and the n-dimensional corollary is called a polytope, but we will not discuss them as we limit ourselves to the 2D case.

3 Floating Point Error

3.1 Imprecision in Floating Point Operations

One of the first things that must be discussed is how one stores geometric information. At the very beginning, one must use some programming language, perhaps, C#, C++, Java, etc. and then create data structures in order to contain all necessary information. The issue there lies in how this information is stored, real numbers are often stored as floating-point numbers, which are imprecise. The reason being that while one could write down the number, say 0.12438592037502830457923 9, to store that would require a large amount of memory as it eventually gets turned into binary. For an overview of floating point numbers in C#, see

this article[1]. As a brief summary, not all numbers can be precisely represented, and even if two numbers are precisely represented, their sum, product, quotient, etc. may not be. Thus one can continue to accumulate little bits of error, and if one is say off by a degree of 1×10^{-5}, if you are to multiply this number by a large number, you have only magnified the error. Thus, one must be cognizant of how error is handled in their algorithms. There is a balance to be made between speed and precision, one could allocate a lot more data in order to store numbers more precisely, but the tradeoff being that the speed of calculations involving the data will take more time.

3.2 Precision in the Calculation of the Area of a Polygon

In order to illustrate this point, we present a more precise algorithm for the computation of the area of a polygon in a paper by Jonathan Shewchuk[2]. A commonly known formula to calculate the area of a polygon with vertices $p_1, p_2, \ldots, p_n$ (ordered such that the polygon consists of lines between the points p_k and p_{k+1}, with $p_{n+1} = p_1$), and such that $p_{k,x}$ denotes the x-coordinate of p_k and $p_{k,y}$ the y-coordinate, that the area is

$$\frac{1}{2}\left|\sum_{k=1}^{n} p_{k,x}p_{k+1,y} - p_{k,y}p_{k+1,x}\right|$$

Note, that the presence of absolute value is because depending on whether the points go clockwise or counterclockwise, you may end up with either the area of the polygon, or the additive inverse of the area of the polygon. A simple way to see this is using Green's theorem. If the set P represents the set of points within the polygon we are looking at, then $\int\int_P 1dA$ is exactly the area of the polygon. To set this up to use Green's formula, let $P(x, y) = 0$ and $Q(x, y) = x$, and then

$$\int\int_P 1dA = \int\int_P \frac{\partial Q}{\partial x} - \frac{\partial P}{\partial y}dA = \oint_{\partial P} Pdx + Qdy = \oint_{\partial P} xdy$$

We can evaluate this along each edge, the let the curve C_k be the straight line between p_k and p_{k+1}, we can parameterize this curve using $\gamma(t) = ((p_{k+1,x}-p_{k,x})t+p_{k,x}, (p_{k+1,y}-p_{k,y})t+p_{k,y})$, and then plugging this in we get

$$\begin{aligned}
\oint_{\partial P} xdy &= \sum_{k=1}^{n}\int_0^1 ((p_{k+1,x} - p_{k,x})t + p_{k,x})(p_{k+1,y} - p_{k,y})dt \\
&= \sum_{k=1}^{n} \frac{(p_{k+1,x} + p_{k,x})(p_{k+1,y} - p_{k,y})}{2} \\
&= \frac{1}{2}\sum_{k=1}^{n} p_{k,x}p_{k+1,y} - p_{k,y}p_{k+1,x} + p_{k+1,x}p_{k+1,y} - p_{k,x}p_{k,y} \\
&= \frac{1}{2}\sum_{k=1}^{n} p_{k,x}p_{k+1,y} - p_{k,y}p_{k+1,x}
\end{aligned}$$

(The fact that those last two terms in the sum disappear is due to the fact that it telescopes and that $p_1 = p_{n+1}$)

This however, can be prone to large errors if the distance of the points of the polygon to the origin are large compared to the relative distance to each other. This is because due to the nature floating point numbers are stored, numbers large in absolute value are less precise than numbers small in absolute value. Thus, if the relative distance of the points is small, when summing the $(p_{k,x}p_{k+1,y} - p_{k,y}p_{k+1,x})$ terms, as each term is small it will be relatively precise, but as all the numbers, $p_{k,x}, p_{k+1,y}, p_{k,y}, p_{k+1,x}$ are large (as the vertices are far from the origin), the multiplication will be very imprecise, which may lead to the calculation of a number very far from the actual area of the polygon. The strategy offered here, is to notice that the calculation of the area of a polygon matters only on the relative positions of the vertices, thus it is translation invariant. Take a square with side lengths 1, regardless of where it is centered its area is 1, thus we can simply translate all the points such that they lie closer to the origin and then conduct this same calculation. Thus, let $p'_k = p_k - p_n$ (the idea here is that if we subtract the same fixed difference from each vertex, the area will not change), and substitute p'_k wherever p_k was before. Thus our formula becomes

$$\frac{1}{2}\left|\sum_{k=1}^{n} p'_{k,x}p'_{k+1,y} - p'_{k,y}p'_{k+1,x}\right| =$$
$$\frac{1}{2}\left|\sum_{k=1}^{n}(p_{k,x} - p_{n,x})(p_{k+1,y} - p_{n,y}) - (p_{k,y} - p_{n,y})(p_{k+1,x} - p_{n,x})\right|$$

One can verify by expanding the expression that this is equivalent to the formula offered earlier, but the benefit is that on average the terms used should be closer to 0. Note that if we have an incredibly large polygon centered on the origin, say a square with the points $(\pm 10^5, \pm 10^5)$, by shifting we may not really improve the accuracy. The increase in accuracy is gained when the polygon is relatively small compared to how far removed it is from the origin. The paper uses forward error analysis to get estimates of the error in both representations, the first one is a function of the size of the coordinates, so each points absolute distance from the origin, while the latter is a function only of their distance to each other, so their relative distance. Note, however, that as mentioned before, there are often tradeoffs between geometric robustness (the precision of geometric calculations) and speed. For the latter algorithm one must perform more calculations as you are take differences of points, which means it will perform slower. It then might be up to the individual to accept one or the other, or write in code which decides somewhat arbitrarily, that if the absolute distance to the origin is too large to the relative distances of the points, that one will perform the second algorithm in favor of precision, else they simply preform the first one in favor of speed. This calculation will also take up time, thus one must know what sort of data they will be working with in order to have a good idea of how they wish to proceed. This is one of the unique challenges applied mathematicians face that pure mathematicians need not worry about in their own fields.

4 Speed

4.1 Big O Notation

Now, while we have gone over error and gave an example of what one might do to address errors in floating point arithmetic, the rest of the paper will focus primarily on speed. In order to measure speed, we will have to introduce the "Big O Notation", which many computer scientists will already be familiar with. Big O notation is a way to describe how fast a function grows, something is $O(n)$ say if it increases linearly. As whatever variable you are working with increases to infinity, it starts to become some constant multiple of whatever is inside the O. As an example, let's say we let n describe the number of points we have, and we have a function which finds the bottom-most point, in the case of a tie it then chooses the left-most point out of all the ones with the lowest y-value. This is $O(n)$ as when we add a new point, we must now check one more time. If instead we had to perform some operation two more times for every added data point, it would still be $O(n)$ as this is a constant multiple of something that is $O(n)$.

