inzva Algorithm Programme 2018-2019

Bundle 10

Math-3 (Geometry)

Editor
Yusuf Hakan Kalaycı

Reviewer
Hasan Bal

Contents

1 Introduction

In this lecture, we will work on a set of well-studied problems gaining their nature from geometry. We will start with the fundamentals of computational geometry, such as vectors, distances, and vector products. Based on these fundamentals we will define our first primitives such as area calculation methods, distance methods, and intersection methods. With the help of these primitives we will study the following:

- Inclusion Problem: whether a point is inside a polygon or not.
- Convex Hull: finding the minimum convex polygon containing all given points.
- Rotating Calipers Method: a tool to find the farthest pair of points.
- Closest Pair: finding the closest pair of given n points.
- Applications of Line-Sweep Method: a very useful technique for approaching many geometry problems.

The reason why we keep Vector Calculus section sufficiently long is that understanding the nature of the space helps to understand many problems. It is strongly recommended to think about exercises.

Lastly, Section 2 is prepared using [1] and it closely follows that reference. Section 3,4,5,6 and 7 are prepared using [4] and implementations of the algorithms given in these sections are taken from this website. Also, this reference consists of efficient implementations of many methods and their detailed discussions. Section 8 is prepared using [8]; one might check this reference to see detailed discussions of many applications. The two methods mentioned in Section 9 are designed using [9] and [11] respectively.

2 Vector Calculus and Analytic Geometry

2.1 The Euclidean 3-space or $\mathbb{R}^3$

Definition 2.1 (The Euclidean 3-space)**.** *The Euclidean 3-space denoted by* $\mathbb{R}^3$ *is the set*

$$\{(x, y, z) | x, y, z \in \mathbb{R}\}$$

To specify the location of a point in $\mathbb{R}^3$ geometrically, we use the right-handed rectangular coordinate system in which three mutually perpendicular coordinate axes meet at the origin. It is common to use the x and y axes to represent the horizontal coordinate plane and the z-axis for the vertical height.

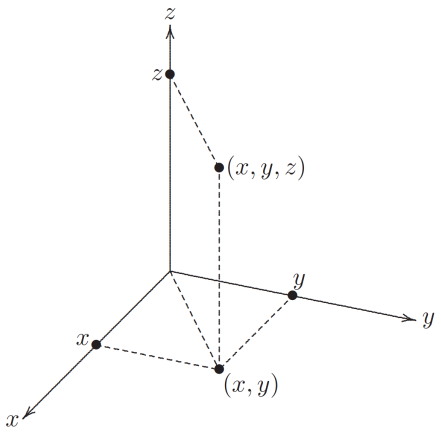


Figure 1: Representation of points in R^3

We usually denote a point P with coordinates (x, y, z) by $P(x, y, z)$. The distance $d(P_1, P_2)$ between two points $P_1(x_1, y_1, z_1)$ and $P_2(x_2, y_2, z_2)$ is given by

$$d(P_1, P_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$$

The following definition and exercise are out of our scope but they can still be inspiring. The definition involves an abstract notion of our euclidean geometry by [3].

Definition 2.2 (Metric Space)**.** *A metric space is ordered pair (X, d) where X is a set and d is a metric(distance measure) on X such that for any $x, y, z \in X$, the following holds*

1. $d(x, y) \geq 0$ *(non-negativity)*
2. $d(x, y) = 0 \iff x = y$
3. $d(x, y) = d(y, x)$ *(symmetry)*
4. $d(x, z) \leq d(x, y) + d(y, z)$ *(sub-additivity or triangle inequality)*

We are challenging a curious reader with the following exercise.

Exercise 2.1. *Show that $\mathbb{R}^3$ is a metric space with euclidean metric.*

An equation in x, y and, z describes a surface in $\mathbb{R}^3$. Let us look at some standard examples of surfaces.

Example 2.1. *(a) $z = 3$ is the equation of a horizontal plane at level 3 above the xy-plane. (b) $y = 2$ is the equation of a vertical plane parallel to the xz-coordinate plane. Every point of this plane has y coordinate equal to 2. (c) Similarly $x = 2$ is the eqaution of a vertical plane parallel to the yz-coordinate plane.*

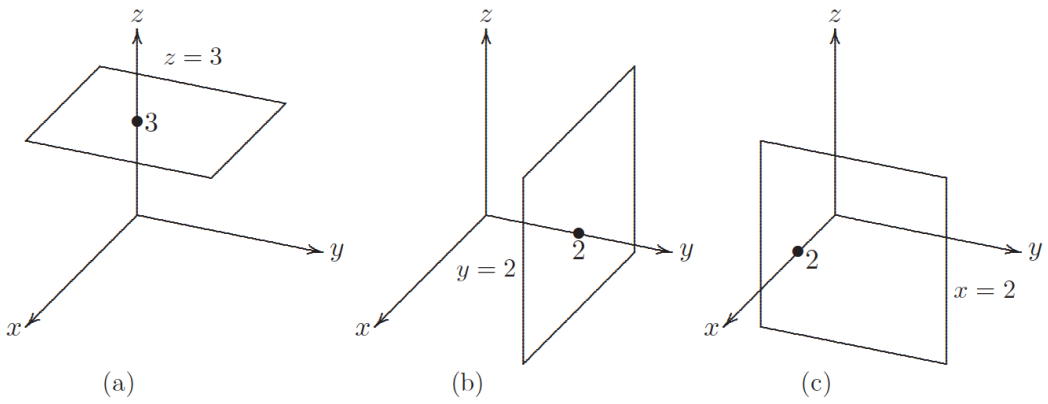


Figure 2: Planes perpendicular to standard basis

Example 2.2. *An equation of a sphere with centre $O(a, b, c)$ and radius r is !!!!!!!!*

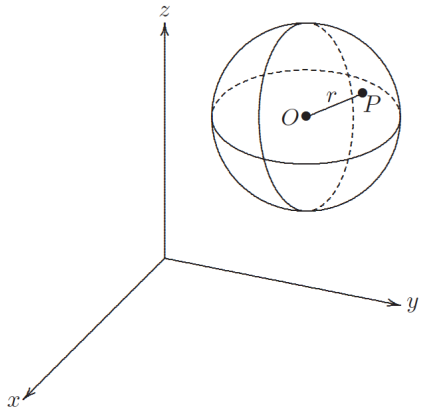


Figure 3: An example of surface in R^3

2.2 Vectors

A 3-dimensional vector is an ordered triple $\boldsymbol{a} = (a_1, a_2, a_3)$ of reals. We will call a_1, a_2, a_3 as components of $\boldsymbol{a}$. Sometimes we can use $\boldsymbol{PQ}$ to denote a vector from a point P to a point Q. In $\mathbb{R}^3$, there are some special vectors which will have special names and can be listed as

- $\boldsymbol{i} = (1, 0, 0)$
- $\boldsymbol{j} = (0, 1, 0)$
- $\boldsymbol{k} = (0, 0, 1)$

These vectors $\boldsymbol{i}, \boldsymbol{j}$ and $\boldsymbol{k}$ consist of our standard basis vectors. Before we show the importance of

those vectors, let's define our addition and multiplication operators which enable us to work on a vector space(what is it?). Vector addition of two vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ is defined as the entrywise addition of their components. Or more formally,

$$\boldsymbol{u} + \boldsymbol{v} = (u_1 + v_1, u_2 + v_2, u_3 + v_3).$$

Multiplication of a vector and a scalar is defined as

$$\lambda \boldsymbol{v} = (\lambda v_1, \lambda v_2, \lambda v_3)$$

Then using definitions given above we can decompose any vector as a linear combination (what is it?) of our standard basis vectors. Formally, $\boldsymbol{v} = v_1\boldsymbol{i} + v_2\boldsymbol{j} + v_3\boldsymbol{k}$ for all $\boldsymbol{v} \in \mathbb{R}^3$. Lastly, we will denote all zero vector as $\mathbf{0} = (0, 0, 0)$ and we will use shorthand notation of $-\boldsymbol{v}$ to represent $-1\boldsymbol{v}$.

As we defined a metric in the previous section, defining a way of measuring the length or magnitude of a vector will be important for our future work. We will define the norm $|\cdot|$ function as following.

$$|\boldsymbol{v}| = \sqrt{v_1^2 + v_2^2 + v_3^2}$$

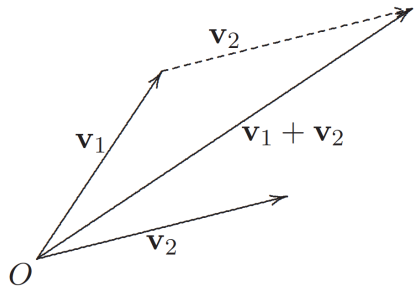


Figure 4: Addition of two vectors

The following exercise is for curious readers.

Exercise 2.2. *The set of all position vectors in $\mathbb{R}^3$ forms a normed vector space over R with norm $||\dot{|}$. (Please see the definition of vector space and normed vector space)*

However, proving the next propositions are exercises for all readers because it is important to understand and internalize the behavior of our Euclidean 3-space.

Proposition 2.1 (Properties of vectors)**.** *Let $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c} \in \mathbb{R}^3$ and $\alpha, \beta \in \mathbb{R}$ then*

1. $\boldsymbol{a} + \boldsymbol{b} = \boldsymbol{b} + \boldsymbol{a}$
2. $\boldsymbol{a} + (\boldsymbol{b} + \boldsymbol{c}) = (\boldsymbol{a} + \boldsymbol{b}) + \boldsymbol{c}$
3. $\boldsymbol{a} + \mathbf{0} = \boldsymbol{a}$
4. $\boldsymbol{a} + -\boldsymbol{a} = \mathbf{0}$

5. $\alpha(\boldsymbol{a}+\boldsymbol{b}) = \alpha\boldsymbol{a} + \alpha\boldsymbol{b}$

6. $\alpha\boldsymbol{a} = \boldsymbol{a}\alpha$

7. $(\alpha+\beta)\boldsymbol{a} = \alpha\boldsymbol{a} + \beta\boldsymbol{a}$

8. $(\alpha\beta)\boldsymbol{a} = \alpha(\beta\boldsymbol{a})$

9. $1\boldsymbol{a} = \boldsymbol{a}$

10. $|\alpha\boldsymbol{a}| = |\alpha||\boldsymbol{a}|$

Exercise 2.3. *Prove the Proposition 2.1 .*

Proposition 2.2. *Prove the triangle inequality* $|v_1 + v_2| \leq |v_1| + |v_2|$ *for all* $v_1, v_2 \in \mathbb{R}^3$.

Exercise 2.4. *Prove the Proposition 2.2 .*

2.3 Dot Product

Until now, we have discussed only the multiplication of a vector and a scalar. However, we can also define the multiplication of two vectors. A natural way to define a kind of vector product is following.

