NOI.PH Training: Geometry 1

Basic Plane Geometry Techniques

Kevin Charles Atienza

Contents

1 Introduction

This module is about geometry, and how it shows up in competitive programming.

Explicit computational geometry doesn't come up that often in IOI. IOI even explicitly forbids the more advanced aspects of geometry. However, that doesn't mean they never come up. Some basic stuff sometimes do. Also, knowing the basics sometimes help in discovering solutions to even non-geometry problems. There are lots of non-geometry problems where solving it is greatly aided by spatial and geometric reasoning! (For example, data structures can be thought of geometrically by thinking of numbers as coordinates.) Finally, simply being exposed to new ideas and algorithms in different contexts will surely be beneficial to you as a problem solver.

Here, we'll focus on the geometry on the **plane**, specifically the Cartesian plane.

For further reading, see [1].

1.1 Computational Geometry

In high school, plane geometry has a certain flavor. You have a bunch of points, lines, circles, shapes, etc., and they have certain properties and relationships, and often you're asked to deduce more properties. Coordinates are disfavored; what we want are elegant[1] conceptual arguments that only utilize the given properties. We build this whole tower of results with coordinate-free deductive arguments starting from Euclid's axioms. This is called **synthetic geometry**. It's pretty common in math olympiad geometry too.

However, usually in programming contests, we use explicit Cartesian coordinates, since that's the simplest way to give spatial input to your program. Instead of giving a bunch of statements like "A, B and C are collinear" or "XY is perpendicular to ZW" which may be complicated, we simply give coordinates to all the points involved, and we instantly have a geometric setup! Also, coordinates are more natural to *compute* with, which is pretty important when you're doing algorithms. So this is why we call it **computational geometry**.

This doesn't mean you should throw away all the intuition you have from synthetic geometry! At least some of those are still useful, and sometimes, the problems themselves may still require them anyway—for example, some Project Euler problems are basically combinations of synthetic geometry and number theory.

So we'll be doing geometry with a focus on algorithms. We will still have some sort of "primitive objects" and "primitive operations" between them.

Our "primitive objects" are still points, lines, line segments, polygons, circles, etc., but we'll include **vectors** to the mix. Vectors provide a handy way to perform many computational geometry algorithms in elegant, "holistic" ways, as opposed to other techniques where you may have to deal with annoying cases. (We'll see more of this later.)

Here are some "primitive operations":

- **Collinearity check.** Given three points, determine if they are collinear or not.
- **Perpendicularity check.** Given two vectors, determine if they are perpendicular or not.
- **Counterclockwise check.** Given points A, B and C, is C to the left[2] of B when viewed from the vantage point of A? This is shortened by many people to simply "ccw."

[1] unless you've been doing two-column proofs, in which case "elegant" is...debatable.

[2] equivalently, counterclockwise

- **Line-line intersection.** Given two lines, find their intersection.
- **Segment-segment intersection.** Given two line segments, do they intersect?
- **Point in triangle.** Given a point and a triangle, is the point inside the triangle or not? (More generally, we have "**point in polygon**".)
- **Polygon area.** Given a polygon, what is its area?
- **Linear transformations.** Given a vector, rotate it, or reflect it, scale it, shear it, etc.

There are many more. Some are more advanced than others.

1.2 Advice

Before we proceed, let me give you some pieces of advice, along with some words of warning:

- **Floating point is evil!** We've detailed this in an earlier module, but the first part of [1] also serves as a warning. After reading that first part, you should truly feel how terrible floating point really is.

  Be careful with floats. Don't use them if you can. If you have to use them, try to minimize the amount of times you have to!

- **Hard to brute force.** One notable thing I've noticed with geometry problems is that in some sense they're an opposite of data structure problems. Data structure problems are often easy to brute force, whereas geometry problems are hard to brute force! This makes them usually one of the harder problems in a set, because you can't do the usual "stress testing" things you can usually do with other problems (except maybe after much effort).

- **Full of edge cases.** One reason it's hard to brute force is that geometry "solutions" tend to have so many tricky cases and edge cases. For example, consider the following problem:

  > Given a point P and a triangle ABC, is the point P inside (or on the boundary of) triangle ABC?

  There are many solutions to this. All of them probably run in $\mathcal{O}(1)$, but still, some of them may be considered "better" than others, particularly at handling edge cases.

  For example, let's consider a clever and seemingly-innocent solution: Consider the three triangles formed by P and every pair of vertices of ABC—namely PAB, PBC and PCA. Notice that P is inside the triangle iff PAB, PBC and PCA together make up exactly ABC, with no excess. So we consider their *areas*. P is inside the triangle iff

  $$Area(PAB) + Area(PBC) + Area(PCA) = Area(ABC).$$

  Elegant! Do you like it? Neat, isn't it?

  However, there's a problem with it, even ignoring the fact that the areas might not be integers.[3] Consider what happens when A, B and C are *collinear*. This makes the triangle ABC degenerate into a line segment and have an area of 0. However, the question of whether P is in the triangle is still meaningful; it simply degenerates to the question of whether P is in the line segment or not.

[3] If the coordinates of the points are integers, can you remove the need for floats?

But our solution fails! If A, B, C and P are all collinear, then the areas of all four triangles are 0, so $0 + 0 + 0 = 0$, and P will always pass our test. But that's clearly wrong! It will also include points in the line determined by ABC but outside the line *segment*.

Things get even worse in the more degenerate case $A = B = C$; now every point in the plane will pass the check!

The lesson here is to be wary of vague intuitions; tricky cases always lurk around geometry problems!

One thing I like to say is that geometry algorithms are *easy to draw, but hard to code*.

- **Draw a lot!** Going off the previous motto, here's my advice: *draw a lot*! Actually, this is general advice which is just helpful in almost all problems, not just geometry ones. But this is geometry, so you can imagine that drawing would be even more helpful! Sometimes, merely drawing something on paper gives you insight on the setup that you wouldn't have gotten if you were just pushing algebraic symbols around. (Of course, algebraic manipulation sometimes help too. Just don't rely on it for everything.)

- **Transform the problem.** Here's another fairly general advice that's helpful even in non-geometry problems: *Simplify the problem by transforming it in some way.* If needed, you may untransform afterward to get the answer.

  In math, this is what happens when you write "without loss of generality" somewhere. In programming problems, it's what happens when we say "without loss of generality, assume that the array values are $0, 1, \ldots, v-1$" (a.k.a. coordinate compression), assuming you've proven that the values are distinct, and only the relative comparisons of the values matter.

  In geometry, you can do lots of easy transformations! For example, consider the point-in-triangle problem above. It can be hard to visualize four "generic" points in the plane, but if we notice that only their relative positions matter, then we can move them around in all sorts of ways to simplify things!

  For example, we can *translate* the whole system so that A becomes the origin. Translation is simply adding fixed values to the x and y coordinates, and undoing a translation is trivial, so we can get the original answer if needed.[4]

  Another thing we can do is to *rotate* things, so with A in the origin, we can also assume that B is on the positive x-axis, say. Finally, we can *reflect* things too, so we can say that C is above (or on) the x-axis.

  This doesn't fully solve the point-in-triangle problem, but now it's much simpler! For example, if $y_P < 0$, then we already know it's not in the triangle because we know the whole triangle is now above the x-axis!

  Again, *transform your geometry problem into something simpler, then untransform later if needed.* I can't stress this enough. This removes a lot of headache.

- **Learn the primitive operations.** Of course, the previous advice means that you should learn how to do these simple transformations properly. More generally, you need to learn the "primitive operations"—the standard, "easy" geometric things you can do.

- **Modular programming!** Implement everything as functions! (Especially the primitive operations.) Then use those functions as subroutines!

  Debugging a geometry program that consists of a bunch of random formulas is just painful. As mentioned above, geometry is already full of edge cases, so good coding habits is all the more important!

[4] In the case of point-in-triangle, there's no need to undo the translation, but in other problems, it may be needed.

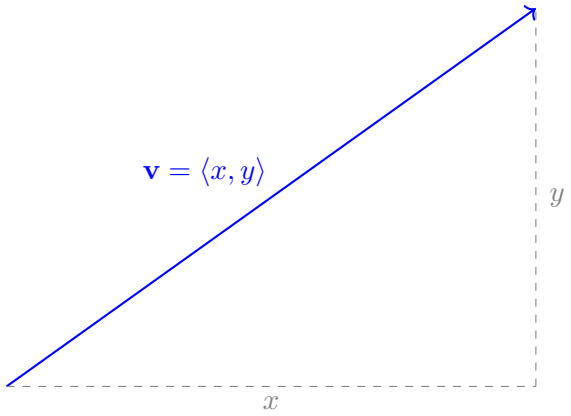
Modularize your code. Use functions extensively to organize your code. Keep track of all subproblems that you can do, from the high-level stuff ("compute the convex hull") to the low-level stuff ("given three points, are they collinear?"). If you have built up this library of functions at different levels of abstraction, you can then mostly piece them together to construct your solution. This is much better than doing everything ad hoc.

- **Object-based programming!** Related to the previous advice: implement your geometry objects as `class`es/`struct`s, then implement their useful methods. For example, implement a 2D Vector class, override the arithmetic operators (plus, minus, etc.), and implement other useful methods like `.cross`, `.dot`, `.abs`, `.rotate`, etc.

2 2D Vectors

A 2D **vector** represents a *direction* in the plane, with a *length*. It is usually written with angle brackets "$\langle x, y \rangle$" unlike points ("(x, y)"). At least for introductory texts, variables denoting vectors are usually written in bold, e.g., $\mathbf{v}$.[5]

Usually, you draw a vector on the plane as an arrow. For example, you can draw $\langle x, y \rangle$ as an arrow from $(0, 0)$ to (x, y). However, a vector doesn't have a specific relationship with the origin, so you don't have to draw it starting from there. For example, you can draw it as an arrow from $(1, 2)$ to $(x + 1, y + 2)$ too.



One question you may ask right now is: *Why vectors?* I can interpret this to mean three things, and I'll try to answer them:

- *Why use vectors at all? Why call a pair of coordinates $\langle x, y \rangle$ a "vector" when you can just deal with x and y separately as variables?* The answer to this is that by packaging the concept of "direction and length" as its own thing, we can think of it as a primitive object in its own right. This makes it a "smaller" and easier-to-grasp concept for our brains and lets it occupy a smaller part of our working memory.

  This also lets us not think about the vertical and horizontal components separately, and think of the *direction* $\mathbf{v}$ generically. This is helpful especially if the problem is not particularly tied to the Cartesian axes.