Note, that there are times when algorithms exist that are say, $O(n^2)$ and $O(n\log(n))$ where the "slower" algorithm, the one that is $O(n^2)$ will actually perform faster than the other one for n less than some incredibly large number. Let's say that something is $O(e^n)$, which grows very very fast, versus something that is say $O(n^3)$. If the algorithm that is $O(e^n)$ actually has a speed like say, $0.0001e^n$, while the $O(n^3)$ is say $1000n^3$, it should be very obvious that for n which aren't very large, that the $O(e^n)$ function performs better than the $O(n^3)$. However, in the limiting case, it also is clear that the $O(e^n)$ function performs worse, take $\lim_{n\to\infty} \frac{0.0001e^n}{1000n^3} = \infty$. When writing code that needs to be very efficient sometimes the programmer will write code which determines when to switch between algorithms. Say we have as before, an algorithm that is $O(n^2)$, and another that is $O(n\log(n))$, but the $O(n^2)$ one performs faster than the $O(n\log(n))$ one for $n < 10$. Then one might write code which uses the first in the case that $n < 10$, but the second when $n \geq 10$. There are however times where this isn't possible, as determining which algorithm is faster might take too much time, or only be known after the problem is solved, or the algorithm has ran.

We can also have functions which are of two or more variables, an example we will see later is of a function for calculating the convex hull, a certain type of polygon, from a set of points. Letting n be the number of points we start with, and h the number of vertices of our final polygon, there exists an algorithm that runs in $O(nh)$ time. As the convex hull can only use points that were initially given, $h \leq n + 1$, meaning in the worse case, this function runs in $O(n^2)$. It is very possible it will run faster, but in order to determine that one would need to know the number of vertices of the final product, which if we could determine, would practically result in us having the convex hull already. Note also, that h is not known immediately, it depends on the output. Algorithms whose time complexity depend on the output are called output-sensitive algorithms, and when dealing with output-sensitive algorithms we will deal with the worst case.

As a final note, the way in which the time it takes to run an algorithm is very subjective to the specific algorithm we examine. For example something of $O(n)$ won't take n seconds,

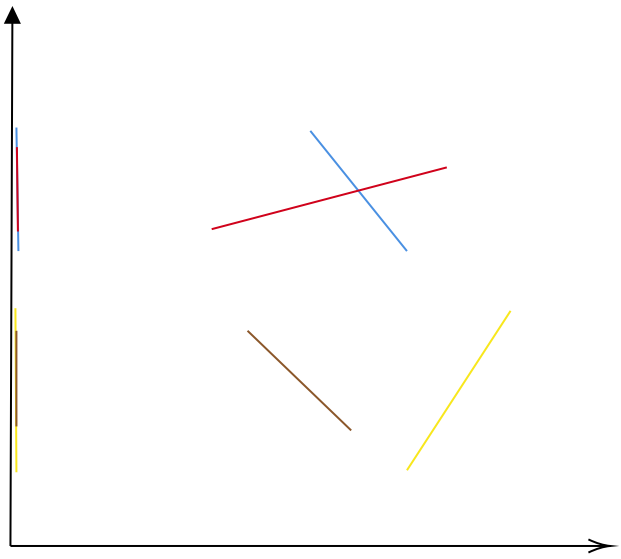
minutes, hours, etc. rather when $n = 5$, it will take 5 times as long as when $n = 1$. For something that is $O(n^2)$, when $n = 5$ it will take $5^2 = 25$ times as long as when $n = 1$. Now, really as these values of n are small this may not be accurate, but it's important to consider that. The way we determine how long something takes might be how many times it has to calculate the intersection of two lines, retrieve and compare values of x or y-coordinates, etc. Thus two algorithms that are say $O(n^2)$ might take wildly different amounts of time to complete for the same values of n.

4.2 A More Efficient Algorithm to Calculate Intersections of Lines

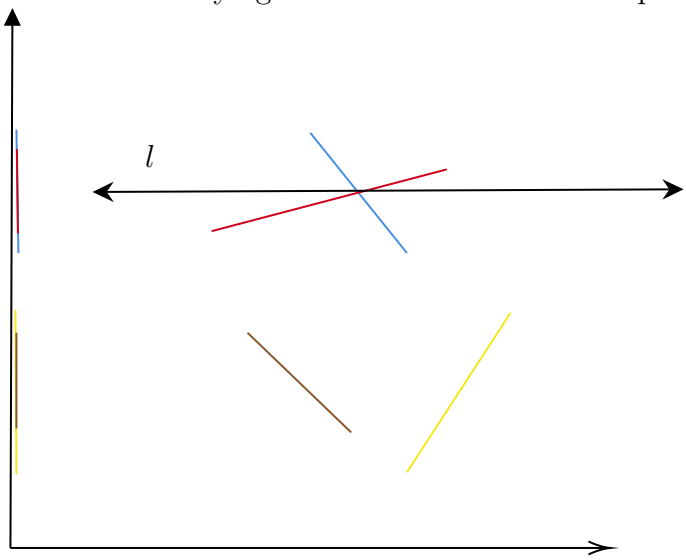
In order to illustrate how the time complexity of algorithms are proven, we will go over an example in the book Computational Geometry: Algorithms and Applications by Berg, Kreveld, Overmars, and Schwarzkopf [3] which describes a way to calculate the intersections of an arbitrary number of line segments. To begin, let's set the stage for the problem. Given n line segments, find all intersection points of the line segments. One could brute-force this, simply take every pair of line segments, and calculate whether they intersect, and if so, report their intersection point. This clearly is $O(n^2)$, as given n line segments, we have n^2 pairs (one can consider the set of segments forming a set S, the pairs are simply elements of $S \times S$ which would have n^2 elements).

Now, if our line segments all intersect each other, this is fine as we would have n^2 intersections. However, this is not always the case, rather one would believe that this is by far the exception. How then, can we develop an algorithm that is faster in most cases, and equal in speed in the worst-case scenario? We want an output-sensitive algorithm, but one that won't be any slower than $O(n^2)$ in the worst case. How can we do this? From your geometric intuition, line segments that are close should have a reasonable chance to intersect, while those that are far away should not be able to intersect unless they come closer together right? Intersection is exactly the case that the distance of two line segments becoming 0, so how can we make this precise, and develop an algorithm that is both faster, but also will not miss any intersections? We now will explore this idea.

Let $S = \{s_1, s_2, \ldots, s_n\}$ be the set of segments we wish to find all intersections of. First, imagine if we squished all the lines down onto the y-axis, we project them all onto the y-axis. If two lines don't intersect here, then they cannot possibly intersect. Intuitively, if the minimum y-value a certain line segment takes is larger than the maximum y-value another line segment takes they cannot possibly intersect.



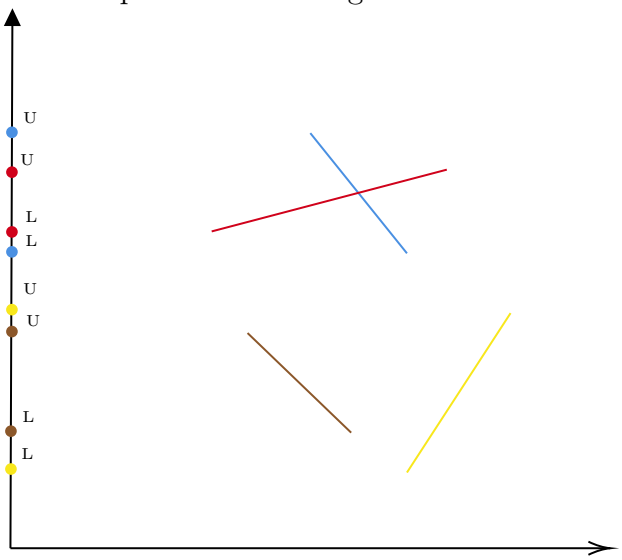
In order to make this precise, we can imagine sliding or sweeping a line l which lies horizontally in the plane downwards and keep track of all the lines which intersect this line at various points as we sweep it down. Only those lines which intersect this *sweep line* at the same time will we test against each other to see if they intersect. As continually dragging this line would be annoying we will define some event points, and only update at these points.