Definition 2.3 (Dot Product)**.** *Let* $\boldsymbol{a}, \boldsymbol{b}$ *be two vectors in* $\mathbb{R}^3$*. The dot product (or sometimes called scalar product) of* $\boldsymbol{a}$ *and* $\boldsymbol{b}$ *is the number* $a \cdot b = a_1b_1 + a_2b_2 + a_3b_3$*. It is sometimes denoted by* $\langle \boldsymbol{a}, \boldsymbol{b} \rangle$*.*

Example 2.3. *Let* $\boldsymbol{a} = (1, 2, 3)$ *and* $\boldsymbol{b} = (-1, 0, -1)$*. Then we can compute dot product of these vectors as*

$$\boldsymbol{a} \cdot \boldsymbol{b} = (1)(-1) + (2)(0) + (3)(-1) = -4.$$

We can state the properties of the dot product as the following proposition.

Proposition 2.3 (Properties of the Dot Product)**.** *Let* $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c} \in \mathbb{R}^3$*, then*

1. $\boldsymbol{a} \cdot \boldsymbol{a} = |\boldsymbol{a}|^2$

2. $\boldsymbol{a} \cdot \boldsymbol{b} = \boldsymbol{b} \cdot \boldsymbol{a}$

3. $\boldsymbol{a} \cdot (\boldsymbol{b} + \boldsymbol{c}) = \boldsymbol{a} \cdot \boldsymbol{b} + \boldsymbol{a} \cdot \boldsymbol{c}$

4. $(\alpha\boldsymbol{a}) \cdot \boldsymbol{b} = \alpha(\boldsymbol{a} \cdot \boldsymbol{b}) = \boldsymbol{a} \cdot (\alpha\boldsymbol{b})$

5. $\boldsymbol{0} \cdot \boldsymbol{a} = \boldsymbol{0}$

Exercise 2.5. *Prove the Proposition 2.3 .*

Following exercise and theorem are for REALLY curious students.

Proposition 2.4. *Let $\boldsymbol{a}$ be a vector in $\mathbb{R}^3$. Then define the function $f : \mathbb{R}^3 \to \mathbb{R}$ as $f(\boldsymbol{x}) = \boldsymbol{x} \cdot \boldsymbol{a}$. Then f is linear(what?).*

Exercise 2.6. *Prove the Proposition 2.4 .*

Okay! It was not that surprising. What about the next statement?

Theorem 2.1 (Riesz representation theorem)**.** *Let $f : \mathbb{R}^3 \to \mathbb{R}$ be any linear (what?) and continuous (what?) function. Then, there exists a unique $\boldsymbol{v} \in \mathbb{R}^3$ such that*

$$f(\boldsymbol{x}) = \boldsymbol{x} \cdot \boldsymbol{v}$$

Now we can continue to our main discussion and obtain the main result of this section.

Theorem 2.2. *If θ is the angle between the vectors $\boldsymbol{a}$ and $\boldsymbol{b}$, then*

$$\boldsymbol{a} \cdot \boldsymbol{b} = |\boldsymbol{a}| \cdot |\boldsymbol{b}| \cos\theta, \quad 0 \leq \theta \leq \pi.$$

Proof. Let $\boldsymbol{OA} = \boldsymbol{a}$ and $\boldsymbol{OB} = \boldsymbol{b}$, where O is the origin and $\theta = \angle AOB$.

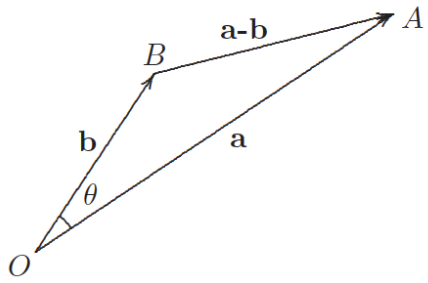


Figure 5: Vector subtraction

Applying cosine rule to $\triangle OAB$, we have

$$|\boldsymbol{a} - \boldsymbol{b}|^2 = |\boldsymbol{a}|^2 + |\boldsymbol{b}|^2 - 2|\boldsymbol{a}| \cdot |\boldsymbol{b}| \cos\theta$$

Also, we know from the properties of the dot product that (why?)

$$|\boldsymbol{a} - \boldsymbol{b}|^2 = (\boldsymbol{a} - \boldsymbol{b}) \cdot (\boldsymbol{a} - \boldsymbol{b}) = |\boldsymbol{a}|^2 + |\boldsymbol{b}|^2 - 2\boldsymbol{a} \cdot \boldsymbol{b}$$

Therefore, $\boldsymbol{a} \cdot \boldsymbol{b} = |\boldsymbol{a}| \cdot |\boldsymbol{b}| \cos\theta$ or

$$\cos\theta = \frac{\boldsymbol{a} \cdot \boldsymbol{b}}{|\boldsymbol{a}| \cdot |\boldsymbol{b}|}$$

□

Two vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ are said to be orhogonal or perpendicular if the angle between them is 90°. In other words,

$$\boldsymbol{a} \text{ and } \boldsymbol{b} \text{ are orthogonal } \iff \boldsymbol{a} \cdot \boldsymbol{b} = 0 \text{ (why?)}$$

.

Now based on this theorem, we will discuss the projection of a vector along another vector. Let $\boldsymbol{a}$ and $\boldsymbol{b}$ be two vectors in $\mathbb{R}^3$. Let's represent $\boldsymbol{a}$ as $\boldsymbol{PQ}$ and $\boldsymbol{b}$ as $\boldsymbol{PR}$.

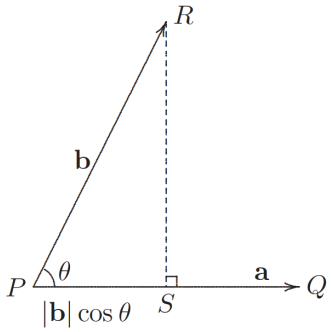


Figure 6: Projection of a vector to another one

Then

$$|\boldsymbol{b}| \cos\theta = \frac{\boldsymbol{a} \cdot \boldsymbol{b}}{|\boldsymbol{a}|} = \frac{\boldsymbol{a}}{|\boldsymbol{a}|} \cdot \boldsymbol{b}.$$

Definition 2.4. *Let $\boldsymbol{a}$ and $\boldsymbol{b}$ be two vectors in $\mathbb{R}^3$.*

1. *The scalar projection of $\boldsymbol{b}$ onto $\boldsymbol{a}$ is $|\boldsymbol{b}| \cos\theta = \frac{\boldsymbol{a}}{|\boldsymbol{a}|} \cdot \boldsymbol{b}$.*

2. *The vector projection of $\boldsymbol{b}$ onto $\boldsymbol{a}$ is $\left(\frac{\boldsymbol{a}}{|\boldsymbol{a}|} \cdot \boldsymbol{b}\right) \frac{\boldsymbol{a}}{|\boldsymbol{a}|} = \frac{\boldsymbol{a} \cdot \boldsymbol{b}}{|\boldsymbol{a}|^2} \boldsymbol{a}$.*

Note that the scalar projection is negative if $\theta > 90°$. Moreover, $\boldsymbol{SR} = \boldsymbol{PR} - \boldsymbol{PS} = \boldsymbol{b} - \frac{\boldsymbol{a} \cdot \boldsymbol{b}}{|\boldsymbol{a}|^2} \boldsymbol{a}$. Thus the distance from R to the line PQ is given by

$$|\boldsymbol{RS}| = \left| \boldsymbol{b} - \frac{\boldsymbol{a} \cdot \boldsymbol{b}}{|\boldsymbol{a}|^2} \boldsymbol{a} \right|$$

Exercise 2.7. *Prove the Cauchy-Schwarz inequality: $|\boldsymbol{a} \cdot \boldsymbol{b}| \leq |\boldsymbol{a}||\boldsymbol{b}|$. Also, determine when the equality holds.*

2.4 The Cross Product

Definition 2.5. *Let $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{R}^3$, then the cross product (or sometimes called vector product) of $\boldsymbol{a}$ and $\boldsymbol{b}$ is*

$$\begin{aligned}\boldsymbol{a} \times \boldsymbol{b} &= (a_2b_3 - a_3b_2, a_3b_1 - a_1b_3, a_1b_2 - a_2b_1) \\ &= \begin{vmatrix} \boldsymbol{i} & \boldsymbol{j} & \boldsymbol{k} \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{vmatrix} \\ &= \begin{vmatrix} a_2 & a_3 \\ b_2 & b_3 \end{vmatrix} \boldsymbol{i} - \begin{vmatrix} a_1 & a_3 \\ b_1 & b_3 \end{vmatrix} \boldsymbol{j} + \begin{vmatrix} a_1 & a_2 \\ b_1 & b_2 \end{vmatrix} \boldsymbol{k}\end{aligned}$$

Exercise 2.8. *Let $\boldsymbol{a} = (1, 3, 4)$ and $\boldsymbol{b} = (2, 7, -5)$. Find $\boldsymbol{a} \times \boldsymbol{b}$.*

Theorem 2.3. *Let $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c} \in \mathbb{R}^3$. Then*

$$\boldsymbol{a} \cdot (\boldsymbol{b} \times \boldsymbol{c}) = \begin{vmatrix} a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \\ c_1 & c_2 & c_3 \end{vmatrix}$$

Exercise 2.9. *Prove the Theorem 2.3 .*

Corollary 2.1. *$\boldsymbol{b} \times \boldsymbol{c}$ is perpendicular to both $\boldsymbol{b}$ and $\boldsymbol{c}$.*

Proof. Use previous theorem and singularity(what?) of the matrix and conclude that the determinant is 0. □

Next result is the main result of this section.

Theorem 2.4. *If θ is the angle between $\boldsymbol{a}$ and $\boldsymbol{b}$, $0 \leq \theta \leq \pi$, then $|\boldsymbol{a} \times \boldsymbol{b}| = |\boldsymbol{a}||\boldsymbol{b}| \sin\theta$.*

Proof. First we need the following identity

$$(a_2b_3 - a_3b_2)^2 + (a_3b_1 - a_1b_3)^2 + (a_1b_2 - a_2b_1)^2 = (a_1^2 + a_2^2 + a_3^2)(b_2^1 + b_2^2 + b_2^3) - (a_1b_1 + a_2b_2 + a_3b_3)^2$$

which can be easily verified by direct simplification of both sides. Using this identity, we have

$$\begin{aligned}|\boldsymbol{a} \times \boldsymbol{b}|^2 &= (a_2b_3 - a_3b_2)^2 + (a_3b_1 - a_1b_3)^2 + (a_1b_2 - a_2b_1)^2 \\ &= (a_1^2 + a_2^2 + a_3^2)(b_2^1 + b_2^2 + b_2^3) - (a_1b_1 + a_2b_2 + a_3b_3)^2 \\ &= |\boldsymbol{a}|^2|\boldsymbol{b}|^2 - (\boldsymbol{a} - \boldsymbol{b})^2 \\ &= |\boldsymbol{a}|^2|\boldsymbol{b}|^2 - |\boldsymbol{a}|^2|\boldsymbol{b}|^2 \cos^2\theta \\ &= |\boldsymbol{a}|^2|\boldsymbol{b}|^2 \sin^2\theta\end{aligned}$$

Since $0 \leq \theta \leq \pi$, $\sin \geq 0$, we have $|\boldsymbol{a} \times \boldsymbol{b}| = |\boldsymbol{a}||\boldsymbol{b}| \sin\theta$. □

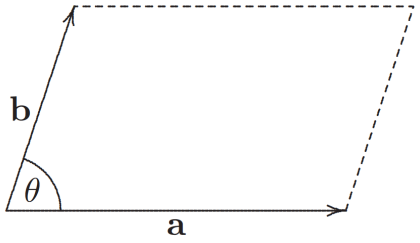


Figure 7: Parallelogram lying between two vectors

It follows from this result that $|\boldsymbol{a} \times \boldsymbol{b}| = |\boldsymbol{a}||\boldsymbol{b}| \sin\theta$ is the area of the parallelogram determined by a and b.