- *Why use vectors at all? I can do all of geometry with the things I learned from high school.* The answer is that vectors allow us to handle *all directions at once*, potentially avoiding certain edge cases if we use some approaches from high school instead. We'll elaborate on this below.

- *Why use vectors at all instead of just points? Aren't vectors and points the same thing—a pair of coordinates?* Well, the answer is that although they may be represented in code by the same thing (a pair of numbers), conceptually they're different. Points represent *locations*, while vectors represent *directions*. For example, it makes sense to add two vectors, but it doesn't make sense to add two points! (It makes sense to subtract two points, but the result is a vector.) We'll answer this question in more detail later.

[5] As for points: in text, they're sometimes denoted by capital letters such as P, but often they're just written with lowercase bold letters as well, like $\mathbf{p}$.

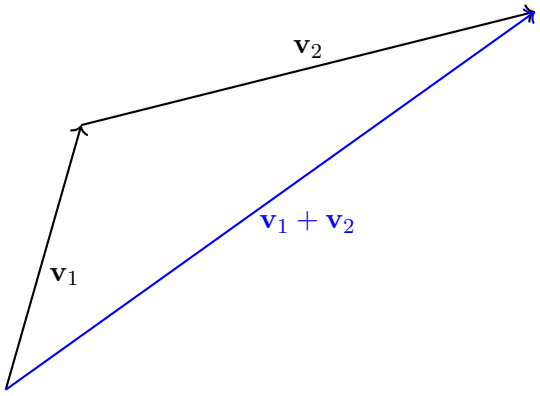
2.1 Vector Arithmetic

You can do arithmetic with vectors, along with other simple operations.

- You can *add* two vectors $\mathbf{v}_1 = \langle x_1, y_1 \rangle$ and $\mathbf{v}_2 = \langle x_2, y_2 \rangle$ by just adding the coordinates:

$$\mathbf{v}_1 + \mathbf{v}_2 = \langle x_1 + x_2, y_1 + y_2 \rangle.$$

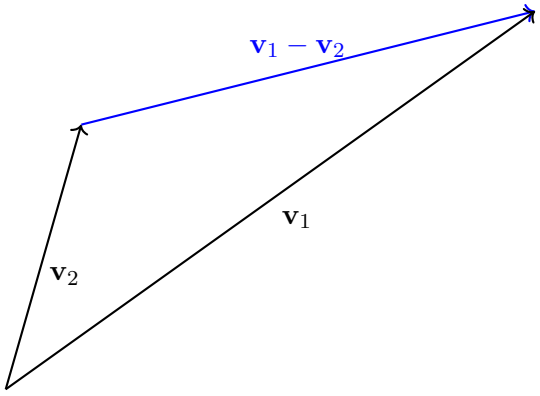
  Geometrically, this corresponds to drawing $\mathbf{v}_2$ starting at the tip of $\mathbf{v}_1$, and drawing the "overall arrow" from the base of $\mathbf{v}_1$ to the tip of $\mathbf{v}_2$.



- You can also *subtract* them:

$$\mathbf{v}_1 - \mathbf{v}_2 = \langle x_1 - x_2, y_1 - y_2 \rangle.$$

  Geometrically, subtraction is the inverse of addition, so $\mathbf{v}_1 - \mathbf{v}_2$ is the vector you need to add to $\mathbf{v}_2$ to form $\mathbf{v}_1$. Geometrically, you draw an arrow from the tip of $\mathbf{v}_2$ to the tip of $\mathbf{v}_1$ (assuming they start from the same point).



  Note that subtracting two *points* works the same way, but the result is a vector, so write it with angle brackets.

- You can also *scale* a vector $\mathbf{v} = \langle x, y \rangle$ by some scaling factor s. This is usually denoted the same way as multiplication—just put them together side-by-side (a.k.a., juxtapose):

$$s\mathbf{v} = \langle sx, sy \rangle.$$

  Geometrically, you really do scale the vector by the factor s. If s is negative, then you flip directions (and then scale by $|s|$). If $s = 0$, then it becomes the zero vector $\langle 0, 0 \rangle$.

  Since numbers in this context represent scaling, they're usually called **scalars**.

- You can take the *length* of a vector $\mathbf{v}$, and it results in a scalar, denoted $|\mathbf{v}|$. It's basically the Euclidean distance:
$$|\mathbf{v}| = \sqrt{x^2 + y^2}.$$
The squared length, $|\mathbf{v}|^2$, is often useful too since it's just $x^2 + y^2$, so it doesn't involve square roots, and it's an integer if the coordinates are integers as well.

Take a moment to really understand why our algebraic definitions correspond to the given geometrical explanations.

The geometric interpretation is good for our brains, and when drawing on paper, but the algebraic definition is how we implement it:

```
struct Vec {
    ll x, y;
    Vec(ll x = 0, ll y = 0): x(x), y(y) {}
    Vec operator+(const Vec& v) const { return {x + v.x, y + v.y}; }
    Vec operator-(const Vec& v) const { return {x - v.x, y - v.y}; }
    Vec operator*(ll s) const { return {s * x, s * y}; }
    double abs() const { return hypot(x, y); } // hypot is built-in
    ll abs2() const { return x * x + y * y; } // square of absolute value
    bool operator==(const Vec& v) const { return x == v.x and y == v.y; }
    bool operator!=(const Vec& v) const { return not (*this == v); }

    // more operations here later
};
```

However, although we implement things this way, *try not to think about the individual coordinates explicitly*!

2.2 Dot and cross products

Now we introduce two relatively nontrivial operations on vectors. There are at least two ways to multiply 2D vectors, the **dot product** denoted $\mathbf{v} \cdot \mathbf{w}$, and the **cross product** denoted $\mathbf{v} \times \mathbf{w}$. They both result in a scalar.[6] However, they mean different things.

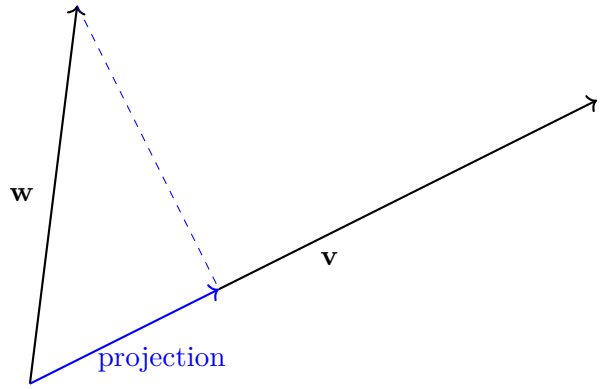
2.2.1 Dot product

Let's first look at the dot product. The **dot product** $\mathbf{v} \cdot \mathbf{w}$ is some sort of measure of how $\mathbf{v}$ and $\mathbf{w}$ are "aligned". It is still some sort of product, so you can think of it as the *product of the lengths of* $\mathbf{v}$ *and* $\mathbf{w}$—$|\mathbf{v}||\mathbf{w}|$—except we scale this down depending on how aligned $\mathbf{v}$ and $\mathbf{w}$ are.

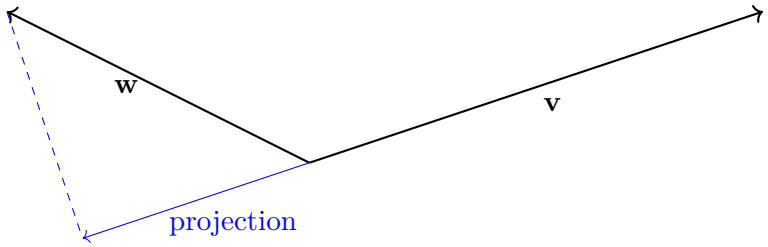
More precisely, the dot product is computed as follows. First, draw the arrows $\mathbf{v}$ and $\mathbf{w}$ from the origin. Then the dot product $\mathbf{v} \cdot \mathbf{w}$ is obtained by first *projecting* $\mathbf{w}$ to the line through $\mathbf{v}$, then multiplying it with the length of $\mathbf{v}$. In other words:
$$\mathbf{v} \cdot \mathbf{w} = |\mathbf{v}| project_{\mathbf{v}}(\mathbf{w}).$$

[6] If you know a bit about these things, then you might object that the cross product should return a *vector* instead of a *scalar*, and that it must be a 3D operation (or perhaps 7D). To which I say that by "cross product" I mean the *wedge product*, an appropriate generalization. In the 2D case, the wedge product of two vectors is a *top-dimensional multivector*. Top-dimensional multivectors have a natural bijection with scalars via the Hodge star operator. So with enough mathematical squinting, it is correct to say that "2D cross products are scalars."

Here's a diagram.

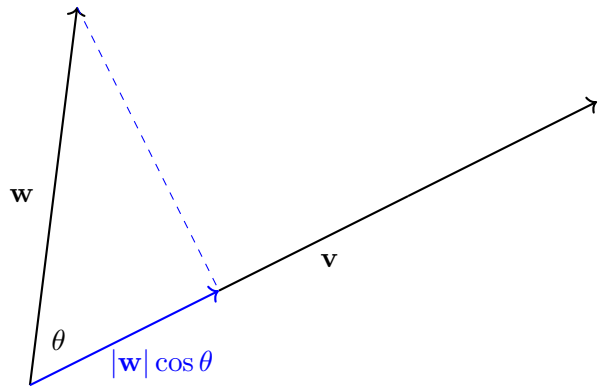


Note that $project_{\mathbf{v}}(\mathbf{w})$ may be negative, in case the projection points opposite $\mathbf{v}$. It is 0 if they're perpendicular.



To get a better grasp on the dot product, it's helpful to fix $\mathbf{v}$ and imagine $\mathbf{w}$ *rotating* around, and thinking about how the projection varies as you do the rotation. I suggest you do this!

We can compute this projection using trig as follows. Let θ be the *counterclockwise* angle from $\mathbf{v}$ to $\mathbf{w}$. (We can assume $0 \le \theta < 2\pi$, with $\theta = 0$ representing the case where $\mathbf{v}$ and $\mathbf{w}$ point to the same direction, and $\theta = \pi$ the case where they point in opposite directions.)



The above diagram shows that we have the formula

$$\mathbf{v} \cdot \mathbf{w} = |\mathbf{v}||\mathbf{w}| \cos\theta. \tag{1}$$

In particular, note that $\cos\theta = \cos(-\theta)$, so this formula shows that the dot product is **commutative**, i.e.,

$$\mathbf{v} \cdot \mathbf{w} = \mathbf{w} \cdot \mathbf{v},$$

which is not obvious at first glance when you only think about the dot product using the projection diagrams above.