Definition 6. Event Point
Certain points at which an algorithm will update

In this particular case, we let the event points be the top and bottom endpoints of every line segment. Making a list of all the event points, and then ordering them descending by their height, we essentially stack all the event points on top of each other. Then, our algorithm will simply jump from point to point, updating and performing things as we hit these event points. In the case that the event point is the upper point of a line segment, we

will add it to a list of line segments intersecting the line, and in the case the event point is the bottom point of a line segment we will remove it.



We need not calculate any intersections this way, once we order all the event points by their height we need only jump from point to point. This however is not enough, points can be intersecting our sweep line at the same time, but be very far apart horizontally. How can we accommodate for this? Well, at any particular point of our sweep line, the only lines that can intersect each other are the ones that are next to each other can intersect. To reflect this, we also order the segments horizontally, when we hit the top endpoint of a line segment, find where it falls in the order, and test it for intersection against its two neighbors. Later, the adjacency might change, when two points intersect, and when you hit the bottom point of a line, where it is removed from the list. We thus have a new event point, intersections.

This seems like it could possibly be faster, but is this okay? If we only consider adjacent lines how can we be sure that we will find all intersections? Well, in order to simplify this, we will ignore degenerate cases for now, they will be easy to deal with. So, assume that no segment is horizontal, that two segments intersect only in one place (meaning they cannot overlap), and that no three segments intersect at a common point. It is obvious that we cannot possibly miss an intersection where an endpoint of a segment lies on another segment as we will be testing those lines for intersection when we hit the event point corresponding to that endpoint. Thus we must consider whether the intersection of the interiors of line segments will always be found. Here we will quote the proof of a lemma verbatim from the book.

Lemma 1. *Let s_i and s_j be two non-horizontal segments whose interiors intersect in a single point p, and assume there is no third segment passing through p. Then there is an event point above p where s_i and s_j become adjacent and are tested for intersection.*

Proof. Let l be a horizontal line above p. If l is close enough to p then s_i and s_j must be adjacent along l. (To be precise, we should take l such that there is no event point on l, nor in between l and the horizontal line through p.) In other words, there is a position on the

sweep line where s_i and s_j are adjacent. On the other hand, s_i and s_j are not yet adjacent when the algorithm starts, because the sweep line starts above all line segments and the status is empty. Hence there must be an event point q where s_i and s_j become adjacent and are tested for intersection. □

The proof here clearly relies on the fact that no line is horizontal and there is no third line passing through p. If that is the case, very close to p it is clear that s_i and s_j have no line in between them, thus are adjacent. This may not be as rigorous as some pure mathematicians may like, who like proofs built up rigorously from axioms and first principles, but this is often the nature of applied mathematics.

So let's think about the algorithm so far, when we hit an event point which corresponds to the upper endpoint of a line segment, we must test it for intersection with its two neighbors. Say s_i and s_k are adjacent on our sweep line, and we hit the top endpoint of a segment s_j which is between s_i and s_k (with respect to their top endpoints), then we add it between s_i and s_k, and test it for intersection with s_i and s_k. After the upper endpoint has been handled, we move onto the next event point. What do we in the case of an intersection? Well if two lines intersect then they swap places, the one that was to the right is now on the left, and the one on the left is on the right. Let s_j, s_k, s_l, and s_m appear in this order on the sweep line. Then, in the case that s_k and s_l intersect, we will swap the order so now it reads s_j, s_l, s_k, s_m and test s_j and s_l for intersection, as well as s_k and s_m. Note that we may end up calculating the intersection of two lines multiple times. Let's say that s_k and s_m were adjacent at some point before, we will have found their intersection again.

The final case is when we hit the bottom edge of a line segment. Let's say we have s_k, s_l, s_m in this order, then hit s_l's bottom endpoint. Then we remove s_l from the queue, and test s_k and s_m for intersection. In order to continue, we must now define things precisely, and go into time complexity. The following sections will omit some explanation of things from computer science, such as binary search trees. The explanation of such things do not fit the theme of the paper, and can easily be found on the internet.

Define an event queue Q. This stores all of the event points we have. We need an operation to move to the next event, and return it so we can handle the event properly. The event is the one highest one below our sweep line. In the case of a tie, two elements with the same y-coordinate, take the one with the smaller x-coordinate. Thus we go from left to right, implying that for horizontal lines, the left endpoint is its upper endpoint, and the right endpoint is its lower endpoint. The event queue must allow for new things to be inserted, as when we calculate an intersection we need to be able to add it. In addition, two event points can be the same, for example two lines with the same endpoint. It is convenient to simply treat this as one event point, thus we need Q to be able to check if an event point is already present.

Define an order $\prec$ on the event which will represent the order we handle events. If p and q are event points then $p \prec q$, if and only if $p_y > q_y$, or in the case that $p_y = q_y$, then $p_x < q_x$. Store these events in a balanced binary search tree, ordered according to $\prec$. At each event point p in Q, store all segments which have as an upper endpoint p. This information will be needed to handle events. Both of these operations, fetching the next event and inserting

an event take $O(\log m)$ time, where m is the number of events in Q.

Next, we need to know the status of the algorithm. This is the ordered sequence of segments which intersect the sweep line, the thing that tells us what are adjacent. The status structure, T, is used to access the neighbors of a segment s, so that we can test them for intersection with s. We must be able to add and remove segments. As there is a well defined order (by their x-value) we can use a balanced binary search tree as a status structure. On page 25 the author offers an in-depth look into the structure, but we will merely summarize the result. In $O(\log n)$ time we are able to update and find the neighbors, and can easily find the segment immediately to the left or right of a point p on our sweep line, or find the segments which contain p. Our algorithm then, in pseudocode provided by the author, looks like so

Algorithm 1. FindIntersections(S)
Input A set S of line segments in the plane.
Output The set of intersection points among the segments in S, with the associated segments which contain the intersection point.
1. Initialize an empty event queue Q. Next, insert the segment endpoints into Q; when an upper endpoint is inserted, the corresponding segment should be stored with it.
2. Initialize an empty status structure T.
3. **while** Q is not empty
4. **do** Determine the next event point p in Q and delete it.
5. HandleEventPoint(p)

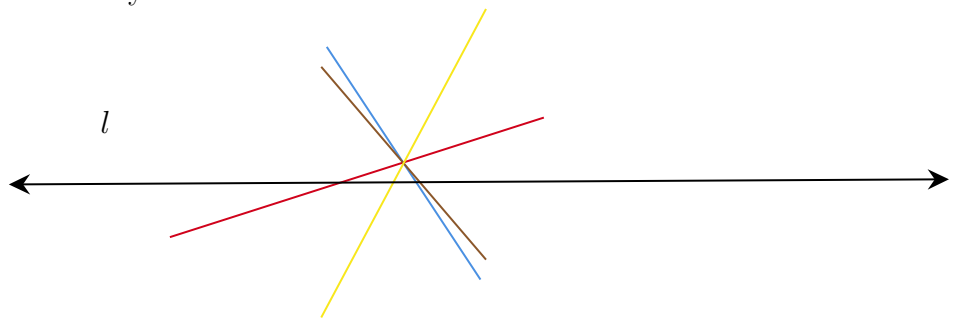
We have already described how to handle event points in our simple case without degeneracies. We will now describe how to handle even the degenerate cases.