$\boldsymbol{a} \times \boldsymbol{b}$ is a vector perpendicular to the plane spanned by $\boldsymbol{a}$ and $\boldsymbol{b}$ with magnitude $|\boldsymbol{a}||\boldsymbol{b}| \sin\theta$, where $0 \leq \theta \leq \pi$ is the angle between $\boldsymbol{a}$ and $\boldsymbol{b}$.

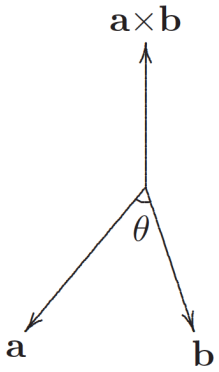


Figure 8: Cross product of two vectors in R^3

There are two possible choices of such a vector. The choice is determined by the right-hand rule: $\boldsymbol{a} \times \boldsymbol{b}$ is directed so that a right-hand rotation about $\boldsymbol{a} \times \boldsymbol{b}$ through an angle θ will carry $\boldsymbol{a}$ to the direction of $\boldsymbol{b}$.

Proposition 2.5 (Properties of the Cross Product)**.** *Let $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}$ be three vectors in $\mathbb{R}^3$ and $\alpha \in \mathbb{R}$.*

1. $\boldsymbol{a} \times \boldsymbol{b} = -\boldsymbol{b} \times \boldsymbol{a}$
2. $(\alpha\boldsymbol{a}) \times \boldsymbol{b} = \alpha(\boldsymbol{a} \times \boldsymbol{b}) = \boldsymbol{a} \times (\alpha\boldsymbol{b})$
3. $\boldsymbol{a} \times (\boldsymbol{b} + \boldsymbol{c}) = \boldsymbol{a} \times \boldsymbol{b} + \boldsymbol{a} \times \boldsymbol{c}$
4. $(\boldsymbol{a} + \boldsymbol{b}) \times \boldsymbol{c} = \boldsymbol{a} \times \boldsymbol{c} + \boldsymbol{b} \times \boldsymbol{c}$
5. $\boldsymbol{a} \cdot (\boldsymbol{b} \times \boldsymbol{c}) = (\boldsymbol{a} \times \boldsymbol{b}) \cdot \boldsymbol{c}$

6. $\boldsymbol{a} \times (\boldsymbol{b} \times \boldsymbol{c}) = (\boldsymbol{a} \cdot \boldsymbol{c})\boldsymbol{b} - (\boldsymbol{a} \cdot \boldsymbol{b})\boldsymbol{c}$

Exercise 2.10. *Prove the Proposition 2.5 .*

3 Area Calculation

3.1 Area of Triangle

We already saw that the cross product of two vectors gives us the area of the parallelogram determined by those vectors. Dividing that quantity by 2 will give us the area of the triangle that lies between those vectors.

As we discussed earlier, we know that the cross product of two vectors can be positive or negative. We also learned that this sign can be determined by the right-hand rule. Now let's see how we can use this property of cross product in $\mathbb{R}^2$ to determine the positions of two vectors in comparison to each other or relation between the orientation of the points.

Let $\boldsymbol{v}$ and $\boldsymbol{v}$ be two vectors in $\mathbb{R}^2$. Then putting the definition of cross product will give us

$$\boldsymbol{u} \times \boldsymbol{v} = \begin{vmatrix} u_2 & u_3 \\ v_2 & v_3 \end{vmatrix} \boldsymbol{i} - \begin{vmatrix} u_1 & u_3 \\ v_1 & v_3 \end{vmatrix} \boldsymbol{j} + \begin{vmatrix} u_1 & u_2 \\ v_1 & v_2 \end{vmatrix} \boldsymbol{k}$$

Since $u_3 = v_3 = 0$, first two determinant values will be zero and the cross product will have only $\boldsymbol{k}$ component. Now, let's denote the $\boldsymbol{k}$ component of that cross product as $(\boldsymbol{u} \times \boldsymbol{v})_k$. Then,

$$(\boldsymbol{u} \times \boldsymbol{v})_k = \begin{cases} \textit{negative} & \text{if } v \text{ lies in left hand side of the } u \\ 0 & \text{if } u \text{ and } v \text{ lies in the same line} \\ \textit{positive} & \text{if } v \text{ lies in right hand side of the } u \end{cases}$$

Then we can also interpret this result for determining the orientation of three ordered points in $\mathbb{R}^2$. Let P, Q, R be points in $\mathbb{R}^2$ then

$$(\boldsymbol{PQ} \times \boldsymbol{PR})_k = \begin{cases} \textit{negative} & \text{if } \boldsymbol{PR} \text{ lies in left hand side of the } \boldsymbol{PQ} \\ 0 & \text{if } \boldsymbol{PQ} \text{ and } \boldsymbol{PR} \text{ lies in the same line} \\ \textit{positive} & \text{if } \boldsymbol{PR} \text{ lies in right hand side of the } \boldsymbol{PQ} \end{cases}$$

or equivalently

$$(\boldsymbol{PQ} \times \boldsymbol{PR})_k = \begin{cases} negative & \text{P,Q and R in counter-clockwise order} \\ 0 & \text{P,Q and R lies in the same line} \\ positive & \text{P,Q and R in clockwise order} \end{cases}$$

We already discussed that $\frac{|\boldsymbol{PQ}\times\boldsymbol{PR}|}{2} = \frac{|(\boldsymbol{PQ}\times\boldsymbol{PR})_k|}{2} = |A(\triangle PQR)|$ and now, we will define the $A(\triangle PQR) := \frac{(\boldsymbol{PQ}\times\boldsymbol{PR})_k}{2}$ as the signed area of the triangle (see that it is scalar). While the sign is positive if the points are in counter-clockwise order, it is negative if the points are in clockwise order.

The following implementation is taken from [4].

```
1  // Assume Point is a class with two properties of double x and y
2  inline double signedTriangleArea( Point P0, Point P1, Point P2 )
3  {
4     return ( (P1.x - P0.x) * (P2.y - P0.y)
5           - (P2.x - P0.x) * (P1.y - P0.y) )/2;
6  }
```

3.2 Area of a Simple Polygon

Based on a signed area of the triangle we will build our area calculation algorithm of any simple polygon (i.e. ones without self-intersection).

Let Ω be a simple polygon in $\mathbb{R}^2$ whose vertices are defined by $V_i = (x_i, y_i)$ for $i = 0, 1 \cdots n$, and define $V_0 = V_n$ and let P be any point. For each edge from V_i to V_{i+1} of Ω $\triangle_i = \triangle PV_iV_{i+1}$ where $i = 0, 1, \cdots n_1$ forms a triangle. Now we can calculate the signed area of the polygon Ω by the following equation

$$A(\Omega) = \sum_{i=0}^{n-1} A(\triangle_i) \text{ where } \triangle_i = \triangle PV_iV_{i+1}$$

Notice that for a counterclockwise oriented polygon, when the point P is on the "inside" left side of an edge $\boldsymbol{V_iV_{i+1}}$, the area of $\triangle_i$ is positive; whereas, when P is on the "outside" right side of an edge $\boldsymbol{V_iV_{i+1}}$, $\triangle_i$ has a negative area. If instead the polygon is oriented clockwise, then the signs are reversed, and inside triangles become negative.

For example, in the Figure 9, the triangles $\triangle_2$ and $\triangle_{n-1}$ have positive area, and contribute positively to the total area of polygon Ω. However, as one can see, only part of $\triangle_2$ and $\triangle_{n-1}$ are actually inside Ω and there is a part of each triangle that is also exterior. On the other hand, the triangles $\triangle_0$ and $\triangle_1$ have negative area, and this cancels out the exterior excesses of positive area triangles. In the final analysis, the exterior areas all get canceled, and one is left with exactly the area of the polygon Ω.

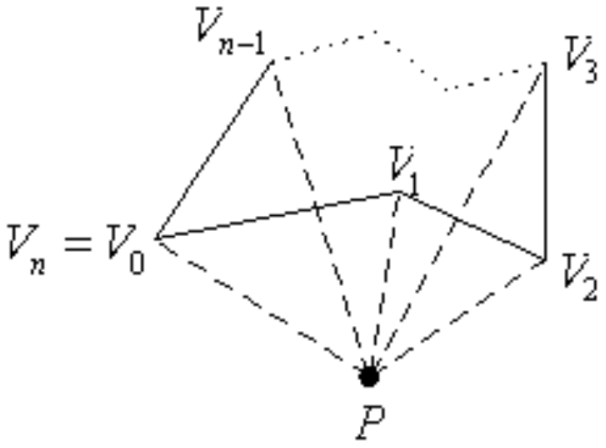


Figure 9: Illustration of polygon area calculation.

One can make the formula more explicit by picking a specific point P and expanding the terms. By selecting $P = (0, 0)$, the area formula of each triangle reduces to $2A(\triangle_i) = (x_i y_{i+1} - x_{i+1} y_i)$. This yields:

$$\begin{aligned} 2A(\Omega) &= \sum_{i=0}^{n-1} (x_i y_{i+1} - x_{i+1} y_i) \\ &= \sum_{i=0}^{n-1} (x_i + x_{i+1})(y_{i+1} - y_i) \\ &= \sum_{i=0}^{n} x_i (y_{i+1} - y_i) \end{aligned}$$

where $V_i = (x_i, y_i)$, with $i (mod\ n)$. (Think about the equations given above!).

The following implementation is taken from [4].

```
1  // area2D_Polygon(): compute the area of a 2D polygon
2  // Input: int n = the number of vertices in the polygon
3  // Point* V = an array of n+1 vertex points with V[n]=V[0]
4  // Return: the (double) area of the polygon
5  double area2D_Polygon( int n, Point* V )
6  {
7     double area = 0;
8     int i, j, k; // indices
9
10    if (n < 3) return 0; // a degenerate polygon
11
12    for (i=1, j=2, k=0; i<n; i++, j++, k++) {
13       area += V[i].x * (V[j].y - V[k].y);
```

```
14      }
15      area += V[n].x * (V[1].y - V[n-1].y); // wrap-around term
16      return area / 2.0;
17  }
```

4 Lines and Planes

4.1 Lines

In space, we can represent lines in multiple ways. As you know, we can represent lines with linear equations. Although linear equations represent lines in 2D, the single linear equation represents a plane in 3D. In addition to linear equations, we can express lines with a point and direction vector. Here is the summary of the expressions with their names and usage:

Type	Equation	Usage
Explicit 2D	$y = f(x) = mx + b$	a non-vertical 2D line
Implicit 2D	$f(x, y) = ax + by + c = 0$	any 2D line
Parametric	$P(t) = P_0 + t\boldsymbol{v_L}$	any line in any dimension

The equivalence of these representations and the conversion between them can be derived easily. However, for this course, we will use the parametric representation and we will discuss distance calculations based on this type of expressions.