While Equation 1 is a good formula, it's not the best formula to use since it uses cos, which means potentially having to use evil floats! Luckily, it turns out that there's a nicer formula, which we'll derive now.

To derive this formula, we first note a very nice property of the dot product: it is *linear in both arguments*! In other words, the functions $\mathbf{v} \mapsto \mathbf{v} \cdot \mathbf{w}$ (for a fixed $\mathbf{w}$) and $\mathbf{w} \mapsto \mathbf{v} \cdot \mathbf{w}$ (for a fixed $\mathbf{v}$) are both linear. A function f is **linear** if it satisfies the following two properties:

- (Scaling) $f(\alpha \mathbf{x}) = \alpha f(\mathbf{x})$ for any number α, and
- (Additivity) $f(\mathbf{x_1} + \mathbf{x_2}) = f(\mathbf{x_1}) + f(\mathbf{x_2})$.

A two-argument function that's linear in each argument is called **bilinear**, so the dot product is bilinear.

Being linear is very nice conceptually, but also computationally. The key property of linear transformations we will use is that it is completely determined by their behavior on any two vectors that *span* the whole plane. For example, suppose $\mathbf{e}_1$ and $\mathbf{e}_2$ span the plane, and we know the values $f(\mathbf{e}_1)$ and $f(\mathbf{e}_2)$. Then for every other vector $\mathbf{x}$, we can compute $f(\mathbf{x})$ as follows. First, since $\mathbf{e}_1$ and $\mathbf{e}_2$ span the plane, by definition, $\mathbf{x}$ can be expressed as $\alpha \mathbf{e}_1 + \beta \mathbf{e}_2$ for some numbers α and β. Then:

$$\begin{aligned} f(\mathbf{x}) &= f(\alpha \mathbf{e}_1 + \beta \mathbf{e}_2) \\ &= f(\alpha \mathbf{e}_1) + f(\beta \mathbf{e}_2) && \text{(additivity)} \\ &= \alpha f(\mathbf{e}_1) + \beta f(\mathbf{e}_2) && \text{(scaling).} \end{aligned}$$

Since $f(\mathbf{e}_1)$ and $f(\mathbf{e}_2)$ are both fixed, this means that $f(\mathbf{x})$ is also completely determined by these values!

A similar thing is true for bilinear functions. Suppose $g(\mathbf{x}, \mathbf{x}')$ is bilinear. Let $\mathbf{x} = \alpha \mathbf{e}_1 + \beta \mathbf{e}_2$ and $\mathbf{x}' = \alpha' \mathbf{e}_1 + \beta' \mathbf{e}_2$. Then

$$\begin{aligned} g(\mathbf{x}, \mathbf{x}') &= g(\alpha \mathbf{e}_1 + \beta \mathbf{e}_2, \mathbf{x}') \\ &= \alpha g(\mathbf{e}_1, \mathbf{x}') + \beta g(\mathbf{e}_2, \mathbf{x}') && \text{(linearity)} \\ &= \alpha g(\mathbf{e}_1, \alpha' \mathbf{e}_1 + \beta' \mathbf{e}_2) + \beta g(\mathbf{e}_2, \alpha' \mathbf{e}_1 + \beta' \mathbf{e}_2) \\ &= \alpha \alpha' g(\mathbf{e}_1, \mathbf{e}_1) + \alpha \beta' g(\mathbf{e}_1, \mathbf{e}_2) + \beta \alpha' g(\mathbf{e}_2, \mathbf{e}_1) + \beta \beta' g(\mathbf{e}_2, \mathbf{e}_2) && \text{(linearity).} \end{aligned}$$

Thus, the values of the function g are completely determined by the four special values $g(\mathbf{e}_1, \mathbf{e}_1)$, $g(\mathbf{e}_1, \mathbf{e}_2)$, $g(\mathbf{e}_2, \mathbf{e}_1)$, and $g(\mathbf{e}_2, \mathbf{e}_2)$.

We can use this to find a nice formula for dot products! First, we need two vectors $\mathbf{e}_1$ and $\mathbf{e}_2$ that span the plane. Then we just have to compute $\mathbf{e}_1 \cdot \mathbf{e}_1$, $\mathbf{e}_1 \cdot \mathbf{e}_2$, $\mathbf{e}_2 \cdot \mathbf{e}_1$, and $\mathbf{e}_2 \cdot \mathbf{e}_2$ to completely determine the dot product. A good set of spanning vectors are the "standard basis vectors" $\mathbf{e}_1 = \langle 1, 0 \rangle$ and $\mathbf{e}_2 = \langle 0, 1 \rangle$. Computing the dot products of these is pretty simple:

- $\mathbf{e}_1 \cdot \mathbf{e}_1 = 1$ because they point to the same direction, so the dot product is just the product of the lengths, which is $1 \cdot 1$. Similarly, $\mathbf{e}_2 \cdot \mathbf{e}_2 = 1$.
- $\mathbf{e}_1 \cdot \mathbf{e}_2 = 0$ because these vectors are perpendicular, which means the projection is zero. Similarly, $\mathbf{e}_2 \cdot \mathbf{e}_1 = 0$.

Using these four special values, we can now compute the dot product of any two vectors $\langle x, y \rangle$ and $\langle x', y' \rangle$. Note that

$$\langle x, y \rangle = x \langle 1, 0 \rangle + y \langle 0, 1 \rangle = x \mathbf{e}_1 + y \mathbf{e}_2.$$

Similarly, we have

$$\langle x', y' \rangle = x' \mathbf{e}_1 + y' \mathbf{e}_2.$$

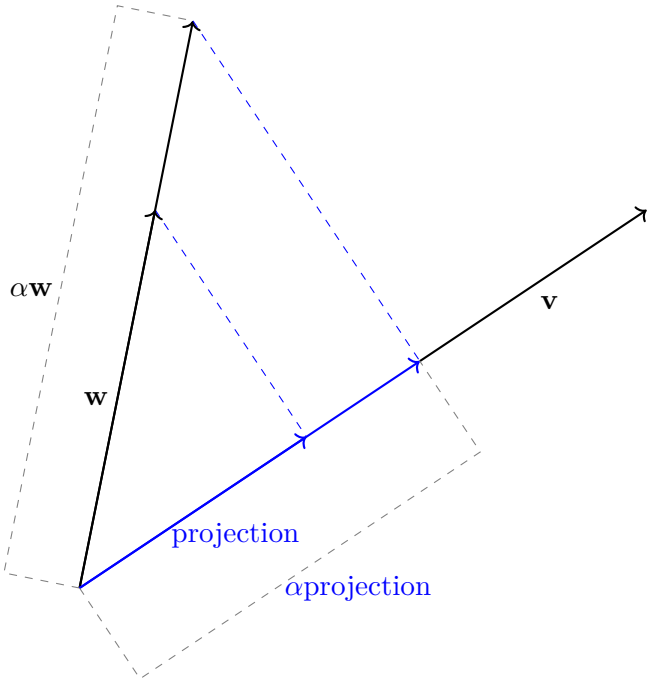
Therefore,

$$\begin{aligned}\langle x, y\rangle \cdot \langle x', y'\rangle &= (x\mathbf{e}_1 + y\mathbf{e}_2) \cdot (x'\mathbf{e}_1 + y'\mathbf{e}_2) \\ &= (xx')(\mathbf{e}_1 \cdot \mathbf{e}_1) + (xy')(\mathbf{e}_1 \cdot \mathbf{e}_2) + (yx')(\mathbf{e}_2 \cdot \mathbf{e}_1) + (yy')(\mathbf{e}_2 \cdot \mathbf{e}_2) \quad \text{(bilinearity)} \\ &= (xx')(1) + (xy')(0) + (yx')(0) + (yy')(1) \\ &= xx' + yy'.\end{aligned}$$

This is the nicer dot product formula we're looking for. Note that there are no trigonometric functions involved, only basic arithmetic! Thus, we discover the non-obvious fact that the dot product of two vectors with integer coordinates is also an integer!

Of course, the above only works if we assume that $\cdot$ is bilinear. So, we need to prove that the functions $\mathbf{v} \mapsto \mathbf{v} \cdot \mathbf{w}$ and $\mathbf{w} \mapsto \mathbf{v} \cdot \mathbf{w}$ are both linear. But actually, since $\cdot$ is commutative, it suffices to show that it's linear in one argument. So WLOG it's enough to show that $\mathbf{w} \mapsto \mathbf{v} \cdot \mathbf{w}$ is linear, for a fixed $\mathbf{v}$.[7]

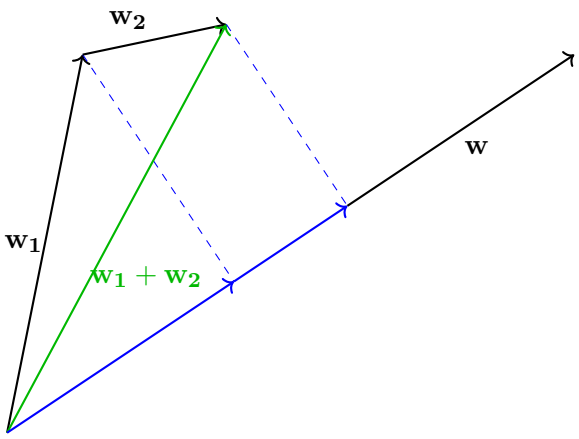
Now, we demonstrate the scaling property. Visually, this is pretty simple: $\alpha\mathbf{w}$ is just $\mathbf{w}$ scaled by a factor α, and you can easily see that the projection is also scaled by the factor α, which means that the dot product is also scaled by α.



Note the two similar triangles in this diagram. Also, note that this works even if $\alpha = 0$ or $\alpha < 0$.

Next, we demonstrate the additivity property. Visually, this is also pretty simple: the projection of $\mathbf{w_1} + \mathbf{w_2}$ can be seen visually as the sum of the projections of $\mathbf{w_1}$ and $\mathbf{w_2}$:

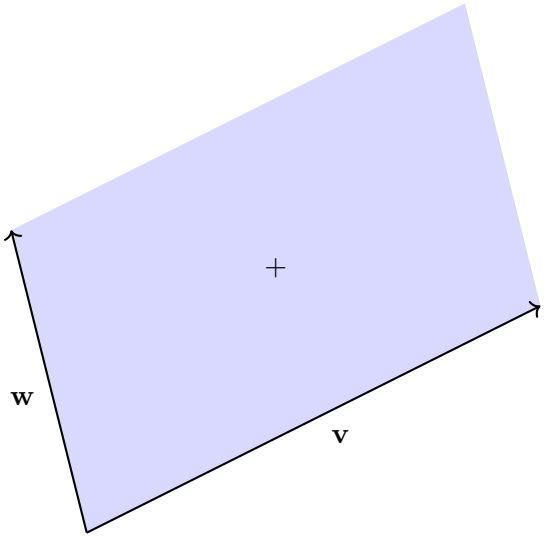
[7] We assume $\mathbf{v} \neq \mathbf{0}$ (where $\mathbf{0} = \langle 0, 0\rangle$ is the zero vector), otherwise the function always returns 0 so it's clearly linear.