Algorithm 2. HandleEventPoint(p)
1. Let $U(p)$ be the set of segments whose upper endpoint is p; these segments are stored with the event point p. (For horizontal segments, the upper endpoint is by definition the left endpoint.)
2. Find all segments stored in T that contain p; they are adjacent in T. Let $L(p)$ denote the subset of segments found whose lower endpoint is p, and let $C(p)$ denote the subset of segments found that contain p in their interior.
3. **if** $L(p) \cup U(p) \cup C(p)$ contains more than one segment
4. **then** Report p as an intersection, together with $L(p), U(p)$, and $C(p)$.
5. Delete the segments in $L(p) \cup C(p)$ from T.
6. Insert the segments in $U(p) \cup C(p)$ into T. The order of the segments in T should correspond to the order in which they are intersected by a sweep line just below p. If there is a horizontal segment, it comes last among all segments containing p.
7. (*Deleting and re-inserting the segments of $C(p)$ reverses their order.*)
8. **if** $U(p) \cup C(p) = \varnothing$
9. **then** Let s_t and s_r be the left and right neighbors of p in T.
10. FindNewEvent(s_l, s_r, p)
11. **else** Let s' be the leftmost segment of $U(p) \cup C(p)$ in T.

12. Let s_l be the left neighbor of s' in T.
13. FindNewEvent(s_l, s', p)
14. Let s'' be the rightmost segment of $U(p) \cup C(p)$ in T.
15. Let s_r be the right neighbor of s'' in T.
16. FindNewEvent(s'', s_r, p).

Note that in lines 8-16 we assume that s_l and s_r exist, if they don't then the corresponding steps should not be performed.

Now why do we go about it this way? Well the first step where one might have questions might be 5. We delete the segments in $L(p)$ as we will go past its lowest point, thus it is no longer needed. We delete points in $C(p)$, as since this is internal it corresponds to the intersection of two lines in their interior, thus they must be flipped. In step 6. since we now allow many lines to intersect at one point, we must determine the new order of the segments in $C(p)$ by looking at how their order is a bit below the intersection, we cannot simply flip the order anymore.



If the if statement in 8. is true, then the point we looked at corresponds only to the lower endpoint of a segment, thus we need only look at the two segments to the left and right, and then test if those two intersect. Else, we either are adding a point since we hit the top of a line, or have an internal intersection, both of which require two sets of segments to be tested against each other, thus we have two calls to FindNewEvent. In order to understand what FindNewEvent does, note that we want to avoid finding intersection points which have already been handled. As all points above our sweep line have been handled, we need only look at points below the line. What then of horizontal lines? Well, as the left-most point is the upper point, and the right-most point the lower, we are interested in points lying to the right of the current event point, as well as those below. So, define *FindNewEvent* as follows,

Algorithm 3. FindNewEvent(s_l, s_r, p)
1. *if* s_l and s_r intersect below the sweep line, or on it and to the right of the current event point p, and the intersection is not yet present in Q
2. *then* Insert the intersection point in Q.

This simply looks only for the intersection points that have a chance of not having already been found, and then inserts it into our queue if it isn't already present. It should be clear that FindIntersections finds only real intersections, but as before, does it find every intersection? To prove this, we will prove a lemma.

Lemma 2. *FindIntersections computes all intersection points and the segments that contain it correctly.*

Proof. The priority of an event is given by whichever point has a larger y-coordinate, and in the case we have a tie, then by that which has a lower x-coordinate. The proof is by induction on the priority of the event points.

Let p be an intersection point, and assume that all points with a higher priority have been calculated correctly. We shall prove that p and all segments which contain it will be calculated correctly. Let $U(p), L(p), C(p)$ be defined as they are in the definition of HandleEventPoint.

Assume p is an endpoint of one or more of the segments. If that is the case, then p is stored as an event point in Q at the start of the algorithm. The segments from $U(p)$ are stored with p at the start, thus they are found. The segments from $L(p)$ and $C(p)$ are stored in T when p is handled, so they will be found in line 2. of HandleEventPoint. Thus, p and all segments involved are determined correctly in the case p is an endpoint of one or more of the segments.

Now, assume p is not the endpoint of any segment. We must show that p is inserted into Q at some point, as in what we proved in Lemma 1 in the case without degeneracies. As all points have p in their interior, order the segments by angle around p, then two neighboring segments s_i and s_j should clearly pop out, two segments which are adjacent by this new ordering. As in the proof of Lemma 2.1, as these two did not start adjacent, there must be an event point with higher priority than p where they become adjacent, thus by induction when q is handled correctly we add p into Q. While the proof of Lemma 2.1 assumed we did not have horizontal lines, but it is easy to adapt the proof. Thus by induction we are done. □

Note the base case is simply the first point which is added which is an endpoint and is handled correctly, thus there is no issue there. Now, how can we calculate the time complexity of this algorithm? Well the answer is that it is output-sensitive. We will prove this as another lemma.

Lemma 3. *The running time of FindIntersections for a set S of n line segments in the plane is $O(n \log n + I \log n)$ where I is the number of intersection points of segments in S.*

Proof. The algorithm starts by constructing the event queue on the endpoints of our segments. As each insertion is $O(\log n)$ and we have $2n$ points to insert this is $O(\log n)$ (note that the 2 does not matter for big O notation). Initializing the status structure takes constant time as initializing it does not depend on anything else. Then the plane sweep begins and all events are handled. The handling of an event has us perform three operations, the event is deleted in Line 4. of FindIntersections, and there can be one or two calls to FindNewEvent. Deletions and insertions are $O(\log n)$ each, we also perform operations on the status structure T which is also $O(\log n)$. The number of operations is linear in $m(p) = |L(p) \cup U(p) \cup C(p)|$ which are involved in the event corresponding to p. If the sum of all $m(p)$ over all event points is m, then the algorithm runs in $O(m \log n)$ time.