We already discussed that the cross product of two vectors gives us the area of the parallelogram that lies between these two vectors. Now, we will use this notion to derive a formula to calculate the distance between a point and a line.

Firstly we will discuss 2D case, let P_0 and P_1 be any two distinct points on a line L and P be any point from which we want to calculate distance to line L. Now, set two vectors $\boldsymbol{v_L} = P_1 - P_0$ and $\boldsymbol{w} = P - P_0$. From section 2 we know that $\boldsymbol{v_L} \times \boldsymbol{w} = |\boldsymbol{v_L}||\boldsymbol{w}|\sin\theta|$ where θ is the angle between two vectors. This quantity also equals to the area of the parallelogram lying between vectors. Dividing the area with the length of base will give us the distance between the point and the base line or formally.

$$d(P, \boldsymbol{L}) = \frac{|\boldsymbol{v_L} \times \boldsymbol{w}|}{|\boldsymbol{v_L}|} = |\boldsymbol{u_L} \times \boldsymbol{w}|$$

where $\boldsymbol{u_L} = \boldsymbol{v_L}/|\boldsymbol{v_L}|$. If one is computing the distances of many points to a fixed line, then it is most efficient to first calculate $\boldsymbol{u_L}$.

Exercise 4.1. *Think about why this works only for 2D spaces.*

For higher dimensions, we can calculate the distance using the projection. Suppose that we have the

same setting as before. We can express our line as: $P(t) := P_0 + t(P_1 - P_0) = P_0 + t\boldsymbol{v_L}$ Now, let's find the projection $P(b)$ of vector $\boldsymbol{w} = \boldsymbol{P_0P}$ to the vector $\boldsymbol{v_L} = \boldsymbol{P_0P_1}$.

$$b = \frac{d(P_0, P(b))}{d(P_0, P_1)} = \frac{|\boldsymbol{w}| \cos\theta}{\boldsymbol{v_L}} = \frac{\boldsymbol{w} \cdot \boldsymbol{v_L}}{|\boldsymbol{v_L}|^2} = \frac{\boldsymbol{w} \cdot \boldsymbol{v_L}}{\boldsymbol{v_L} \cdot \boldsymbol{v_L}}$$

and thus:

$$d(P, \boldsymbol{L}) = |P - P(b)| = |\boldsymbol{w} - b\boldsymbol{v_L}| = |\boldsymbol{w} - (\boldsymbol{w} \cdot \boldsymbol{u_L})\boldsymbol{u_L}|$$

where $\boldsymbol{u_L}$ denotes unit vector as defined above and θ denotes the angle between the vectors.

By using this notion we can find the distance between a point and a line segment. Firstly let's discuss the concepts of rays and line segments. A **ray R** is a half line originating at a point P0 and extending indefinitely in some direction. It can be expressed parametrically as $P(t)$ for all $t \geq 0$ with $P(0) = P_0$ as the starting point. A finite segment S consists of the points of a line that are between two endpoints P_0 and P_1. Again, it can be represented by a parametric equation with $P(0) = P_0$ and $P(1) = P_1$ as the endpoints and the points $P(t)$ for $0 \leq t \leq 1$ as the segment points.

The thing that is different about computing distances of a point P to a ray or a segment is that the base $P(b)$ of the perpendicular from P to the extended line $\boldsymbol{L}$ may be outside the range of the ray or segment. In this case, the actual shortest distance is from the point P to the start point of the ray or one of the endpoints of a finite segment. The following figure illustrates the possible extreme conditions for the rays and line segments.

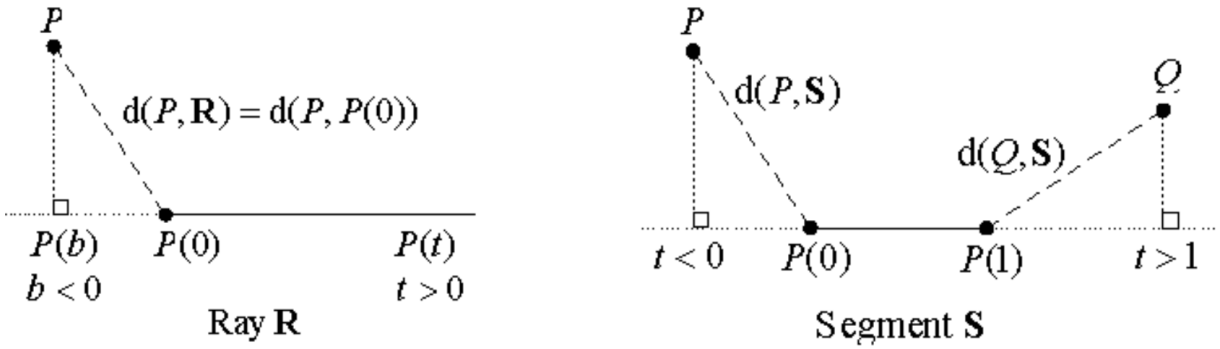


Figure 10: Point-Ray and Point-Line Segment distance calculations

In order to find the distance to a line segment, we will first calculate the value of b as above. Then, by checking that whether $P(b)$ lies on the line segment or not, we will find the nearest point of the line segment to the point.

The following implementation is taken from [4].

```
1  // Copyright 2001 softSurfer, 2012 Dan Sunday
2  // This code may be freely used, distributed and modified for any
     purpose
3  // providing that this copyright notice is included with it.
4  // SoftSurfer makes no warranty for this code, and cannot be held
5  // liable for any real or imagined damage resulting from its use.
6  // Users of this code must verify correctness for their application.
7
8
9  // Assume that classes are already given for the objects:
10 // Point and Vector with
11 // coordinates {float x, y, z;} (z=0 for 2D)
12 // appropriate operators for:
13 // Point = Point  Vector
14 // Vector = Point - Point
15 // Vector = Scalar * Vector
16 // Line with defining endpoints {Point P0, P1;}
17 // Segment with defining endpoints {Point P0, P1;}
18 //===================================================================
19
20 // dot product (3D) which allows vector operations in arguments
21 #define dot(u,v) ((u).x * (v).x + (u).y * (v).y + (u).z * (v).z)
22 #define norm(v) sqrt(dot(v,v)) // norm = length of vector
23 #define d(u,v) norm(u-v) // distance = norm of difference
24
25
26 // dist_Point_to_Line(): get the distance of a point to a line
27 // Input: a Point P and a Line L (in any dimension)
28 // Return: the shortest distance from P to L
29 float dist_Point_to_Line( Point P, Line L)
30 {
31     Vector v = L.P1 - L.P0;
32     Vector w = P - L.P0;
33
34     double c1 = dot(w,v);
35     double c2 = dot(v,v);
36     double b = c1 / c2;
37
38     Point Pb = L.P0 + b * v;
39     return d(P, Pb);
40 }
41
42
43 // dist_Point_to_Segment(): get the distance of a point to a segment
44 // Input: a Point P and a Segment S (in any dimension)
```

```
45 // Return: the shortest distance from P to S
46 float dist_Point_to_Segment( Point P, Segment S)
47 {
48     Vector v = S.P1 - S.P0;
49     Vector w = P - S.P0;
50
51     double c1 = dot(w,v);
52     if ( c1 <= 0 )
53         return d(P, S.P0);
54
55     double c2 = dot(v,v);
56     if ( c2 <= c1 )
57         return d(P, S.P1);
58
59     double b = c1 / c2;
60     Point Pb = S.P0 + b * v;
61     return d(P, Pb);
62 }
```

4.2 Planes

In order to keep these notes as compact as possible we won't dive into the details of the distance calculations between points and planes. However, a curious reader may consult to the excellent reference of [4] for distance calculations and their implementations.

5 Intersection

We discussed the parametric representation of lines in Section 4. In this section, we will deal with intersection points of lines and line segments of 2D Euclidean space. However, for more details about intersections in higher dimensions and intersections of higher dimensional surfaces (planes), one can consult the [4].

For the 2D case, we will examine two possible configurations separately. We will work on parallel lines and then work on the non-parallel case.

5.1 Parallel Lines

Two lines are parallel if and only if their direction vectors are collinear, namely when the two vectors $\boldsymbol{u} = \boldsymbol{P_1} - \boldsymbol{P_0}$ and $\boldsymbol{v} = \boldsymbol{Q_1} - \boldsymbol{Q_0}$ are linearly related as $\boldsymbol{u} = a\boldsymbol{v}$ for some real number a.

When we discussed the properties of dot product, we already saw that two vectors are perpendicular when their dot product is 0. Now, we will use this notion to check whether two vectors are co-linear or not. Suppose that $\boldsymbol{u} = (u_1, u_2)$, then a perpendicular vector $\boldsymbol{u}^\perp$ can be defined as $\boldsymbol{u}^\perp = (-u_2, u_1)$. One can easily check that $\boldsymbol{u} \cdot \boldsymbol{u}^\perp = 0$. Now, by considering the value of $\boldsymbol{v} \cdot \boldsymbol{u}^\perp$, we can check whether two vectors are co-linear or not.

In order to check whether a point P_0 lies on a line $Q(t)$, one can use simple linear calculations. If P_0 lies on the $Q(t)$ and $Q(t)$ is represented by a point Q_0 and direction vector $\mathbf{v}$ as $Q(t) = Q_0 + t\boldsymbol{v}$, we can write

$$P_0 = Q_0 + t_0\boldsymbol{v} \text{ , for some } t_0$$

Then, we can use this notion to check our desire, let $\boldsymbol{w} = \boldsymbol{Q_0P_0}$ or $\boldsymbol{w} = Q_0 - P_0$. Then if P_0 lies on $Q(t)$ then $\boldsymbol{w}$ and $\boldsymbol{v}$ must be co-linear, and we can check this as we have discussed above. We can calculate the value of t_0 as $t_0 = \frac{w_0}{v_1}$.

If both lines are finite segments, then they may (or may not) overlap. In this case, solve for t_0 and t_1 such that $P_0 = Q(t_0)$ and $P_1 = Q(t_1)$. If the segment intervals $[t_0, t_1]$ and $[0, 1]$ are disjoint, there is no intersection. Otherwise, intersect the intervals (using max and min operations) to get $[r_0, r_1] = [t_0, t_1] \cap [0, 1]$. Then the intersection segment is $\boldsymbol{Q(r_0)Q(r_1)} = \boldsymbol{P_0P_1} \cap \boldsymbol{Q_0Q_1}$. This works in any dimension.