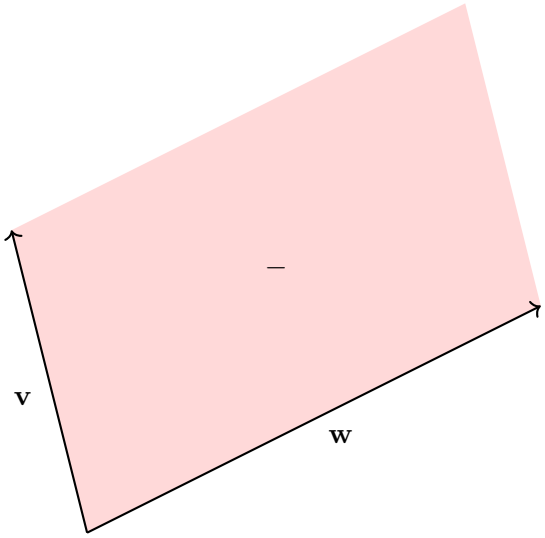
Technically, the above aren't completely rigorous proofs, only some sort of "visual demonstrations", but I hope that's enough to convince you![8]

2.2.2 Cross product

Next, let's look at the cross product. The **cross product** $\mathbf{v} \times \mathbf{w}$ represents a "signed area" (an "area" that has a sign); in particular, it's the signed area of the parallelogram formed by $\mathbf{v}$ and $\mathbf{w}$. The absolute value $|\mathbf{v} \times \mathbf{w}|$ is the actual area, but its sign depends on the orientation of the vectors. If $\mathbf{w}$ is counterclockwise of $\mathbf{v}$, then $\mathbf{v} \times \mathbf{w}$ is positive; if it's clockwise, then it's negative; and if it's aligned/anti-aligned, then the cross product is zero.



(a) $\mathbf{v} \times \mathbf{w}$ is positive



(b) $\mathbf{v} \times \mathbf{w}$ is negative

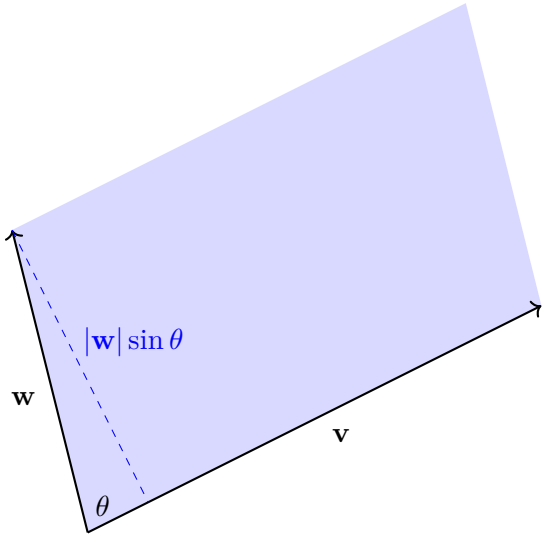
Just like with the dot product, you can get a better feel for the cross product by fixing $\mathbf{v}$ and *rotating* $\mathbf{w}$ around, and thinking about how the parallelogram (and its "sign") varies as you do the rotation. I suggest you do this!

You can use the "right-hand rule" to determine the sign. First, point the four fingers of your right hand towards the direction $\mathbf{v}$ then rotate your hand until they point to $\mathbf{w}$. If your thumb is pointing upwards/towards you as you do this, then the cross product is positive. If it's downwards/away from you, then the cross product is negative. This right-hand rule thing is better demonstrated in person![9]

[8] A rigorous proof is quite a bit trickier. Feel free to discover one on your own.

[9] A fond memory I have is seeing people waving their hands about during a physics exam; cross products are

You can see that $\mathbf{v} \times \mathbf{w}$ is basically $|\mathbf{v}||\mathbf{w}|$ multiplied by a factor that depends on the angle between them (just like the dot product). We can compute it by using the fact that the area of a parallelogram is its base times height:



Therefore, we have the formula

$$\mathbf{v} \times \mathbf{w} = |\mathbf{v}||\mathbf{w}| \sin \theta. \tag{2}$$

Note that the cross product is not commutative! Indeed, swapping the vectors reverses the orientation, so the cross product will negate. And we can see this with Equation 2. Note that $\sin(-\theta) = -\sin \theta$, so this formula shows that the cross product is **anticommutative**:

$$\mathbf{v} \times \mathbf{w} = -\mathbf{w} \times \mathbf{v}.$$

But again, Equation 2 summons the evil floating-point demons because of sin, so we want something better. Luckily, the cross product is **bilinear** too, which means we can determine it completely by computing its values at four special values: $\mathbf{e}_1 \times \mathbf{e}_1$, $\mathbf{e}_1 \times \mathbf{e}_2$, $\mathbf{e}_2 \times \mathbf{e}_1$, and $\mathbf{e}_2 \times \mathbf{e}_2$. Again using the standard basis ($\mathbf{e}_1 = \langle 1, 0 \rangle$ and $\mathbf{e}_2 = \langle 0, 1 \rangle$), we have:

- $\mathbf{e}_1 \times \mathbf{e}_1 = 0$ because the parallelogram is degenerate and has 0 area. Similarly, $\mathbf{e}_2 \times \mathbf{e}_2 = 0$.
- $\mathbf{e}_1 \times \mathbf{e}_2 = 1$ because the parallelogram is basically a unit square, and it's positive since $\mathbf{e}_2$ is counterclockwise of $\mathbf{e}_1$.
- $\mathbf{e}_2 \times \mathbf{e}_1 = -1$ because of anticommutativity, or noticing that it's the same unit square but with the opposite orientation.

Therefore, we can compute an arbitrary cross product as

$$\begin{aligned}
\langle x, y \rangle \times \langle x', y' \rangle &= (x\mathbf{e}_1 + y\mathbf{e}_2) \times (x'\mathbf{e}_1 + y'\mathbf{e}_2) \\
&= (xx')(\mathbf{e}_1 \times \mathbf{e}_1) + (xy')(\mathbf{e}_1 \times \mathbf{e}_2) + (yx')(\mathbf{e}_2 \times \mathbf{e}_1) + (yy')(\mathbf{e}_2 \times \mathbf{e}_2) \\
&= (xx')(0) + (xy')(1) + (yx')(-1) + (yy')(0) \\
&= xy' - yx'.
\end{aligned}$$

Note again how nice this is. No trig functions! And we also discover the nontrivial fact that the cross product of two vectors with integer coefficients is an integer!

Of course, we still have to prove that $\times$ is bilinear.

crucial in basic physics, e.g., electromagnetism.

Exercise 2.1. Convince yourself visually that $\times$ is bilinear.

Hint: You only need to do it for one argument because of anticommutativity.

2.2.3 Summary

Let's summarize what we have.

- The **dot product** represents "how aligned" two vectors are. The dot product of two vectors pointing in the same direction is simply the product of their lengths. The dot product decreases as the angle decreases. If they're perpendicular, the dot product is always 0. For obtuse angles, the dot product is negative.

  The dot product is commutative.

- The **cross product** represents a signed area; specifically, the signed area of a parallelogram formed by the two vectors. Its absolute value is the actual area, but its sign depends on the orientation of the vectors. If $\mathbf{w}$ is counterclockwise of $\mathbf{v}$, then $\mathbf{v} \times \mathbf{w}$ is positive; if it's clockwise, then it's negative; and if it's aligned/anti-aligned, then the cross product is zero.

  You can use the right-hand rule to determine the sign.

  The cross product is **not** commutative. It's anticommutative: $\mathbf{v} \times \mathbf{w} = -\mathbf{w} \times \mathbf{v}$.

- Both products are *bilinear*. So the dot product satisfies the following for all vectors $\mathbf{v}$, $\mathbf{w}$ and $\mathbf{x}$ and scalar s:

$$\begin{aligned} s(\mathbf{v} \cdot \mathbf{w}) &= (s\mathbf{v}) \cdot \mathbf{w} = \mathbf{v} \cdot (s\mathbf{w}) \\ \mathbf{v} \cdot (\mathbf{w} + \mathbf{x}) &= \mathbf{v} \cdot \mathbf{w} + \mathbf{v} \cdot \mathbf{x} \\ (\mathbf{v} + \mathbf{w}) \cdot \mathbf{x} &= \mathbf{v} \cdot \mathbf{x} + \mathbf{w} \cdot \mathbf{x}. \end{aligned}$$

  and the same for the cross product:

$$\begin{aligned} s(\mathbf{v} \times \mathbf{w}) &= (s\mathbf{v}) \times \mathbf{w} = \mathbf{v} \times (s\mathbf{w}) \\ \mathbf{v} \times (\mathbf{w} + \mathbf{x}) &= \mathbf{v} \times \mathbf{w} + \mathbf{v} \times \mathbf{x} \\ (\mathbf{v} + \mathbf{w}) \times \mathbf{x} &= \mathbf{v} \times \mathbf{x} + \mathbf{w} \times \mathbf{x}. \end{aligned}$$

  In higher math, things that are bilinear (or more generally, multilinear) are usually called *tensors*.

The dot and cross products have formulas involving cos and sin. However, I stress that you shouldn't be using those much. They are better as conceptual facts. Computationally, it's better to use the following formulas:

$$\begin{aligned} \langle x, y \rangle \cdot \langle x', y' \rangle &= xx' + yy' \\ \langle x, y \rangle \times \langle x', y' \rangle &= xy' - yx'. \end{aligned}$$

In code:

```
ll   dot(const Vec& v) const { return x * v.x + y * v.y; }
ll cross(const Vec& v) const { return x * v.y - y * v.x; }
```

If you're having trouble remembering the cross product, use the right-hand rule. (This will be mentioned again below.) You'll probably only have two candidates for the cross product formula, and only one of them agrees with the right-hand rule. (Use a concrete example like $\langle 1, 0 \rangle \times \langle 0, 1 \rangle = 1$.)

2.2.4 Applications

The dot and cross product are very useful primitives. Using them, we can now implement a few other primitive operations easily!