Clearly $m = O(n + k)$ where k is the size of the output. To see this, when $m(p) > 1$, we report all the segments involved in the event, and the only time that $m(p) = 1$ is when p is the endpoint of a segment. Thus we have at the least, n points where we handle events, but we also have a variable amount depending on how many intersections we have, which correspond to points in $C(p)$. We thus want to show that $m = O(n + I)$ where I is the number of intersection points. To show this, interpret the set of segments as a planar graph embedded in the plane. Its vertices are the endpoints of segments and intersection points of segments, and the edges are the pieces of segments which connect various vertices. Consider an event point p. It is a vertex of the graph, and $m(p)$ is bounded by the degree of the vertex (the number of edges connected to p). Thus, m is the sum of the degrees of all vertices. Every edge contributes one to the degree of two vertices (its start and endpoint), thus m is bounded by $2n_e$ where n_e is the number of edges. We now seek to bound n_e in terms of n and I. By definition, we can have at most $2n + I$ vertices n_v, this is only obtained when no edges of segments coincide. $2n$ endpoints and I intersection points. It is well known that $n_e = O(n_v)$, but we shall prove it here. Every face is bounded by at least three edges, assuming we have at least three segments. And a single edge can bound at most two faces. Thus, n_f, the number of faces, is less that $2n_e/3$. We now use, without proof, Euler's formula, which states that for any planar graph with n_v vertices, n_e edges, and n_f faces, the following relation holds,

$$n_v - n_e + n_f \geq 2$$

Equality holds if and only if the graph is connected. Plugging in our bounds on n_v and n_f we get that

$$2 \leq (2n + I) - n_e + \frac{2n_e}{3} = (2n + I) - n_e/3$$

Thus, $n_e \leq 6n + 3I - 6$, which means that $m \leq 12n + 6I - 12$, thus our bound holds, as this shows that $m = O(n + I)$. We thus get that the time complexity is $O(m \log n) = O(n \log n + I \log n)$. □

So, we have shown and proven the time complexity of an output-sensitive algorithm, which should perform better than our brute force method when the number of intersections is small relative to the number of line segments we have. In the worst case, we can have $I = n^2$ correct? In that case our algorithm becomes... $O(n^2 \log n)$? As it turns out, our algorithm is actually worse than the other one in the case that the number of intersections is somewhat maximal. We also never addressed when two lines lie on top of the other for the sake of simplicity here. There do exists algorithms which run in $O(n \log n + I)$ time as the author explains on page 41, one of which is by Balaban [4]. This problem of finding the intersections of line segments is one of the fundamental problems in computational geometry, and much work has gone into finding solutions to this that are fast.

4.3 Recap on Speed

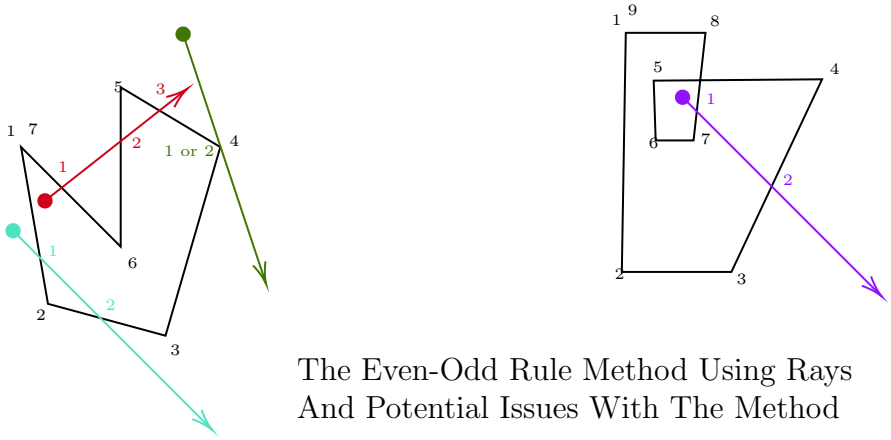
Hopefully, we have illustrated the methodology that is used to prove things in computational geometry, calculate the time complexity of algorithms, etc. While the details of the proof

may be hard to understand at time, hopefully the reader can feel like they have at least a small understanding of how things work in the field of computational geometry.

5 How Can Pure Mathematics Influence Computational Geometry?

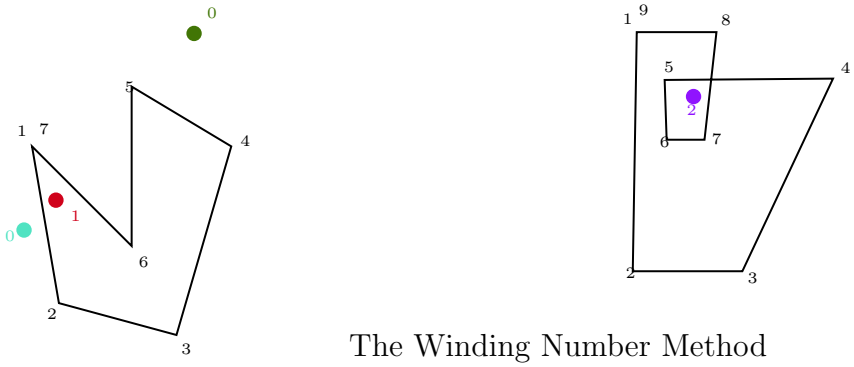
5.1 Point in Polygon Calculation

A classic problem that often occurs is to determine if a point lies within a polygon. One way to determine this is to look at the winding number of the polygon around the point. To the reader who has had a course in complex analysis this should be a familiar concept. The benefit of looking at the winding number is that it does not fail in degenerate cases like the more common method which extends a ray out from the point. The idea there is that if the point is in the polygon, that a ray from the point will intersect the polygon an odd number of times. Once when it leaves, and then if the ray enters the polygon again, then it will have to leave, thus it will enter an even amount of times. If it is outside, either the ray never intersects, or it will intersect an even number of times, when it enters it must leave. This however fails if not handled correctly when the ray intersects a vertex (one must be sure to count it twice), or when the polygon is not simple.



The Even-Odd Rule Method Using Rays
And Potential Issues With The Method

In the case of a general polygon, we can have a polygon wind around a point more than once, a byproduct of the fact that it can intersect itself. Instead, when one calculates the winding number, as long as the winding number is not 0, then the point lies inside the polygon. So, how can we make this rigorous? We will reference the paper by Hormann and Agathos entitled "The point in polygon problem for arbitrary polygons" [5]. The winding number is a way to measure the amount of times a certain curve in the plane winds around a point. It is intimately related to the argument principle in complex analysis, and we will demonstrate a way to calculate it.



The Winding Number Method
(Numbers Next To Points Indicate
The Winding Number)

The winding number $\omega(R, C)$ of a point R with respect to a closed curve C which is given by a function $\gamma(t) = (x(t), y(t))$ for $t \in [a, b]$, with $C(a) = C(b)$ is the number of revolutions made around R while traveling once along C. It is defined for every point R which does not appear in the image of γ, the curve C, and can be calculated by integrating the differential of the angle the point $C(t)$ takes with respect the positive horizontal axis. This is a function $\varphi(t)$. In the case we have a closed curve, which all polygons we work with are, this yields a number of the form $\omega \cdot 2\pi$, with $\omega \in \mathbb{Z}$ denoting the winding number.