5.2 Non-parallel Lines

When the two lines or segments are not parallel, they might intersect in a unique point. In 2D Euclidean space, infinite lines always intersect. In higher dimensions, they usually miss each other and do not intersect. But if they intersect, then their linear projections onto a 2D plane will also intersect. So, one can simply restrict situation to two coordinates, for which $\boldsymbol{u}$ and $\boldsymbol{v}$ (direction vectors) are not parallel, compute the 2D intersection point I at $P(s_I)$ and $Q(t_I)$ for those two coordinates, and then test if $P(s_I) = Q(t_I)$ for all coordinates. To compute the 2D intersection point, consider the two lines and the associated vectors in the diagram:

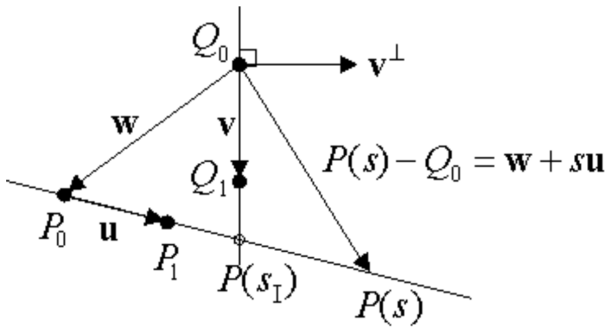


Figure 11: Illustration of finding intersection of two lines

To determine s_I, we have the vector equality $P(s) - Q_0 = \boldsymbol{w} + s\boldsymbol{u}$ where $\boldsymbol{w} = P_0 - Q_0$. At the intersection, the vector $P(s_I) - Q_0$ is perpendicular to $\boldsymbol{v}^\perp$, and this is equivalent to the perp product condition that $\boldsymbol{v}^\perp \cdot (\boldsymbol{w} + s\boldsymbol{u}) = 0$. Solving this equation, we get:

$$s_I = \frac{-\boldsymbol{v}^\perp \cdot \boldsymbol{w}}{\boldsymbol{v}^\perp \cdot \boldsymbol{u}} = \frac{v_2 w_1 - w_1 v_2}{v_1 u_2 - v_2 u_1}$$

Note that the denominator $\boldsymbol{v}^\perp \cdot \boldsymbol{u} = 0$ only when the lines are parallel as previously discussed. Similarly, solving for $Q(t_I)$, we get:

$$t_I = \frac{\boldsymbol{u}^\perp \cdot \boldsymbol{w}}{\boldsymbol{u}^\perp \cdot \boldsymbol{v}} = \frac{u_1 w_2 - u_2 w_1}{u_1 v_2 - u_2 v_1}$$

The denominators are the same up to sign, since $\boldsymbol{u}^\perp \cdot \boldsymbol{v} = -\boldsymbol{v}^\perp \cdot \boldsymbol{u} = 0$, and should only be computed once if we want to know both s_I and t_I. However, knowing either is enough to get the intersection point $I = P(s_I) = Q(t_I)$.

Further, if one of the two lines is a finite segment (or a ray), say P_0P_1, then the intersect point is in the segment only when $0 \le s_I \le 1$ (or $s_I \ge 0$ for a ray). If both lines are segments, then both solution parameters, s_I and t_I, must be in the $[0, 1]$ interval for the segments to intersect. Although this sounds simple enough, the code for the intersection of two segments is a bit delicate since many special cases need to be checked (see the following implementation).

The following implementation is taken from [4].

```
1  // Copyright 2001 softSurfer, 2012 Dan Sunday
2  // This code may be freely used and modified for any purpose
3  // providing that this copyright notice is included with it.
4  // SoftSurfer makes no warranty for this code, and cannot be held
5  // liable for any real or imagined damage resulting from its use.
6  // Users of this code must verify correctness for their application.
7
8
9  // Assume that classes are already given for the objects:
10 // Point and Vector with
11 // coordinates {float x, y, z;}
12 // operators for:
13 // == to test equality
14 // != to test inequality
15 // Point = Point  Vector
16 // Vector = Point - Point
17 // Vector = Scalar * Vector (scalar product)
18 // Vector = Vector * Vector (3D cross product)
19 // Line and Ray and Segment with defining points {Point P0, P1;}
20 // (a Line is infinite, Rays and Segments start at P0)
```

```
21 // (a Ray extends beyond P1, but a Segment ends at P1)
22 // Plane with a point and a normal {Point V0; Vector n;}
23 //===================================================================
24
25
26 #define SMALL_NUM 0.00000001 // anything that avoids division overflow
27 // dot product (3D) which allows vector operations in arguments
28 #define dot(u,v) ((u).x * (v).x + (u).y * (v).y + (u).z * (v).z)
29 #define perp(u,v) ((u).x * (v).y - (u).y * (v).x) // perp product (2D)
30
31 // intersect2D_2Segments(): find the 2D intersection of 2 finite
      segments
32 // Input: two finite segments S1 and S2
33 // Output: *I0 = intersect point (when it exists)
34 // *I1 = endpoint of intersect segment [I0,I1] (when it exists)
35 // Return: 0=disjoint (no intersect)
36 // 1=intersect in unique point I0
37 // 2=overlap in segment from I0 to I1
38 int intersect2D_2Segments( Segment S1, Segment S2, Point* I0, Point* I1
       )
39 {
40     Vector u = S1.P1 - S1.P0;
41     Vector v = S2.P1 - S2.P0;
42     Vector w = S1.P0 - S2.P0;
43     float D = perp(u,v);
44
45     // test if they are parallel (includes either being a point)
46     if (fabs(D) < SMALL_NUM) { // S1 and S2 are parallel
47         if (perp(u,w) != 0 || perp(v,w) != 0) {
48             return 0; // they are NOT collinear
49         }
50         // they are co-linear or degenerate
51         // check if they are degenerate points
52         float du = dot(u,u);
53         float dv = dot(v,v);
54         if (du==0 && dv==0) { // both segments are points
55             if (S1.P0 != S2.P0) // they are distinct points
56                 return 0;
57             *I0 = S1.P0; // they are the same point
58             return 1;
59         }
60         if (du==0) { // S1 is a single point
61             if (inSegment(S1.P0, S2) == 0) // but is not in S2
62                 return 0;
63             *I0 = S1.P0;
64             return 1;
```

```
65        }
66        if (dv==0) { // S2 a single point
67           if (inSegment(S2.P0, S1) == 0) // but is not in S1
68               return 0;
69           *I0 = S2.P0;
70           return 1;
71        }
72        // they are co-linear segments - get overlap (or not)
73        float t0, t1; // endpoints of S1 in eqn for S2
74        Vector w2 = S1.P1 - S2.P0;
75        if (v.x != 0) {
76               t0 = w.x / v.x;
77               t1 = w2.x / v.x;
78        }
79        else {
80               t0 = w.y / v.y;
81               t1 = w2.y / v.y;
82        }
83        if (t0 > t1) { // must have t0 smaller than t1
84               float t=t0; t0=t1; t1=t; // swap if not
85        }
86        if (t0 > 1 || t1 < 0) {
87           return 0; // NO overlap
88        }
89        t0 = t0<0? 0 : t0; // clip to min 0
90        t1 = t1>1? 1 : t1; // clip to max 1
91        if (t0 == t1) { // intersect is a point
92           *I0 = S2.P0 + t0 * v;
93           return 1;
94        }
95
96        // they overlap in a valid subsegment
97        *I0 = S2.P0 + t0 * v;
98        *I1 = S2.P0 + t1 * v;
99        return 2;
100    }
101
102    // the segments are skew and may intersect in a point
103    // get the intersect parameter for S1
104    float sI = perp(v,w) / D;
105    if (sI < 0 || sI > 1) // no intersect with S1
106       return 0;
107
108    // get the intersect parameter for S2
109    float tI = perp(u,w) / D;
110    if (tI < 0 || tI > 1) // no intersect with S2
```

```
111         return 0;
112
113     *I0 = S1.P0 + sI * u; // compute S1 intersect point
114     return 1;
115 }
116
117 // inSegment(): determine if a point is inside a segment
118 // Input: a point P, and a collinear segment S
119 // Return: 1 = P is inside S
120 // 0 = P is not inside S
121 int inSegment( Point P, Segment S)
122 {
123     if (S.P0.x != S.P1.x) { // S is not vertical
124         if (S.P0.x <= P.x && P.x <= S.P1.x)
125             return 1;
126         if (S.P0.x >= P.x && P.x >= S.P1.x)
127             return 1;
128     }
129     else { // S is vertical, so test y coordinate
130         if (S.P0.y <= P.y && P.y <= S.P1.y)
131             return 1;
132         if (S.P0.y >= P.y && P.y >= S.P1.y)
133             return 1;
134     }
135     return 0;
136 }
```

5.3 Further Readings

If you are curious about intersections of types line-plane, plane-plane, more than 2 planes, triangle-plane and triangle-triangle, then you should definitely take a look at [4].

6 Inclusion Problem

Inclusion problem is defined as determining the inclusion of a point P in a 2D simple polygon. For this problem there are two common methods. These methods actually work for non-simple planar polygons as well.

6.1 The Crossing Number Method

This method counts the number of times a ray starting from a point P crosses a polygon boundary edge separating its inside and outside. If this number is even, then the point is outside; when the crossing number is odd, the point is inside. This is easy to understand intuitively. Each time the ray crosses a polygon edge, its in-out parity changes (since a boundary always separates inside from outside, why?). Eventually, any ray must end up beyond and outside the bounded polygon. So, if the point is inside, the sequence of crossings must be: in > out > ... > in > out, and there is an odd number of them. Similarly, if the point is outside, there are an even number of crossings in the sequence: out > in > ... > in > out.

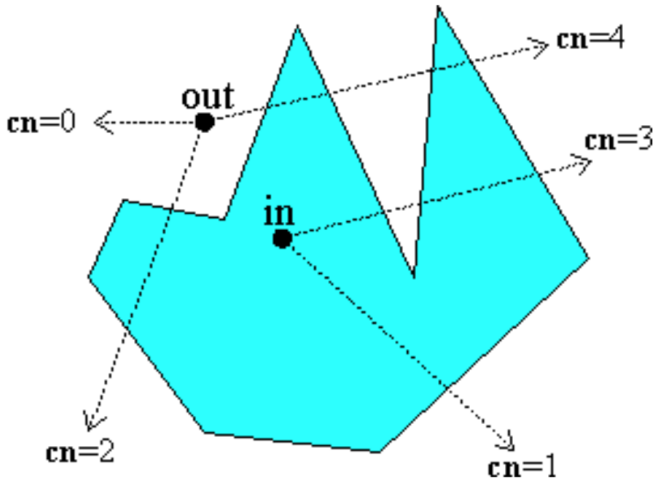


Figure 12: Illustration of crossing number method

In implementing an algorithm for the crossing number method, one must ensure that only crossings that change the in-out parity are counted. In particular, special cases where the ray passes through a vertex must be handled properly. These include the following types of ray crossings:

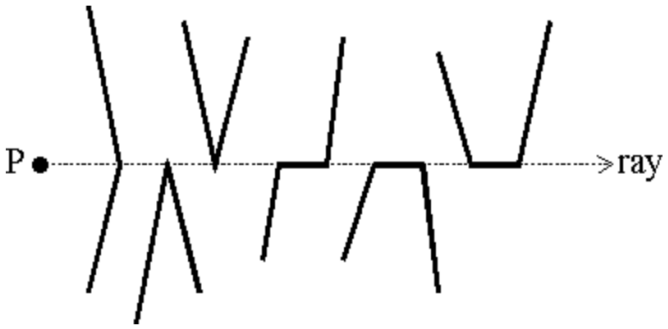


Figure 13: An edge case for crossing number method which must be avoided

One can use the following edge crossing rules to avoid the problems that occurred at the figure.