Exercise 2.2. Explain how to use the dot product to measure the length of a vector.

Exercise 2.3. Explain how to perform the parallel check ("are vectors $\mathbf{v}$ and $\mathbf{w}$ parallel/antiparallel?") using the cross product.

Exercise 2.4. Explain how to perform the collinearity check ("are A, B and C collinear?") using the cross product.

Exercise 2.5. Explain how to perform the perpendicularity check ("are vectors $\mathbf{v}$ and $\mathbf{w}$ perpendicular?") using the dot product.

Exercise 2.6. Suppose you've discovered (using Exercise 2.3, with a cross product) that $\mathbf{v}$ and $\mathbf{w}$ are either parallel or antiparallel. How do we distinguish between these two cases?

Hint: Use the dot product.

Exercise 2.7. Explain how to use the cross product to solve the "counterclockwise check" primitive operation: Given points A, B and C, is C to the left of B when viewed from the vantage point of A?

By the way, vectors are intimately tied with the idea of **linear transformations**. A good resource for this would be 3Blue1Brown's Essence of Linear Algebra video series: [2].

2.3 Why Vectors?

Here, I'll try to answer the following question:

> *Why use vectors at all? I can do all of geometry with the things I learned from high school.*

We'll use the following problem as motivation. It's one of the primitive operations I mentioned above.

Problem 2.1 (Collinearity check)**.** Given three points A, B and C, determine if they are collinear or not.

Let's solve it first with some high-school techniques, then with vectors, then see which one's better. (Can you guess which one?)

- **High-school way.** We want to check whether AB and BC are part of the same line. Since they already intersect, all that remains is to check that they have the same *slopes*.

The slope between two points $A = (x_a, y_a)$ and $B = (x_b, y_b)$ is given by

$$\frac{y_b - y_a}{x_b - x_a},$$

so we now know that A, B and $C = (x_c, y_c)$ are collinear iff

$$\frac{y_b - y_a}{x_b - x_a} = \frac{y_c - y_b}{x_c - x_b}.$$

So we're done...right?

Unfortunately, no! There are at least two issues:

- Note that we performed division twice. However, we know that *floats are evil*! This solution can run into some rounding errors.
- More importantly, the solution suffers from potential division by zero, which can happen if B is placed in the same vertical line as A or C. This is trouble! To fix this, we'd have to consider vertically-placed points as special cases. Messy!

- **Vector way.** Let's use the variables **a**, **b** and **c** for the points. We want to check that these three are collinear. It's the same as saying the vectors $\mathbf{v} := \mathbf{b} - \mathbf{a}$ and $\mathbf{w} := \mathbf{c} - \mathbf{b}$ point in the same direction (up to scaling). But this is easily checked with the cross product! The vectors **v** and **w** are collinear if and only if

$$\mathbf{v} \times \mathbf{w} = 0.$$

This is much simpler, and doesn't involve special cases! Vertical lines are handled neatly here.

Now it turns out that for something as simple as collinearity checking, there's a way of dealing with the problem mentioned above in the high-school way. And of course, there should be a way! However, this is unnecessary since we can just use vectors to simplify the task, and we don't have to deal with vertical lines as a special case. And you can imagine that for harder problems, not using vectors is basically making things unnecessarily harder for you.

If you actually compare the vector way with a corrected version of the high-school way, you'll see that they're performing essentially the same computation. However, we derived it using vectors without much difficulty, while we've had to struggle with the high-school way a bit.

Here's another one, again one of the primitive operations above.

Problem 2.2 (Line-line intersection)**.** Given four points A, B, C and D, find the intersection of lines AB and CD.

Let's solve it both ways again.

- **High-school way.** How do we represent lines? Maybe "$y = mx + b$" will do. So we need to find the (m, b) values for lines AB and CD, say (m_1, b_1) and (m_2, b_2), and once we do that, we can solve for the intersection by solving a system of two linear equations, as follows:

$$\begin{aligned} m_1 x + b_1 = y &= m_2 x + b_2 \\ b_1 - b_2 &= (m_2 - m_1)x \\ x &= \frac{b_1 - b_2}{m_2 - m_1}. \end{aligned}$$

Substituting this to $y = m_1x + b_1$ (or the other line) gives us the y-coordinate of the intersection.

So all that remains is to find the "$y = mx + b$" form of the line determined by two points $A = (x_1, y_1)$ and $B = (x_2, y_2)$. Substituting this gives us two equations

$$y_1 = mx_1 + b$$
$$y_2 = mx_2 + b.$$

This is again a system of two linear equations, so we can solve it as follows (first by subtracting the equations):

$$y_1 - y_2 = m(x_1 - x_2)$$
$$m = \frac{y_1 - y_2}{x_1 - x_2}.$$

Finally, we can get b easily once we get m. So it seems like we're done.

But like before, we're not! Again, there are several issues:

- We didn't handle the case where the two lines are parallel/coinciding. So we need to check first whether $m_1 = m_2$; if so, we exit immediately.
- More importantly, the form $y = mx + b$ fails for vertical lines! In this case, we'll get infinite slope, and the line would be $y = \infty x + b$ which doesn't make much sense. The equation for vertical lines looks like "$x = c$", so to fix the solution, we'll have to handle vertical lines as special cases again!

It turns out that for line-line intersection, there are other ways to fix the problem, e.g., by using a different equation form like $ax + by = c$. But why do that, when we can just use vectors instead?

- **Vector way.** There's a nice way to represent a line going through two points $\mathbf{p}$ and $\mathbf{q}$. The "direction" of the line is given by the vector $\mathbf{v} := \mathbf{q} - \mathbf{p}$. So the points on the line are precisely those that can be obtained by starting from point $\mathbf{p}$ and walking along $\mathbf{v}$ by some amount. Thus, the line can be expressed as follows:

$$\{\mathbf{p} + \mathbf{v}t \mid t \in \mathbb{R}\}.$$

Let's use the variables $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$ and $\mathbf{d}$ for the points. Let $\mathbf{v} = \mathbf{b} - \mathbf{a}$ be the direction vector of the first line, so the first line is simply

$$\{\mathbf{a} + \mathbf{v}t \mid t \in \mathbb{R}\}.$$

We want to find a number t such that $\mathbf{a} + \mathbf{v}t$ is on the line determined by $\mathbf{c}$ and $\mathbf{d}$, which is equivalent to saying that these three points are collinear. But we already know how to check for collinearity! These points are collinear if and only if

$$(\mathbf{a} + \mathbf{v}t - \mathbf{c}) \times (\mathbf{d} - \mathbf{c}) = 0.$$

We can now solve for t algebraically!

$$\begin{aligned}
(\mathbf{a} + \mathbf{v}t - \mathbf{c}) \times (\mathbf{d} - \mathbf{c}) &= 0 \\
(\mathbf{v}t + \mathbf{a} - \mathbf{c}) \times (\mathbf{d} - \mathbf{c}) &= 0 \\
(\mathbf{v}t) \times (\mathbf{d} - \mathbf{c}) + (\mathbf{a} - \mathbf{c}) \times (\mathbf{d} - \mathbf{c}) &= 0 \\
t(\mathbf{v} \times (\mathbf{d} - \mathbf{c})) &= -(\mathbf{a} - \mathbf{c}) \times (\mathbf{d} - \mathbf{c}) \\
t &= \frac{-(\mathbf{a} - \mathbf{c}) \times (\mathbf{d} - \mathbf{c})}{\mathbf{v} \times (\mathbf{d} - \mathbf{c})}.
\end{aligned}$$

These calculations are justified by the fact that $\times$ is bilinear. Now that we have the t, we can substitute it to $\mathbf{a} + \mathbf{v}t$ to get the point of intersection!

The only time this can go wrong is if the denominator is zero, i.e.,

$$\mathbf{v} \times (\mathbf{d} - \mathbf{c}) = 0.$$

But remembering that $\mathbf{v} = \mathbf{b} - \mathbf{a}$ is the direction vector of the first line and $\mathbf{d} - \mathbf{c}$ is the direction vector of the second, we see that this is zero precisely when the two lines are pointing in the same direction, i.e., parallel or coinciding!

I hope I've convinced you that vectors are worthwhile! We don't have to special-case vertical or horizontal lines with this approach, reducing the number of edge cases we need to consider. Computational geometry is already full of edge cases, so we shouldn't unnecessarily add more!

2.4 Simple linear transformations

Remember one of our earlier lessons: *transform then un-transform*? It's very useful! But of course, it's only possible if you actually know how to do the transforms and un-transforms in the first place. We will discuss a couple here.

2.4.1 Translation

This is one of the easiest ones. **Translation** by a vector $\mathbf{v}$ means that we move everything along the vector $\mathbf{v}$. (Note that a vector not only has a direction but also has a length, so a vector completely specifies a translation.) This is useful if only the *relative* positions of the stuff matter. Examples of problems where only relative positions matter are: computing the area of a polygon, determining if a point is in a polygon, finding the intersection of two lines (you have to un-translate, of course). Actually, this is true in most computational geometry tasks; after all, coordinates are usually arbitrary, so $(0, 0)$ isn't particularly special (unless the problem statement gives it a special meaning, of course).

The transformation is pretty simple. If we denote translation by $\mathit{Trans}_{\mathbf{v}}$, then we have:

$$\mathit{Trans}_{\mathbf{v}}(\mathbf{x}) = \mathbf{x} + \mathbf{v}.$$

The un-transform of $\mathit{Trans}_{\mathbf{v}}$—a.k.a., its inverse $\mathit{Trans}_{\mathbf{v}}^{-1}$—is simply subtracting $\mathbf{v}$:

$$\mathit{Trans}_{\mathbf{v}}^{-1}(\mathbf{x}) = \mathbf{x} - \mathbf{v}.$$

We can also write this as:

$$\mathit{Trans}_{\mathbf{v}}^{-1} = \mathit{Trans}_{-\mathbf{v}}.$$

2.4.2 Scaling

Scaling by a scalar factor s is basically resizing. However, it may also involve flipping if s is negative. This is useful if only the *relative* distances matter in your problem, not the absolute distances—for example, determining if a point is in a polygon. Then scaling may allow you to make one of the lengths equal to 1, for example.

The simplest version involves scaling relative to the origin. In this case, we simply multiply by s:

$$\mathit{Scale}_s(\mathbf{x}) = \mathbf{x} \cdot s.$$

Its un-transform is division:

$$Scale_s^{-1}(\mathbf{x}) = \mathbf{x}/s.$$

Clearly, this is only possible if $s \neq 0$. We can also write this as

$$Scale_s^{-1} = Scale_{1/s}.$$

This is scaling relative to the *origin*, which you can check by noticing that the origin stays fixed at the origin. But what if we want to scale relative to a different point, say $\mathbf{p}$? In this case, we can apply the transform-untransform technique: simply translate things so that $\mathbf{p}$ moves to the origin, then do the scaling (which we already know), then untranslate!