Without loss of generality, assume $R = (0, 0)$, which we can do as the winding number is translation invariant, so we just shift the polygon such that the point R is now the origin. This lets us express $\varphi(t)$ as $\arctan(y(t)/x(t))$, and thus

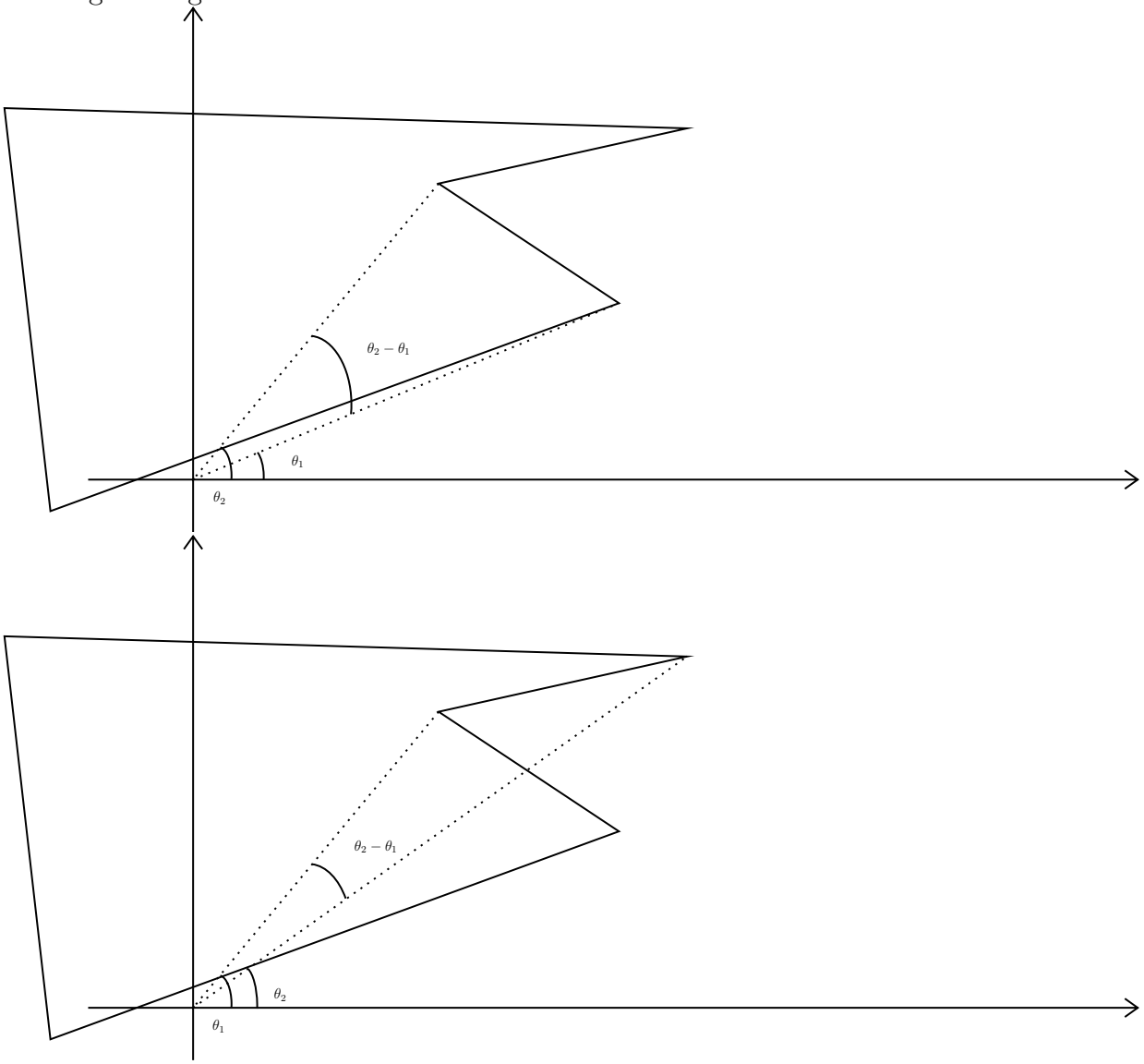
$$\omega(R, C) = \frac{1}{2\pi} \int_a^b \frac{d\varphi}{dt}(t)dt = \frac{1}{2\pi} \int_a^b d\varphi(t) = \frac{1}{2\pi} \int_a^b \frac{y'(t)x(t) - y(t)x'(t)}{x(t)^2 + y(t)^2} dt$$

As we are dealing with a polygon, this integral is easy to calculate. A polygon P is made up of points, call them $p_1, p_2, \ldots, p_n = p_0$ (note this polygon has $n - 1$ vertices), and it is piecewise linear. We can define the curve by $t \mapsto (x_i(t - i), y_i(t - i)),\ t \in [i, i + 1]$, with $(x_i(t), y_i(t)) = tP_{i+1} + (1 - t)P_i$. This is the obvious map, with each interval $[i, i + 1]$ simply mapping the line which goes from P_i to P_{i+1}, which is one edge of the polygon. Thus, we can get the following equations, which we will present without proof. The proof is in Appendix A of the paper.

$$\begin{aligned}
\omega(R, P) &= \frac{1}{2\pi} \sum_{i=0}^{n-1} \int_0^1 \frac{y'(t)x_i(t) - y_i(t)x_i'(t)}{x_i(t)^2 + y_i(t)^2} dt \\
&= \frac{1}{2\pi} \sum_{i=0}^{n-1} \arccos\left(\frac{P_i \cdot P_{i+1}}{\|P_i\| \, \|P_{i+1}\|}\right) \cdot \operatorname{sign} \det \begin{bmatrix} P_i^x & P_{i+1}^x \\ P_i^y & P_{i+1}^y \end{bmatrix} \\
&= \frac{1}{2\pi} \sum_{i=0}^{n-1} \varphi_i
\end{aligned}$$

Where φ_i is the signed angle between the edges of $\overline{RP_i}$ and $\overline{RP_{i+1}}$. This is just the difference of the angle at which P_i and P_{i+1} are relative to the horizontal plane, which one could see

easily from the fundamental theorem of line integrals. When we break this up into each piece which is simply a line, we are looking at $\int_{P_i}^{P_{i+1}} d\varphi(t) = \varphi(P_{i+1}) - \varphi(P_i)$, which is why this is a signed angle.



Now, the second equation there may look nice, however the inverse trig function is costly computationally, and thus there are better ways to make use of the fact that the third equation is simply the difference between the angles of endpoints of all edges of the polygon which are faster, which the paper goes into, but I elect not to go into as it is complicated and not very enlightening. Thus, if the winding number is 0, the polygon does not encompass our point, and if it is an integer $n \neq 0$, then it goes around our point a net n times counterclockwise, if n is positive, and a net $-n$ times clockwise if n is negative, thus as long as the winding number is not 0 our point lies inside the polygon.

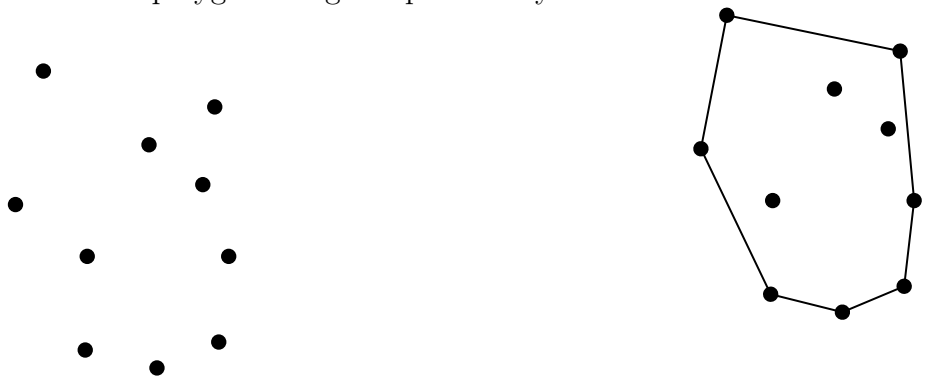
5.2 Why Do We Care?

While it was a brief excursion, hopefully the reader finds it at least somewhat believable that there are real rigorous, pure mathematics that can underlie what computational geometrists do. It is precisely the fact that one can relate a point in polygon calculation to a complex contour integral (the context where the winding number is often discussed) which led me personally to become interested in the field of computational geometry, and hopefully the reader finds it interesting as well.