Edge Crossing Rules:

1. an upward edge includes its starting endpoint, and excludes its final endpoint
2. a downward edge excludes its starting endpoint, and includes its final endpoint
3. horizontal edges are excluded
4. the edge-ray intersection point must be strictly right of the point P

On the other hand, one can trust the power of randomness and check their points' inclusion with different random vectors. The most occurring result will give us the correct solution with very high probability.

The following implementation is taken from [4].

```
1  typedef struct {int x, y;} Point;
2
3  cn_PnPoly( Point P, Point V[], int n )
4  {
5     int cn = 0; // the crossing number counter
6
7     // loop through all edges of the polygon
8     for (each edge E[i]:V[i]V[i+1] of the polygon) {
9        if (E[i] crosses upward as long as Rule 1
10        || E[i] crosses downward as long as Rule 2) {
11          if (P.x < x_intersect of E[i] with y=P.y) // Rule #4
12              ++cn; // a valid crossing to the right of P.x
13       }
14    }
15    return (cn&1); // 0 if even (out), and 1 if odd (in)
16
17 }
```

6.2 Winding Number Method

The Winding Number Method counts the number of times the polygon winds around the point P. The point is outside only when this "winding number"= 0; otherwise, the point is inside.

In order to count the number of times the polygon winds around the point P, we will only consider the edges that lie at the left part of the point. Subtracting the number of downward edges at the left from the number of upward edges at the left will give us the winding number of that point.

We won't dive into the details of this method or give the implementation. However, it is strongly suggested to see [4].

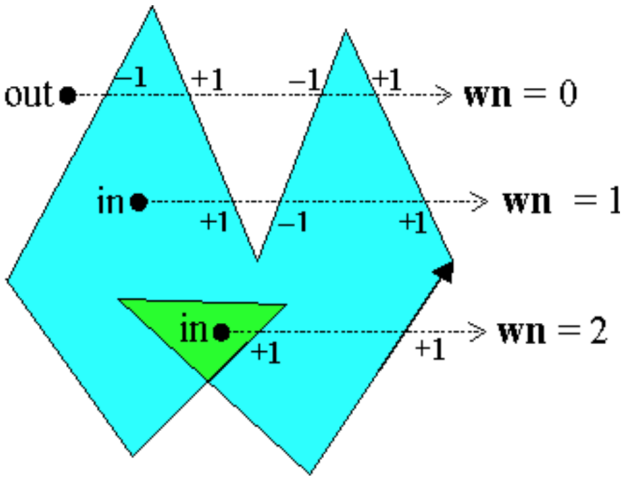


Figure 14: Illustration of winding number method

7 Convex Hull

The convex hull of a geometric object (point set, polygon or polygon set) is the smallest convex set containing that object. There are many equivalent definitions for a convex set S. The most basic of these is:

Definition 7.1. *A set S is **convex** if whenever two points P and Q are inside S, then the whole line segment **PQ** is also in S.*

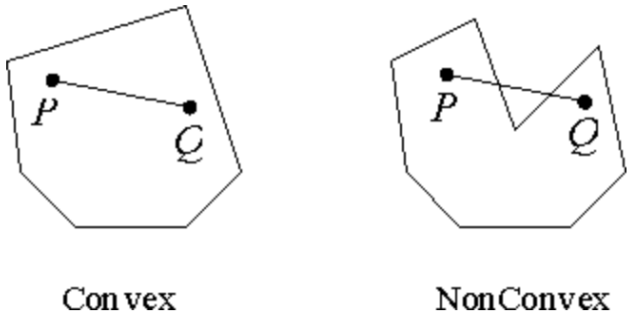


Figure 15: Examples of convex and non-convex sets

Definition 7.2. *A set S is **convex** if it is exactly equal to the intersection of all the half planes containing it.*

In different dimensions convex sets have different names, for example in 2D it is convex polygon and in 3D it is polyhedron. However, in this lecture we will only work on 2D.

7.1 Graham-Scan Algorithm

Our first algorithm is known as Graham-Scan Algorithm due to [5]. Before we discuss the algorithm we will first define a primitive which is required for this algorithm. As we discussed earlier the area calculation of the triangle has a sign which denotes whether three points in counter-clockwise(CCW) order or clockwise(CW) order. We will use this calculation mechanism to check whether a point lies on the left half-plane of a vector or the right one.

Suppose we have a set of points S and we want to find the convex hull of those points. We will firstly find the rightmost point from the lowest points of the set. Let's say that point P_0. This can be done in $O(n)$ time complexity. Then by considering that point as origin (suppose that we translated points to make this point as origin), we will sort each point P according to the angle between the $\boldsymbol{P_0P}$ and x-axis in increasing order. Then we will be in a situation similar to the following figure.

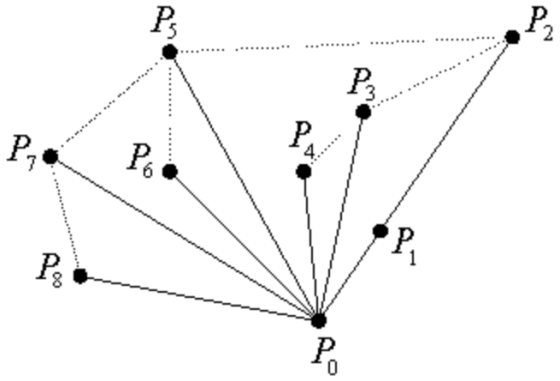
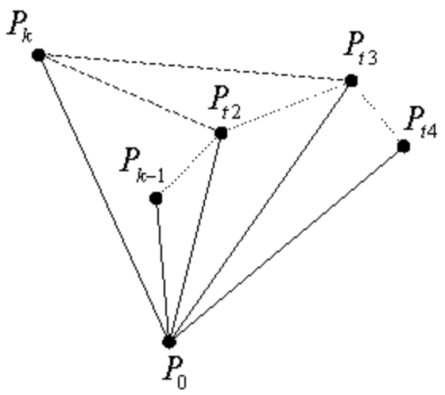


Figure 16: Points sorted according to their radial angles

When sorting those points one can use trigonometric functions that belong to the programming environment but this might be inaccurate and inefficient. Instead of trigonometric functions, one can also use the isLeft function to compare locations of the points relative to each other. One important trick to speed up the process is that if two points have the same radial angle then nearer one should be ignored since it can not belong to the convex hull(why?).

We next iterate through all points in a determined order. We will use a stack data structure to keep track of which points belong to convex hull. We will start with two points in the stack P_0 and P_1. Then at k^{th} step of iterations we will look at the relative position of the next point with respect to the vector determined by the top two points of the stack. If the new point lies at the left part of that vector then that point will be added to the top of the stack. If it lies at the right part of the vector, then the top of the vector must be removed from the stack until finding such a vector (Think about why we can eventually find such a vector.). Note that each point will be added to the stack at first consideration, and then it may be removed or not. The following figure illustrates one interesting situation.



Old Stack $= S_{k-1} = \{P_0, \ldots, P_{t3}, P_{t2}, P_{k-1}\}$
P_k right of line $P_{t2}P_{k-1} \Rightarrow$ pop P_{k-1} off stack
P_k right of line $P_{t3}P_{t2} \Rightarrow$ pop P_{t2} off stack
P_k left of line $P_{t4}P_{t3} \Rightarrow$ push P_k onto stack
New Stack $= S_k = \{P_0, \ldots, P_{t3}, P_k\}$

Figure 17: Example of iterations at stack

Exercise 7.1. *Convince yourself why the algorithm works. Think about what is the importance of having points in increasing radial order.*

The following implementation is taken from [4].

```
1    Input: a set of points S = {P = (P.x,P.y)}
2
3    Select the rightmost lowest point P0 in S
4    Sort S radially (ccw) about P0 as a center {
5       Use isLeft() comparisons // see below for the implementation
6       For ties, discard the closer points
7    }
8    Let P[N] be the sorted array of points with P[0]=P0
9
10   Push P[0] and P[1] onto a stack OMEGA
11
12   while i < N
13   {
14      Let PT1 = the top point on OMEGA
15      If (PT1 == P[0]) {
16         Push P[i] onto OMEGA
17         i++ // increment i
18      }
19      Let PT2 = the second top point on OMEGA
20      If (P[i] is strictly left of the line PT2 to PT1) {
21         Push P[i] onto OMEGA
22         i++ // increment i
23      }
24      else
25         Pop the top point PT1 off the stack
26   }
27
28   Output: OMEGA = the convex hull of S.
```

7.2 Andrew's Monotone Chain Algorithm

Andrew's algorithm [6] is very similar to the previous one but in comparison to Graham's algorithm, this algorithm sorts the points according to their coordinates.

Suppose as in the previous case, you have a set of points S and you want to find the convex hull containing these points. Firstly, sort all the points by increasing x and then y (when there are points with same x coordinates) coordinate values. Let $P_{min,min}$ be the point with minimum x coordinate and from those points minimum y coordinate. And similarly define $P_{min,max}$, $P_{max,min}$, $P_{max,max}$. Note that sometimes $P_{min,max} = P_{min,min}$ and $P_{max,min} = P_{max,max}$. Define two lines L_{min} which connects $P_{min,min}$ and $P_{max,min}$ and L_{max} connecting $P_{min,max}$ and $P_{max,max}$. An example situation can be shown in the following figure.

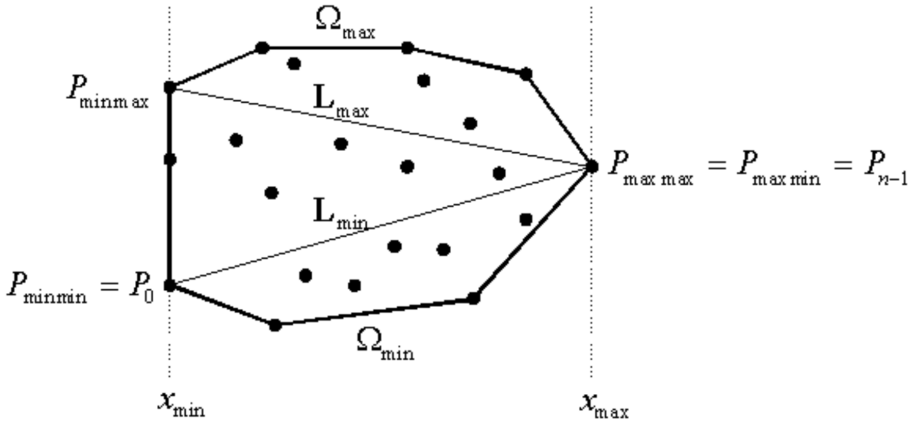


Figure 18: Illustration of Andrew's Monotone Chain Algorithm

Now, the aim of the algorithm is finding upper hull Ω_{max} which is above the L_{max} and finding lower hull Ω_{min} which is below the L_{min}. Then the convex hull Ω will be constructed by connecting these two hulls.