Formally, we do this:

$$Scale_{s,\mathbf{p}}(\mathbf{x}) = Trans_{-\mathbf{p}}^{-1}(Scale_s(Trans_{-\mathbf{p}}(\mathbf{x}))).$$

In terms of functions, we can use the function composition notation:

$$Scale_{s,\mathbf{p}} = Trans_{-\mathbf{p}}^{-1} \circ Scale_s \circ Trans_{-\mathbf{p}}.$$

In terms of code, I suggest implementing $Trans_{\mathbf{v}}$ and $Scale_s$ as functions, and simply doing the composition above to implement $Scale_{s,\mathbf{p}}$, rather than "simplifying the algebra" yourself to obtain an expression to compute it "all at once".[10] There isn't much reason to do yourself what the computer is good at doing—that is, *compute*!

As you can see, implementing the general scaling function becomes easy after you implement the special case at the origin. The transform-untransform technique is really useful!

2.4.3 Reflection

Now, we discuss **reflection**. Generally, we specify a line to reflect on. The simplest cases are when we just want to reflect about the x or y axis. This is pretty simple: it only involves flipping the sign of some coordinate.

- Reflection about the x-axis: $Refl_{x\text{-axis}}(x, y) = (x, -y)$.
- Reflection about the y-axis: $Refl_{y\text{-axis}}(x, y) = (-x, y)$.

Note that reflecting about the x-axis flips the sign of the y-coordinate, or vice versa. Don't be confused! One way I remember which is which is to just see what happens with the point $(1, 1)$.

You can also reflect about the diagonal line $y = x$ relatively easily by swapping coordinates:

$$Refl_{\text{line } y=x}(x, y) = (y, x).$$

What about other lines? For example, consider the line through the origin at direction $\mathbf{v}$. How do we reflect on that? Well, we can do the transform-untransform technique: we rotate things so that $\mathbf{v}$ ends up in one of the axes, then we reflect, then we unrotate. However, we haven't discussed rotation yet (though we will pretty soon), so let's derive this in a different way.

The key thing we will use is that reflection is *linear*, that is, any reflection R satisfies

$$R(\mathbf{v} + \mathbf{w}) = R(\mathbf{v}) + R(\mathbf{w})$$
$$R(s \cdot \mathbf{v}) = s \cdot R(\mathbf{v}).$$

[10] In cases where you're forced to use floating point, you may have to simplify the algebra to reduce floating-point errors—because less operations usually means better accuracy.

Geometrically, this means that reflection sends lines to lines. (Check this!)

Also, note that reflection about $\mathbf{v}$ should send $\mathbf{v}$ to itself. On the other hand, any vector $\mathbf{w}$ perpendicular to $\mathbf{v}$ must be sent to $-\mathbf{w}$. We can get one such perpendicular vector by rotating the vector by 90 degrees. (And we can check that they're indeed perpendicular with the dot product: $\mathbf{v} \cdot \mathbf{w} = 0$.) But now, note that any vector $\mathbf{x}$ can be represented by a linear combination of $\mathbf{v}$ and $\mathbf{w}$. Thus, we can now fully describe reflection since we've already specified it for $\mathbf{v}$ and $\mathbf{w}$.

Suppose $\mathbf{x} = \alpha\mathbf{v} + \beta\mathbf{w}$. Then

$$\begin{aligned} \mathit{Refl}_{\mathbf{v}}(\mathbf{x}) &= \mathit{Refl}_{\mathbf{v}}(\alpha\mathbf{v} + \beta\mathbf{w}) \\ &= \alpha\mathit{Refl}_{\mathbf{v}}(\mathbf{v}) + \beta\mathit{Refl}_{\mathbf{v}}(\mathbf{w}) && \text{(linearity)} \\ &= \alpha\mathbf{v} + \beta(-\mathbf{w}) \\ &= \alpha\mathbf{v} - \beta\mathbf{w}. \end{aligned}$$

Adding $\mathbf{x} = \alpha\mathbf{v} + \beta\mathbf{w}$ to this, we get:

$$\begin{aligned} \mathit{Refl}_{\mathbf{v}}(\mathbf{x}) + \mathbf{x} &= 2\alpha\mathbf{v} \\ \mathit{Refl}_{\mathbf{v}}(\mathbf{x}) &= 2\alpha\mathbf{v} - \mathbf{x}. \end{aligned}$$

This is good. It means that we will be able to reflect if we're able to compute α. We can do that using the dot product! If we compute $\mathbf{x} \cdot \mathbf{v}$, we get

$$\begin{aligned} \mathbf{x} \cdot \mathbf{v} &= (\alpha\mathbf{v} + \beta\mathbf{w}) \cdot \mathbf{v} \\ &= \alpha\mathbf{v} \cdot \mathbf{v} + \beta\mathbf{w} \cdot \mathbf{v} \\ &= \alpha|\mathbf{v}|^2 + \beta \cdot 0 && \text{(remember that } \mathbf{v} \text{ is perpendicular to } \mathbf{w}) \\ &= \alpha|\mathbf{v}|^2 \\ \alpha &= \frac{\mathbf{x} \cdot \mathbf{v}}{|\mathbf{v}|^2}. \end{aligned}$$

We crucially used the fact that $\mathbf{v}$ and $\mathbf{w}$ are perpendicular, so the dot product is zero, which deletes the β term. Finally, substituting this α, we get

$$\begin{aligned} \mathit{Refl}_{\mathbf{v}}(\mathbf{x}) &= 2\alpha\mathbf{v} - \mathbf{x} \\ &= 2\frac{\mathbf{x} \cdot \mathbf{v}}{|\mathbf{v}|^2}\mathbf{v} - \mathbf{x}. \end{aligned}$$

This is now our reflection formula!

Okay, that derivation was a little long, and the final result isn't quite easy to remember, so **I suggest not memorizing it**! I think a better takeaway for you here is to note that the dot and cross products are very powerful; they will be your ~~bread and butter~~ kanin and toyo when doing these geometry calculations! Their bilinearity property is very convenient for calculation!

Anyway, we now know how to reflect about any arbitrary line through the origin. But what about lines not going through the origin? Well, we can do that simply by translating and then untranslating! I'll leave the details to you.

2.4.4 Rotation

Rotation involves, well, rotating stuff around a point. In its general form, you specify some amount to rotate, and also the point $\mathbf{p}$ to rotate about. However, using the transform-untransform technique, we can translate stuff so that we're just rotating about the origin! So really, the only thing we need to describe is how to rotate about the origin.

The main issue is: how do we represent the amount to rotate with? There are a few subtleties to this, which we'll get to later, but for now, let's note that it's easy to rotate by multiples of 90 degrees. Rotating by 90 degrees (counterclockwise) is simply

$$Rot_{90^\circ}(x, y) = (-y, x).$$

(Check this!) So rotating 90 degrees repeatedly, we get

$$(x, y) \to (-y, x) \to (-x, -y) \to (y, -x) \to (x, y) \to \dots$$

which repeats every four rotations, as expected. Also, note that rotation by 180 degrees is the same as scaling by -1.

Now, what about other angles? Well, there's a problem. If you specify an angle, then generally, you need to compute its cos and sin to get the coordinates of the rotated point. But floats are evil! So this is probably not what we want.

Here's another idea. Let $\mathbf{u}$ represent the direction that we want to rotate by. Specifically, the angle that $\mathbf{u}$ makes with the positive x-axis will be our angle of rotation. Let's now figure out the rotation formula.

The key thing we will use is that rotation is *linear*, just like reflection. (Indeed, you can check that rotation sends lines to lines.)

But now, any vector $\mathbf{x} = \langle x, y\rangle$ can be represented as

$$\mathbf{x} = \langle x, y\rangle = x\langle 1, 0\rangle + y\langle 0, 1\rangle.$$

Therefore, for our rotation $Rot_{\mathbf{u}}$, we have

$$\begin{aligned} Rot_{\mathbf{u}}(\mathbf{x}) &= Rot_{\mathbf{u}}(x\langle 1, 0\rangle + y\langle 0, 1\rangle) \\ &= xRot_{\mathbf{u}}(\langle 1, 0\rangle) + yRot_{\mathbf{u}}(\langle 0, 1\rangle) \qquad \text{(linearity)}. \end{aligned}$$

Thus, all we need to do is figure out what the rotation does to the two special vectors $\langle 1, 0\rangle$ and $\langle 0, 1\rangle$. Writing $\mathbf{u} = \langle a, b\rangle$,

- $Rot_{\mathbf{u}}(\langle 1, 0\rangle)$. Note that $\langle 1, 0\rangle$ points to the positive x-axis, which means it must end up at $\mathbf{u} = \langle a, b\rangle$! (Well, we need to ensure that it still has length 1...more on this later.)
- $Rot_{\mathbf{u}}(\langle 0, 1\rangle)$. Note that $\langle 0, 1\rangle$ is just the 90-degree rotation of $\langle 1, 0\rangle$, so its rotation must also be 90-degrees ahead of the previous result. But we already know how to rotate by 90 degrees! Thus, the result is $\langle -b, a\rangle$. (Again, we need to ensure that $|\mathbf{u}| = 1$.)

This only works if $\mathbf{u}$ has unit length, but that's easy to arrange for: simply scale by $\frac{1}{|\mathbf{u}|}$, and we get a unit-length vector! (This requires floats or fractions though, which you might not want. So one thing you can do is to just not scale at all, and accept that your rotation also scales by some amount. Doing it this way, we can stay in integer coordinates!) Therefore, assuming $\mathbf{u}$ has unit length, that is, $a^2 + b^2 = 1$, we have

$$\begin{aligned} Rot_{\mathbf{u}}(\mathbf{x}) &= xRot_{\mathbf{u}}(\langle 1, 0\rangle) + yRot_{\mathbf{u}}(\langle 0, 1\rangle) \\ &= x\langle a, b\rangle + y\langle -b, a\rangle \\ &= \langle ax - by, bx + ay\rangle, \end{aligned}$$

and we can now do rotation!

Again, I suggest not memorizing this formula—you might just get confused with which term gets a $+$ or a $-$ sign. Instead, we note that the two terms are eerily similar to the formula for the dot and cross product!