6 Convex Hull Algorithms

6.1 What is a Convex Hull

The convex hull is a specific type of polygon which is unique to any set of points. Taking any set of points, the convex hull is the smallest convex polygon which encloses all the points (one is also allowed to have the points be on the boundary of the polygon). To do this, you construct the polygon using the points in your set as the vertices.



A Set of Points and Its Associated Convex Hull

As the reader might have already discovered in their own work, convex sets are generally easier to work with than concave sets, and this applies to polygons as well. The convex hull is a classic problem in computational geometry, and fast solutions have been developed. Given a set of points, if one knows what a convex hull is, one could manually construct it with relative ease. The problem with translating this to a computer which cannot see the points, and only knows the x and y values of each points is a problem which we will now explore a bit.

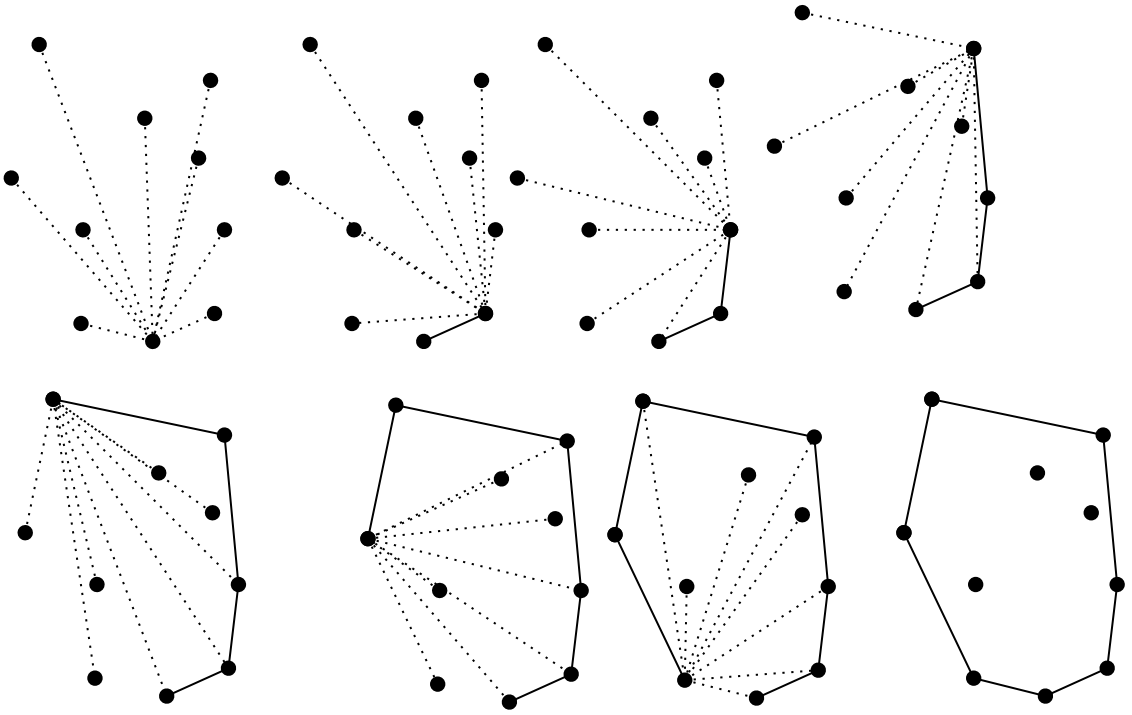
6.2 The Jarvis March Algorithm

The first algorithm we present is often referred to as the Jarvis March Algorithm or Gift Wrapping. It was described by Jarvis in 1973 [6]. The algorithm runs in $O(nh)$ time where n is the number of points we begin with, and h is the number of points in the convex hull. Note, that $h \leq n+1$, thus the algorithm runs in worst case at $O(n^2)$ time, but will perform

better when h is small relative to n. Note, that it is perfectly possible to have a convex hull with $n+1$ points, take the regular n-gon, it will have every point in its convex hull, thus this worst case scenario is very possible.

The actual methodology is very simple, and intuitive. First, we identify the bottom-most point, and in case of a tie, we take the left most point. From this point, it is very obvious that the next point will be above, and so in order to maximize the size (thereby ensuring the polygon remains convex / contains every point), we find the point which has the smallest angle with the positive x-axis, and then take that point to be the next point. From that point on, we have two points, p_1 and p_2. Now to find the point p_{k+2} we find the point p_i which maximizes the angle $\angle p_k p_{k+1} p_i$ and take this p_i to be p_{k+2} until we return to the beginning (this angle cannot be a reflex angle, else we will have constructed a concave polygon, and by virtue of how the algorithm this isn't possible). One can imagine it like this, imagine we have all the points as pegs on a board. We tie a string to the bottom-left most point, then pull the string taut downwards, and then pull it in a large arc going counter clockwise. The string will get caught on other pegs, until we come back and the string meets our initial peg, and we tie the end to the first peg. All the points which the string is touching form the vertices of our convex hull, and they are the ones which are the "most-left" with respect to our counterclockwise motion.

The Jarvis March Method

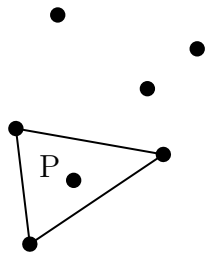


Finding the first point can be done in $O(n)$ time. Then, to find the next point we require to check $n-1$ angles, so this runs in $O(n)$ time. We must do this h times, as we do it once for each point in our convex hull, but in addition we have the $O(n)$ operation to find the first point, but this does not matter in the limit. We thus have a $O(nh)$ algorithm, as we

perform $n-1$ operations h times, plus the initial point finding. The paper actually begins by finding a point outside the convex hull, and then performing this process of finding the first point hit by a swinging radial arm, but the method I outline will work equally well. The advantage of this is that it is simple, and works faster than the $O(n \log n)$ method I will describe next when $h < \log n$, something that has a chance of occurring no matter the size of n.

6.3 The Graham Scan Algorithm

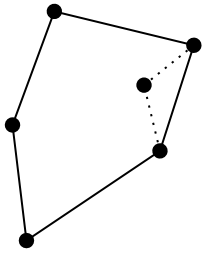
The Graham Scan is an algorithm that runs in $O(n \log n)$ time. It is slightly more complicated than the Jarvis March, and was described by Graham in 1972 [7]. The algorithm proceeds like so. Identify a point which is inside the convex hull. One method to do so is to find 3 points which are not colinear, and then taking the centroid of the triangle formed by the three points. This can be done in $O(n)$ time. Call this point P.



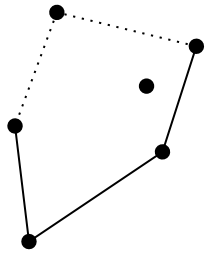
Next, with respect to some arbitrary half line from P (taking a line going right works just fine), express the points in polar form, of the form $r_n e^{i\theta_n}$, this can be done in $O(n)$ times. We can then order them in order by θ_n, which can be done in $O(n \log n)$ time (this operation dominates). Then, if $\theta_k = \theta_{k+1}$, we can delete the shorter one as the smaller can not be on the convex hull (it is further in). Any point with $r_k = 0$ can also be deleted.