While finding the upper and lower hull we will iterate the stack idea we have used in the previous algorithm to keep of track which points belong to hull. For the Ω_{max}, we will iterate the process in increasing x order and consider each point if it lies above the L_{max}. When considering a point we will check whether a point lies at right side of the line determined by the top two elements of the stack (similar to the previous algorithm). Using this comparison we will add the new point to the stack or remove the top points from the stack until we find such a case. An analogous version of this can be defined for lower hull as well.

The following implementation is taken from [4].

```
// Copyright 2001 softSurfer, 2012 Dan Sunday
// This code may be freely used and modified for any purpose
// providing that this copyright notice is included with it.
// SoftSurfer makes no warranty for this code, and cannot be held
// liable for any real or imagined damage resulting from its use.
// Users of this code must verify correctness for their application.


// Assume that a class is already given for the object:
// Point with coordinates {float x, y;}
//===================================================================


// isLeft(): tests if a point is Left|On|Right of an infinite line.
// Input: three points P0, P1, and P2
// Return: >0 for P2 left of the line through P0 and P1
// =0 for P2 on the line
// <0 for P2 right of the line
// See: Algorithm 1 on Area of Triangles
inline float isLeft( Point P0, Point P1, Point P2 )
{
    return (P1.x - P0.x)*(P2.y - P0.y) - (P2.x - P0.x)*(P1.y - P0.y);
}


// chainHull_2D(): Andrew's monotone chain 2D convex hull algorithm
// Input: P[] = an array of 2D points
// presorted by increasing x and y-coordinates
// n = the number of points in P[]
// Output: H[] = an array of the convex hull vertices (max is n)
// Return: the number of points in H[]
int chainHull_2D( Point* P, int n, Point* H )
{
    // the output array H[] will be used as the stack
    int bot=0, top=(-1); // indices for bottom and top of the stack
    int i; // array scan index

    // Get the indices of points with min x-coord and min|max y-coord
    int minmin = 0, minmax;
    float xmin = P[0].x;
    for (i=1; i<n; i++)
        if (P[i].x != xmin) break;
    minmax = i-1;
    if (minmax == n-1) { // degenerate case: all x-coords == xmin
        H[++top] = P[minmin];
```

```
46          if (P[minmax].y != P[minmin].y) // a nontrivial segment
47              H[++top] = P[minmax];
48          H[++top] = P[minmin]; // add polygon endpoint
49          return top+1;
50      }
51
52      // Get the indices of points with max x-coord and min|max y-coord
53      int maxmin, maxmax = n-1;
54      float xmax = P[n-1].x;
55      for (i=n-2; i>=0; i--)
56          if (P[i].x != xmax) break;
57      maxmin = i+1;
58
59      // Compute the lower hull on the stack H
60      H[++top] = P[minmin]; // push minmin point onto stack
61      i = minmax;
62      while (++i <= maxmin)
63      {
64          // the lower line joins P[minmin] with P[maxmin]
65          if (isLeft( P[minmin], P[maxmin], P[i]) >= 0 && i < maxmin)
66              continue; // ignore P[i] above or on the lower line
67
68          while (top > 0) // there are at least 2 points on the stack
69          {
70              // test if P[i] is left of the line at the stack top
71              if (isLeft( H[top-1], H[top], P[i]) > 0)
72                  break; // P[i] is a new hull vertex
73              else
74                  top--; // pop top point off stack
75          }
76          H[++top] = P[i]; // push P[i] onto stack
77      }
78
79      // Next, compute the upper hull on the stack H above the bottom hull
80      if (maxmax != maxmin) // if distinct xmax points
81          H[++top] = P[maxmax]; // push maxmax point onto stack
82      bot = top; // the bottom point of the upper hull stack
83      i = maxmin;
84      while (--i >= minmax)
85      {
86          // the upper line joins P[maxmax] with P[minmax]
87          if (isLeft( P[maxmax], P[minmax], P[i]) >= 0 && i > minmax)
88              continue; // ignore P[i] below or on the upper line
89
90          while (top > bot) // at least 2 points on the upper stack
91          {
```

```
92          // test if P[i] is left of the line at the stack top
93          if (isLeft( H[top-1], H[top], P[i]) > 0)
94              break; // P[i] is a new hull vertex
95          else
96              top--; // pop top point off stack
97      }
98      H[++top] = P[i]; // push P[i] onto stack
99  }
100 if (minmax != minmin)
101     H[++top] = P[minmin]; // push joining endpoint onto stack
102
103 return top+1;
104 }
```

Exercise 7.2. *Convince yourself why algorithm works. Think about what is the importance of having in increasing x order.*

8 Rotating Calipers

Let $\Omega = (p_1, p_2, \cdots p_n)$ be a convex polygon whose points are in clockwise order and no three consecutive points are co-linear(lie at the same line). Our aim is finding the diameter of the polygon Ω. The diameter of a polygon can be written as

$$diam(\Omega) = \max_{p,q \in \Omega} \{d(p, q)\}$$

A line L is a line of support of Ω if the interior of Ω. lies completely to one side of L. Following figure illustrates two parallel lines of support.

Theorem 8.1. *The diameter of a convex polygon Ω is the greatest distance between parallel lines of support of Ω.*

For the proof, please see the [REF2].

A pair of vertices p_i and p_j are an antipodal pair if they admit parallel lines of support. The algorithm of Shamos [7] generates all O(n) antipodal pairs of vertices and selects the pair with the largest distance as the diameter-pair. The procedure resembles rotating a pair of dynamically adjustable calipers once around the polygon. Consider the following figure. To initialize the algorithm in a direction such as the x-axis is chosen and the two antipodal vertices p_i and p_j can be found in O(n) time (points with max x coordinate and min x coordinate). To generate the next antipodal pair we consider the angles that the lines of support at p_i and p_j make with edges $\boldsymbol{p_i p_{i+1}}$ and $\boldsymbol{p_j p_{j+1}}$, respectively. Let angle $\theta_j < \theta_i$. Then we rotate the lines of support by an angle θ_j, and p_{j+1}, p_i becomes the next antipodal pair. This process is continued until we come full circle to the starting position. In the event that $\theta_j = \theta_i$ three new antipodal pairs are generated.

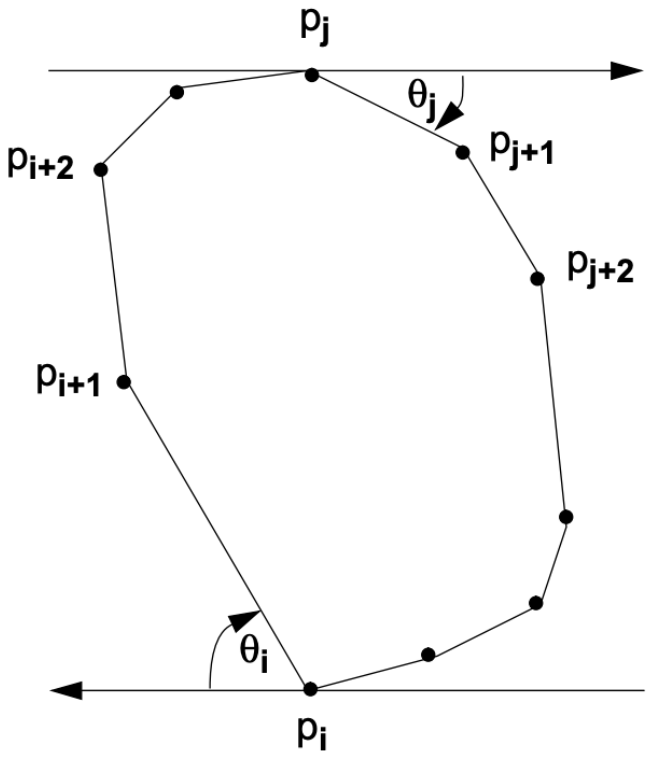


Figure 19: Illustration of an iteration of rotating calipers algorithm

In implementation, we will start with a random point and find its antipodal pair. Then at each step we will compare the angles between the line of supports and next edges of the polygon for both antipodal pair. Then we will rotate our calipers (pair of support lines) with the minimum angle. In this case one of our line of support will contain an edge of the polygon. This process will be terminated when a full cycle is completed. One can see [14] for C++ implementations of the algorithm.

8.1 Applications

In this section, we will list some problems which can be solved by using rotating calipers algorithm. A curious reader may consult to [8].

- **The smallest area enclosing rectangle**: can be solved by using two pairs of calipers perpendicular to each other.
- **The maximum distance between two convex polygons**: can be solved by finding the line of supports for each polygon and iterate rotating idea similar to diameter problem.
- **Merging Convex Hulls**: with using calipers idea this can be done in O(n) time.

9 Closest Pair

In this section we will study a well defined problem known as closest pair. Suppose that you have a set of points $\{p_1, \cdots, p_n\}$ find the pair of points $\{p_i, p_j\}$ that are closest together. Or formally,

$$d(p_i, p_j) \leq d(p_t, p_k) \quad \forall t, k \in \{1, \cdots k\} \text{ and } t \neq k$$

A naive approach can be designing algorithm which measures distance between every pairs. With this approach we can find the closest pair in $O(n^2)$ time complexity. However, we can improve this running time to $O(n \log n)$.

To find such a pair we will examine two different methods. These two methods are very important concepts which can be applied to numerous problems.

9.1 Divide and Conquer

Firstly, we will study the Divide and Conquer algorithm which is designed for the closest pair problem, using [9] and [10]. In fact, we are familiar with this concept from the merge-sort algorithm. The idea is similar; we will divide the set of points to (almost) equal sized two sets and calculate the desired quantity for those sets. Then, we will merge two sets by finding the closest pairs of the combined set. The algorithm can be summarized as following.

We will use induction to build up our algorithm. For the base case, if we have a set of size 2 then the result will be the only pairs. Suppose that we have an algorithm which finds desired pair for a set of size less than n. Our divide phase makes use of this sub-solver.

Firstly, we will sort all the points according to their x coordinates. Then, we will find a line L where (almost) half of the points lie at the left side of the line. Then we will use the sub-solver to find the distance d_{left} of closest pair of the left part and the distance d_{right} of closest pair of the right part. Each sub-solver will call also another sub-solver to find its closest pair recursively.