Indeed, if we let $\mathbf{w} = \langle a, -b \rangle$ be the reflection of $\mathbf{u}$ across the x-axis, then we simply have

$$\mathbf{w} \cdot \mathbf{x} = ax + (-b)y = ax - by$$
$$\mathbf{w} \times \mathbf{x} = ay - (-bx) = bx + ay.$$

In other words, we can express the rotation of $\mathbf{x} = \langle x, y \rangle$ as:

$$Rot_{\mathbf{u}}(\mathbf{x}) = \langle \mathbf{w} \cdot \mathbf{x}, \mathbf{w} \times \mathbf{x} \rangle,$$

where $\mathbf{w} = Refl_{x\text{-axis}}(\mathbf{u})$ (simply flip the y-coordinate). Note that $\mathbf{u}$ has to be a unit-length vector ($|\mathbf{u}| = 1$) for this to be a pure rotation—if not, it's a combination of rotation (by the angle of $\mathbf{u}$) and a scaling of $|\mathbf{u}|$.

Exercise 2.8. Show that the following transform

$$(x, y) \to (x - y, x + y)$$

is just rotation by 45-degrees and scaling by $\sqrt{2}$.

Exercise 2.9. What does

$$(x, y) \to (x + y, x - y)$$

do?

Again, the way to remember which is which is not by memorizing, but drawing some examples on paper and seeing what happens. Drawing is key!

Exercise 2.10. Show that

$$Rot_{\mathbf{u}}^{-1}(\mathbf{x}) = \langle \mathbf{u} \cdot \mathbf{x}, \mathbf{u} \times \mathbf{x} \rangle.$$

In particular, "inverse rotation" might be easier to remember than direct rotation because we don't have to compute $\mathbf{w}$, the reflection about the x-axis! So I recommend remembering this instead of direct rotation.

What about general angles θ? Well, we can reduce it to the above by noticing that rotation by an angle θ is, using our notation above, the same as rotating with respect to $\mathbf{u} = \langle \cos\theta, \sin\theta \rangle$:

$$Rot_{\theta} = Rot_{\langle \cos\theta, \sin\theta \rangle}.$$

If we want to write that out explicitly, simply substitute to get:

$$Rot_{\theta}(\langle x, y \rangle) = \langle x\cos\theta - y\sin\theta, x\sin\theta + y\cos\theta \rangle.$$

But again, introducing these trig functions also introduces the evil floating-point numbers, so be careful! I suggest using the previous form with $\mathbf{u}$—if it has integer coordinates, then you stay at integer coordinates (if you don't need to make $\mathbf{u}$ a unit-length vector).

2.4.5 Remembering these formulas

So there were quite a few formulas above. How do you remember them? The answer is: you don't...mostly.

There are many many ways to rederive them. Of course, you can just memorize our derivations above. But there's an easier way. The key is remembering that rotation and reflection (or at least, their versions about the origin) are *linear*. One key property of linear transformations is that their behavior can be completely determined by their behavior on any two special vectors that *span* the whole plane. For example, note that $\langle 1, 0\rangle$ and $\langle 0, 1\rangle$ span the whole plane, since any vector can be represented as a "weighted sum" of these two. (Usually called "*linear combination*".) In general, if $\mathbf{u}$ and $\mathbf{v}$ are two vectors that span the plane, then any vector $\mathbf{x}$ can be written as $\alpha\mathbf{u} + \beta\mathbf{v}$, so for any linear transformation T,

$$\begin{aligned} T(\mathbf{x}) &= T(\alpha\mathbf{u} + \beta\mathbf{v}) \\ &= \alpha T(\mathbf{u}) + \beta T(\mathbf{v}) \qquad \text{(linearity)}. \end{aligned}$$

In other words, once we specify $T(\mathbf{u})$ and $T(\mathbf{v})$, we've completely specified T! (We usually call this "extending by linearity".)

You can use this property to rederive the formulas for rotation and reflection, by simply specifying its behavior on two well-chosen vectors that span the plane.

- For rotations, a good choice are just the standard basis vectors $\langle 1, 0\rangle$ and $\langle 0, 1\rangle$. The first one is trivial to rotate, and the second is just a 90-degree rotation of that!
- For reflections say about $\mathbf{v}$, a good choice would be $\mathbf{v}$ and its 90-degree rotation.
- Alternatively, for reflections, just remember how to reflect in a very special case—say about the x-axis—and then just rotate-unrotate to that case!

Also, you probably want to remember that 90-degree counterclockwise rotation is simply $(x, y) \to (-y, x)$. But even if you forget this formula, you can just rederive it easily as well! You'll probably at least remember that it's one of $(x, y) \to (\pm x, \pm y)$ or $(x, y) \to (\pm y, \pm x)$, so you have 8 candidates to choose from. Well, you can check which one it is by drawing the point $(1, 2)$ on paper, noticing that it must be sent to $(-2, 1)$ (verify!), and noticing that only one of these works, namely $(x, y) \to (-y, x)$. In other words, drawing saves the day! Alternatively, you can just check which one sends $(1, 0)$ to $(0, 1)$ and $(0, 1)$ to $(-1, 0)$; again, only one will work!

Sometimes just to prevent stupid sign errors like this, I add `assert`s like this at the beginning of my code:

```
Vec e1 = {1, 0}, e2 = {0, 1}, e3 = {-1, 0};
assert(e1.rotate90ccw() == e2);
assert(e2.rotate90ccw() == e3);
```

The running time is negligible, but this allows us to catch the possible sign error. You can do a similar thing for other functions/methods you're worried about. For example, you can also do something like this:

```
Vec e1 = {1, 0}, e2 = {0, 1};
assert(e1.cross(e2) == 1);
assert(e2.cross(e1) == -1);
```

So the main lesson here is that you don't have to memorize much, as long as you know the key properties of these geometric tools (and programming tools).

2.4.6 Transform-untransform

In the above, we used the "transform-untransform" step to reduce the general case to a simpler case (which is easier to handle of course). This is such a useful technique in computational geometry that it's worth mentioning again.

In general, the form of "transform-untransform" is as follows:

$$T^{-1}FT. \tag{3}$$

Here, F represents your solution to the simpler case, and T represents the "transform" which reduces the general case to the simpler case (and T^{-1} is the un-transform of course). You'll sometimes hear this form as a "conjugation."

You will encounter conjugation as you study more math. For example, in group theory, sometimes conjugation is introduced as the form (3) without motivation, say

$$\phi_T(F) = TFT^{-1}.$$

And then afterward, a couple of nice properties are proven, such as

$$\begin{aligned}
\phi_T(FG) &= \phi_T(F)\phi_T(G) \\
\phi_{TU}(F) &= \phi_T(\phi_U(F)) \\
\phi_T(\text{identity}) &= \text{identity} \\
\phi_T(F^{-1}) &= \phi_T(F)^{-1}.
\end{aligned}$$

If it's the first time you've seen conjugation and it wasn't motivated properly, then these formulas may seem magical. You might think: Why is this random expression TFT^{-1} so nice like this? But once you notice that it's just the abstract form of our "transform-untransform" idea, then the equations above not only becomes clear, but also natural, even. For example, you can read them as follows (note that you should read TFT^{-1} right-to-left to analogize with function composition):

- "$\phi_T(F) = TFT^{-1}$." Suppose I can do F. How can I do F on a setup that has been transformed via T? Well, I can T-untransform first, then do F, then T-transform, hence "TFT^{-1}". So we can summarize "$\phi_T(F)$" as "Do F on the T-transformed world."
- "$\phi_T(FG) = \phi_T(F)\phi_T(G)$." I want to do "$FG$" on the T-transformed world. Doing FG is the same as doing G then doing F, so naturally, doing G then F on the T-transformed world ("$\phi_T(FG)$") is the same as doing G on the T-transformed world then doing F on the transformed world ("$\phi_T(F)\phi_T(G)$").

I invite you to find natural interpretations of the other formulas!

3 Lines and Line Segments

Now, let's move on to lines. From high school, you usually represent them as $ax + by = c$, or $y = mx + b$, or maybe even $\frac{x}{a} + \frac{y}{b} = 1$ (or $= 0$). However, as demonstrated above, it's better to represent lines with vectors.

As usual, I recommend drawing stuff on your scratch paper as you read the following, since it makes things easier to follow.

3.1 Representation

We've already seen one above. Given a point $\mathbf{p}$ and a direction vector $\mathbf{v}$, we can represent the line going through $\mathbf{p}$ along the direction $\mathbf{v}$ as

$$\{\mathbf{p} + \mathbf{v}t \mid t \in \mathbb{R}\}. \tag{4}$$

Another common way to represent a line is to just specify two points in it, say $(\mathbf{p}, \mathbf{q})$, since any line is determined by two points. It should be easy to convert between the previous representation and this one.

There's another way. Given a point $\mathbf{p}$ and a vector $\mathbf{w}$, what's the set of all points $\mathbf{x}$ such that $\mathbf{x} - \mathbf{p}$ is perpendicular to $\mathbf{w}$? Notice that it's a line! Furthermore, we can check for perpendicularity using the dot product! Thus, a line can also be represented as

$$\{\mathbf{x} \mid (\mathbf{x} - \mathbf{p}) \cdot \mathbf{w} = 0\}.$$

Indeed, $\mathbf{p}$ satisfies this condition, and as well as those along the line through $\mathbf{p}$ that's perpendicular to $\mathbf{w}$.

Yet another representation is simply

$$\{\mathbf{x} \mid \mathbf{x} \cdot \mathbf{w} = c\}.$$

This is actually a variation of the above if you notice that

$$(\mathbf{x} - \mathbf{p}) \cdot \mathbf{w} = 0 \implies \mathbf{x} \cdot \mathbf{w} = \mathbf{p} \cdot \mathbf{w},$$

and the right-hand side is a constant. Also, if you expand out the dot product, you'll see that this is related to the equation $ax + by = c$.

The thing about these last two definitions is that they're only lines *in 2D*. If you try to do the same thing for 3D, you get a plane instead. (And for higher dimensions, you get a hyperplane.) Whereas with Equation 4, you always get a line regardless of dimension.

These different representations have advantages and disadvantages. You use whichever one's most helpful for your current task.

- The form $\mathbf{p} + \mathbf{v}t$ is useful when you need a concrete representation of all points on the line, because $\mathbf{p} + \mathbf{v}t$ is essentially a bijection between the real number *line* ($t \in \mathbb{R}$) and your line! It can't get any more concrete than that.

  However, it's harder to check if a given point $\mathbf{q}$ is in your line. You need to find a t such that $\mathbf{p} + \mathbf{v}t = \mathbf{q}$. It's doable, but you have to perform some algebraic manipulations first.