At this point, we have a set $S = \{r_1 e^{i\theta_1}, \ldots, r_m e^{i\theta_m}\}$ points, so take three points which are consecutive, say $r_k e^{i\theta_k}, r_{k+1} e^{i\theta_{k+1}}, r_{k+2} e^{i\theta_{k+2}}$, and we have $\theta_k < \theta_{k+1} < \theta_{k+2}$. At this point we will diverge a bit from what the paper describes, as a simpler solution exists. The point is to identify whether or not going from our first point, to the second point, to the third point constitutes a right turn, or a left turn. A left turn indicates all points are in the convex hull, while a right turn indicates that the middle point is interior. We also could have that all points are colinear. In order to determine this, take the vector v_1 from the first point to the second, and v_2 from the second to the third. Simply look at the z-value of their cross product (we must pretend now v_1 and v_2 are 3D vectors, so assume their z-value is 0 to simplify the calculation). If the result is 0, they are colinear, if it is positive, we have a left turn, and if it is negative it is a right turn.

An Example Of A Right Turn



An Example Of A Left Turn



So we must now describe what to do in either case. First, if it is a right turn. As the middle point is not in our convex hull, we delete the point $r_{k+1}e^{i\theta_{k+1}}$ from S, and then repeat this process using the point $r_{k-1}e^{i\theta_{k-1}}, r_k e^{i\theta_k}, r_{k+2}e^{i\theta_{k+2}}$ (reduce all the indices modulo m).

In the case it is a left turn, we essentially just move forward. Repeat the process with the points $r_{k+1}e^{i\theta_{k+1}}, r_{k+2}e^{i\theta_{k+2}}, r_{k+3}e^{i\theta_{k+3}}$.

At each repeated iteration of this, we either remove a point from the set of points which the convex hull might contain, or we move along inside of S. An induction argument will show that this terminates in at most $2m$ steps, where $m \leq n$. This process thus runs in $O(n)$ time.

This algorithm thus runs in $O(n \log n)$ time, which is due to the process of sorting the points as it dominates. A few actual implementation-side things can be done to speed this process up, but it won't eliminate the fact that this algorithm runs in $O(n \log n)$ time. For example, rather than finding the initial point P, much like the Jarvis March method, one can first find the bottom most point, and then in case of tie the left-most point. Then, if we consider the angle all the points make with a line extending out to the right of our bottom-left most point, all the points make an angle less than π. Thus, rather than actually calculating all of the points' angles, and then sorting them, one can sort them by simply observing the value they make on a function which is monotone on $[0, \pi]$, such as the cosine function, which one can calculate very easily using the dot product. It won't change the long run behavior, but in an actual implementation where speed is key it may help.

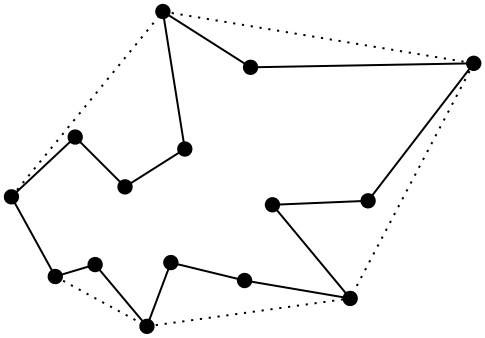
Note, as noted before, that this algorithm can potentially be worse than the Jarvis March when the number of points in the convex hull is small relative to the total number of points.

6.4 Other Efficient Algorithms

Here, we will simply discuss a few other algorithms which exist, but without any explanation of them. There are many other algorithms which run in $O(n \log n)$ time, but they are simply different in their methodology, which might further complicate, or simplify, the implementation for any individual programmer. Instead, I want to take the time to discuss the next class of algorithm, one that runs in $O(n \log h)$ time. The first one comes from a paper entitled, quite comically, "The Ultimate Planar Convex Hull Algorithm?" [8]. A later simplification was described by Chan, and is commonly known as Chan's Algorithm which also runs in $O(n \log h)$ time [9].

6.5 Generalizations to the Convex Hull

The convex hull as we have described is for the planar case, when we look for making a 2D polygon. By means of rotation, this can be done for a 2D polygon in any dimensions, for example in the 3D case one can simply rotate the set of points so they all have the same z-value, generate the convex hull, and then apply an inverse rotation to put everything back into place. There are however, generalizations of the convex hull into higher dimensions, finding the convex set encompassing a set of points in 3-space, or even n-space, in which case you will be generating a polyhedron or a polytope. Another thing the reader may be curious if there exists so-called "concave hull" algorithms, from a set of points can one generate the set which encompasses them all with minimal area?



The "Concave Hull" Along
With The Convex Hull

This kind of thing might be useful when say, generating boundary lines based off of a random sample of points for say, satellite imaging. There is a generalization of the convex hull, called alpha shapes. All convex hulls are alpha shapes, but not all alpha shapes are convex hulls. By use of alpha shapes, there have been algorithms developed to find so-called concave hulls, and we point the reader in the direction of this article [10]

7 Conclusion

In conclusion, computational geometry is an interesting fusion of the at times somewhat ad hoc nature of programming, and the rigor steeped field of mathematics. The methods of proof are different than that of traditional mathematics field, and often require more knowledge of computing, computer science, etc. than the average mathematician may have, but also at times can require a solid basing in theoretical mathematics the average computer scientist may not possess. It is a useful field in applied mathematics, and one which is still very alive with research, one example many mathematicians may know about is the travelling salesman problem. One famous millennium problem is to solve the famous $P = NP$ problem, one way to do so is to come up with a successful algorithm for the travelling salesman which runs in polynomial time. While most researchers believe that $P \neq NP$, and the chances you can find such an algorithm are perhaps zero, if you were to find one, you are entitled to one

million dollars. This problem falls into computational geometry, so perhaps you too want to try tackling the problem?

References

[1] J. Skeet. https://csharpindepth.com/articles/FloatingPoint. [Online; accessed 19-May-2019].

[2] J. R. Shewchuk, "Lecture notes on geometric robustness," tech. rep., 1999.

[3] M. d. Berg, O. Cheong, M. v. Kreveld, and M. Overmars, *Computational Geometry: Algorithms and Applications*. Santa Clara, CA, USA: Springer-Verlag TELOS, 2nd ed. ed., 2000.

[4] I. J. Balaban., "An optimal algorithm for finding segment intersections.," *In Proc. 11th Annu. ACM Sympos. Comput. Geom., pages 211-219*, 1995.

[5] K. Hormann and A. Agathos, "The point in polygon problem for arbitrary polygons," *Computational Geometry*, vol. 20, no. 3, pp. 131 – 144, 2001.

[6] R. Jarvis, "On the identification of the convex hull of a finite set of points in the plane," *Information Processing Letters*, vol. 2, no. 1, pp. 18 – 21, 1973.

[7] R. L. Graham, "An efficient algorithm for determining the convex hull of a finite planar set," *Inf. Process. Lett.*, vol. 1, pp. 132–133, 1972.

[8] D. G. Kirkpatrick and R. Seidel, "The ultimate planar convex hull algorithm," *SIAM J. Comput.*, vol. 15, pp. 287–299, Feb. 1986.

[9] T. M. Chan, "Optimal output-sensitive convex hull algorithms in two and three dimensions," *Discrete & Computational Geometry*, vol. 16, pp. 361–368, Apr 1996.

[10] M. Duckham, L. Kulik, M. Worboys, and A. Galton, "Efficient generation of simple polygons for characterizing the shape of a set of points in the plane," *Pattern Recognition*, vol. 41, pp. 3224–3236, 10 2008.