Next phase is conquer or sometimes called merge phase. At the beginning of this phase we will set $d = \min\{d_{left}, d_{right}\}$. d would be the answer, except maybe L split a close pair. Then, we need to deal with the near points to L. It is obvious that we do not need to consider points which are d unit far from L since they can not produce any closer pair. Then we will look the points which lie inside of the $2d$ width strip around the L.

Let S_y be an array of the points in that region, sorted by decreasing y-coordinate value. The bad case is the strip might contain all points; then we can not try every pair which can be obtained from S_y. However, the following theorem helps us to avoid this situation.

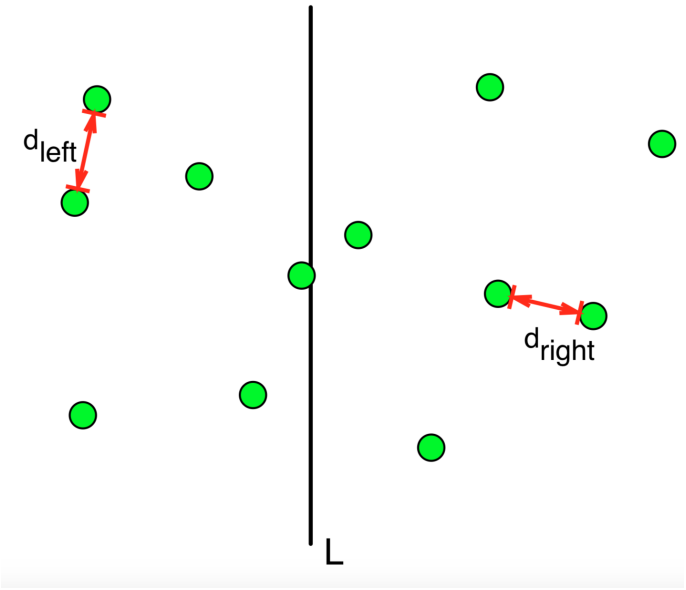


Figure 20: Divide phase of the algorithm

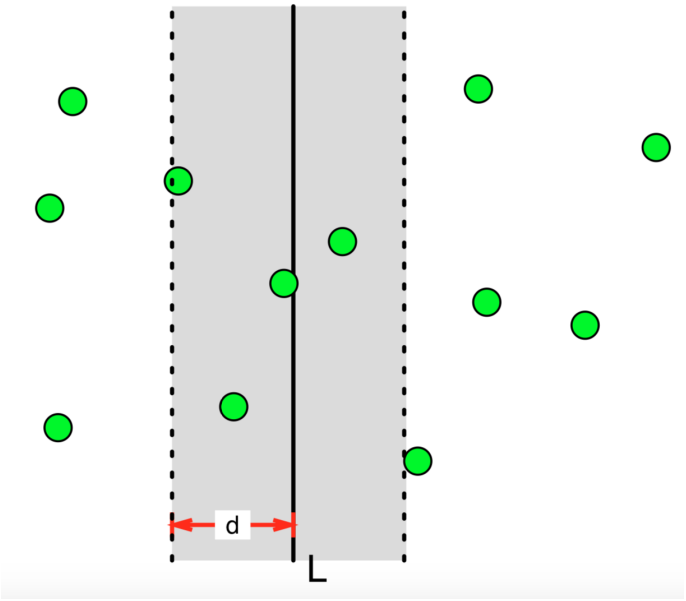


Figure 21: Merge phase and strip around line L

Theorem 9.1. *Suppose* $S_y = \{p_1, \cdots, p_m\}$. *If* $d(p_i, p_j) < d$ *then* $j - i \leq 15$.

In other words, if two points in S_y are close enough in the plane, they are close in the array S_y. Then for each point we can look for a constant amount of other points to find our desired pair.

Proof. Suppose that we divided the region up into squares with sides of length $d/2$. Then, each box contains at most 1 point (why?).

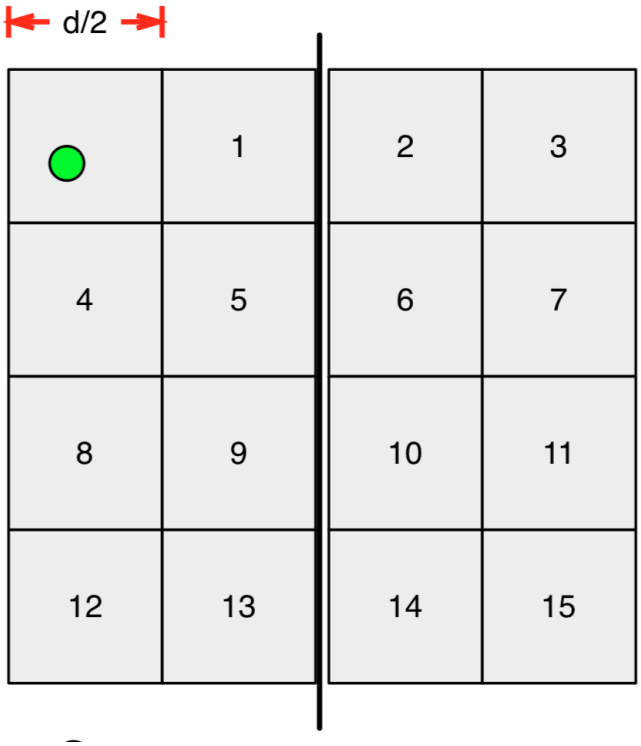


Figure 22: Line strip divided into squares

If 2 points are separated by > 15 indices, then at least 3 full rows separate them. However, the height of 3 rows is $3d/2$ which is bigger than d. So the two points are farther than d apart. □

Exercise 9.1. *Try to improve the constant* 15 *in the theorem.*

The following implementation is taken from [9].

```
1  ClosestPair(Px, Py):
2    if |Px| == 2: return dist(Px[1],Px[2]) // base
3    d1 = ClosestPair(FirstHalf(Px,Py)) // divide
4    d2 = ClosestPair(SecondHalf(Px,Py))
5    d = min(d1,d2)
6    Sy = points in Py within d distance of L // merge
7    For i = 1,...,|Sy|:
8      For j = 1,...,15:
9         d = min( dist(Sy[i], Sy[j]), d )
10 Return d
```

One can merge two regions by an algorithm similar to merge-sort to obtain $O(n)$ merging time complexity. Since we can divide sets into sub-sets at most $O(\log n)$ depth recursion and at each depth we do operations with $O(n)$ time complexity. Complexity of the algorithm is the same with

merge sort.

9.2 Line Sweep

As we mentioned above, the algorithmic technique we shall use is the line sweep method. This means that we will be sweeping a vertical line across the set of points, keeping track of certain data, and performing certain actions every time a point is encountered during the plane sweep. As we sweep the line, we will maintain the following data:

- The closest pair among the points encountered
- The distance d between the points in the above pair
- All points within a strip of distance d to the left of the sweep line (see the next figure). These points will be stored in an ordered set D (sorted by their y-coordinates)

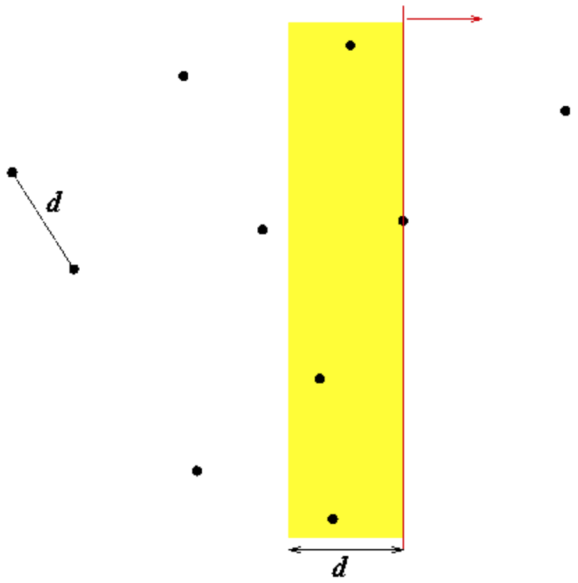


Figure 23: Illustration of sweep-line method where active points are colored yellow

Now, every time the sweep line encounters a point p, we will perform the following actions:

1. Remove the points further than d to the left of p from the ordered set D that is storing the points in the strip.
2. Determine the point on the left of p that is closest to it
3. If the distance between this point and p is less than d (the current minimum distance), then update the closest current pair and d.

Steps 1 and 3 need no further elaboration, but step two is a little unclear. It might seem like a method with $O(n)$ time complexity for each encountered point. However, as in the previous case we can consider only the constant amount of points lying at left side of the line.

Exercise 9.2. *Think about why this algorithm works. You can see [11] for the proof of correctness.*

The following implementation is taken from [12].

```
#define x first
#define y second

Following implementation is taken from [9].
typedef pair<long long, long long> pll;
   ...
set <pll> boundingBox;
boundingBox.insert(points[0]); //Points have been already sorted by x-
   coordinate
for (int i = 1; i < numPoints; i++)
{
   while ((left < i) && (points[i].x - points[left].x >
      shortestDistSoFar))
   {
      boundingBox.erase(points[left]);
      left++;
   }

   for (auto it = boundingBox.lower_bound(pll(points[i].y -
      shortestDistSoFar, points[i].x - shortestDistSoFar)); it !=
      boundingBox.end() && it->y <= points[i].y + shortestDistSoFar; it
      ++)
   {
      if (dist(*it, points[i]) < shortestDistSoFar)
      {
         shortestDistSoFar = dist(*it, points[i]);
         best1 = *it;
         best2 = points[i];
      }
   }
   boundingBox.insert(points[i]);
}
```

Besides the problem of closest pair, Line-Sweep method has many different applications and it is very useful technique to solve different kinds of geometric problems. In order to see different applications of this technique one can see the excellent blog post of Bruce Merry [12]

References

[1] Wong Yan Loi, MA1104 Multivariable Calculus Lecture Notes

[2] Stewart,J., Calculus, Brooks/Cole., 5th edition, 2003

[3] https://www.wikiwand.com/en/Metric_space

[4] http://geomalgorithms.com/

[5] Graham, R.L. (1972). "An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set", Information Processing Letters. 1 (4): 132–133.

[6] A. M. Andrew. (1979). "Another Efficient Algorithm for Convex Hulls in Two Dimensions", Info. Proc. Letters 9, 216-219.

[7] Shamos, Michael (1978). "Computational Geometry" (PDF). Yale University. pp. 76–81.

[8] Toussaint, Godfried T. (1983). "Solving geometric problems with the rotating calipers". Proc. MELECON '83, Athens. CiteSeerX 10.1.1.155.5671

[9] https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/closepoints.pdf

[10] Kleinberg, Tardos, (2005). "Algorithm Design". Section 5.4. Addison-Wesley Longman Publishing Co., Inc.

[11] https://www.cs.mcgill.ca/~cs251/ClosestPair/ClosestPairPS.html

[12] https://www.topcoder.com/community/competitive-programming/tutorials/line-sweep-algorithms/

[13] https://stackoverflow.com/questions/27674554/understanding-specific-implementation-of-sweep-line-closest-pair

[14] https://ideone.com/FxjD9D