- The form $\mathbf{x} \cdot \mathbf{w} = c$ is useful when you want to easily check whether a point $\mathbf{q}$ is on the line: you simply substitute $\mathbf{q}$ and check!

  However, it's harder to get hold of the points—can you even give me one point on the line right now?

Exercise 3.1. Devise an algorithm to check whether a point $\mathbf{q}$ is of the form $\mathbf{p} + \mathbf{v}t$, and also to find the corresponding t.

Hint: Start with the equation $\mathbf{p} + \mathbf{v}t = \mathbf{q}$ and use some standard vector algebraic tools.

Hint: Cross-product both sides by $\mathbf{v}$, and also dot-product both sides by $\mathbf{v}$.

Exercise 3.2. Devise an algorithm to find a single point $\mathbf{x}$ that satisfies $\mathbf{x} \cdot \mathbf{w} = c$.

Hint: Intersect it with a random line! Most lines are probably not parallel to it.

Hint: Let $\mathbf{x} = \mathbf{v}t$ (i.e., a line through the origin) then solve for t. We already know how to intersect two lines!

Hint: You need to make sure you don't divide by zero. Can you give a vector that's surely not parallel to $\mathbf{w}$?

Of course, it's useful if we're able to convert between these representations. We have three representations so far:

$$\mathbf{p} + \mathbf{v}t \tag{5}$$
$$(\mathbf{x} - \mathbf{p}) \cdot \mathbf{w} = 0 \tag{6}$$
$$\mathbf{x} \cdot \mathbf{w} = c. \tag{7}$$

We've already seen how to convert from (6) to (7). So we only need to consider two more conversions: (7) to (5) and (5) to (6).

Exercise 3.3. Show how to convert from (7) to (5).

Hint: Use Exercise 3.2.

Hint: What's the direction of the line? (Recall the meaning of the related form 6.)

Exercise 3.4. Show how to convert from (5) to (6).

Hint: You really only need to know what $\mathbf{w}$ should be.

Hint: Use insights from the previous exercise, Exercise 3.3.

Actually, there's a fourth representation: just a pair of points $(\mathbf{p}, \mathbf{q})$ representing two points in a line! But this is easy:

Exercise 3.5. Explain how to convert between the two-point representation and (5).

3.1.1 Line segments

The above are all representations of lines. What about line segments? The simplest is to just specify its two endpoints, say $(\mathbf{p}, \mathbf{q})$. You can also use Equation 4 by limiting the range of t, like so:

$$\{\mathbf{p} + \mathbf{v}t \mid 0 \le t \le 1\}. \tag{8}$$

Note that $t = 0$ and $t = 1$ represents the endpoints.

Finally, you can easily represent rays as well.

Exercise 3.6. What are some natural representations of rays?

3.2 Intersection of two lines

We've already seen how to do this above. But now, we'll show a slightly different way.

For this, we will need our two lines to be represented in two different ways:

$$\{\mathbf{p} + \mathbf{v}t \mid t \in \mathbb{R}\}.$$
$$\{\mathbf{x} \mid \mathbf{x} \cdot \mathbf{w} = c\}.$$

The intersection point is now clearly a point $\mathbf{x}$ of the form $\mathbf{p}+\mathbf{v}t$ that also satisfies the equation $\mathbf{x} \cdot \mathbf{w} = c$. But now, we can just substitute!

$$\begin{aligned} \mathbf{x} \cdot \mathbf{w} &= c \\ (\mathbf{p} + \mathbf{v}t) \cdot \mathbf{w} &= c \\ \mathbf{p} \cdot \mathbf{w} + t\mathbf{v} \cdot \mathbf{w} &= c \\ t &= \frac{c - \mathbf{p} \cdot \mathbf{w}}{\mathbf{v} \cdot \mathbf{w}}. \end{aligned}$$

Now that we have a t, we can substitute it to $\mathbf{p}+\mathbf{v}t$ to get the point of intersection $\mathbf{x}$. You can then verify afterward that it satisfies $\mathbf{x} \cdot \mathbf{w} = c$!

Exercise 3.7. Verify that our solution $\mathbf{x}$ satisfies $\mathbf{x} \cdot \mathbf{w} = c$.

Important note: The intersection point may not have integer coordinates, even if the input points are lattice points! For example, the line through $(0, 0)$ and $(1, 1)$ and the line through $(0, 1)$ and $(1, 0)$ intersect at $(1/2, 1/2)$ which is not a lattice point. But it's at least guaranteed to have *rational* coordinates, so you may still be able to represent them exactly. (Although you need to check for overflow if the coordinates are large-enough integers.)

3.3 Intersection of two line segments

Let's now move to segment-segment intersection.

Problem 3.1 (Segment-segment intersection)**.** Given four points A, B, C and D, find the intersection of *line segments* AB and CD.

Actually, we can just solve this by intersecting the *lines* containing the segments, and then checking if the point is in both segments. For this to work, we need the following subroutine:

Exercise 3.8 (Point in segment)**.** Find an algorithm to check whether a point P is inside the line segment AB.

Hint: There are two parts: checking if collinear, and checking if in the bounding box.

There are at least three versions of this subroutine:

1. The case where all points have integer coordinates.

2. The case where A and B have integer coordinates and P has rational coordinates.

3. The case where all points have floating-point coordinates.

The second and third ones are the most relevant to segment-segment intersection, but the first one may be useful as well. For the third version, you'll probably have to do ε hacks, so be very careful when doing it! Avoid it if possible.

Another way is noticing the following:

Exercise 3.9. Recall that the *line* **ab** can be represented as

$$\{\mathbf{a} + \mathbf{v}t \mid t \in \mathbb{R}\}$$

with $\mathbf{v} = \mathbf{b} - \mathbf{a}$.

Show that $\mathbf{p} = \mathbf{a} + \mathbf{v}t$ is in the *line segment* **ab** iff $0 \leq t \leq 1$.

Using this, we may represent both line segments in $\mathbf{p} + \mathbf{v}t$ form and equate them, e.g.,

$$\mathbf{p} + \mathbf{v}t = \mathbf{q} + \mathbf{w}t'.$$

Solving for t and t' amounts to solving a system of two linear equations, and we can simply check that $0 \leq t \leq 1$ and $0 \leq t' \leq 1$ afterward.

However, all the above have a sort-of downside: at some point we will have to deal with rational numbers (because the intersection point may have rational coordinates), which can either make calculations tedious, or cause overflow (or even summon floating-point demons). So it's still worthwhile to consider the *decision version*, where we don't have to find the intersection point but we just want to know if it exists:

Problem 3.2 (Segment-segment intersection)**.** Given four points A, B, C and D, determine whether or not the *line segments* AB and CD intersect. (You don't need to find the intersection.)

This can be solved without finding the intersection! All we need to do is check whether the points are in the right location relative to the segments:

- Check that C and D lie in "opposite sides" of the line AB.
- Check that A and B lie in "opposite sides" of the line CD.

But this is solvable using one of our primitives: *counterclockwise check*! (Exercise 2.7) So we only need a couple of counterclockwise checks to solve the problem.

Even better, we can simplify things a bit:

Exercise 3.10. Show how to extend the solution to Exercise 2.7 to handle a slightly more general version: Given points A, B and C, return a number v such that:

- $v > 0$ if is C to counterclockwise of B when viewed from the vantage point of A;
- $v < 0$ if is C to clockwise of B when viewed from the vantage point of A;
- $v = 0$ if C is collinear with AB.

Hint: Use the cross product.

Exercise 3.11. Show that C and D lie in opposite sides of the line AB iff $(C - A) \times (B - A)$ and $(D - A) \times (B - A)$ have different signs.

So we can solve Problem 3.2 using four cross products (and comparing signs)! Just be careful when one or both of these cross products are zero; think about what the output should be!

However, remember that I said computational geometry is full of edge cases? Well, here's one:

Exercise 3.12. All the above solutions don't handle these edge cases:

- The case where the line segments are collinear.
- The case where either line segment is degenerate ($A = B$ or $C = D$), or both.

Explain how to fix the solutions to handle these cases.

Luckily, the solution works in the strict case:

Exercise 3.13. Show that the solution basically works for the *strict* version of Problem 3.2: Determine whether or not the *line segments* AB and CD intersect *strictly in their interior* (not endpoints).

Exercise 3.14. Explain how to find the intersection of two rays.

4 Polygons

This section is not completely written yet, sorry! Here are some points for now:

- You can represent a polygon simply as a sequence of points, either clockwise or counterclockwise.
- You can compute the *signed area* using a bunch of cross products, or equivalently, the *shoelace formula*.[11] Mathologer has a good video. Please watch it. Take the absolute value to get the actual (unsigned) area.
- If the coordinates are lattice points, then the area is either an integer of half an integer.
- The sign of the signed area determines whether the vertices were enumerated clockwise or counterclockwise.
- Some polygons are not *simple*! There are several kinds of degeneracies:
  - The polygon may have straight angles (180°).
  - The polygon may have zero-length edges!
  - The polygon may have zero area.
  - The polygon may have self-intersections.
- Some of these degeneracies can be handled by "preprocessing" your vertex list in some way. Some cannot; they're "inherent" in some way. Some are even hard to compute efficiently, such as determining if there are self-intersections.
- An important primitive operation is **point in polygon**: Given a point and a polygon, determine if a point is in a polygon.
- We've seen one solution to this, or at least to the special case *point in triangle*: Compute the areas of PAB, PBC and PCA and check that the sum is ABC. However, we've seen that this generally fails for degenerate polygons!
- Generally, we first check whether P is in the boundary by using point-in-segment (Exercise 3.8) for each edge of the polygon, so for the rest of the algorithm, we only need to check if P is in the interior.
- There are two common approaches to solve point-in-polygon:
  - *Crossing number* method: Draw a horizontal ray from the point P. The number of crossings determines whether or not the point is inside or not.
  - *Winding number* method: Compute the total "amount" that you revolve around the point as you walk along the polygon's boundary. If it's nonzero, then the point is inside.
- Almost always, winding number is better. It's easier to implement and works in some cases where there are self-intersections. Crossing number has a ton of edge cases and only really works for simple polygons.

[11] This is basically a discrete version of (a special case of) Green's theorem.

- If the coordinates are lattice points, then there's a neat formula called **Pick's theorem** relating the area A, the number of lattice points in the interior I, and the number of lattice points in the boundary B:

$$A = I + \frac{B}{2} - 1.$$

This theorem has lots of proofs! Wikipedia has a natural one, but you can find other interesting ones online.

5 Basic Geometry Algorithms

This section is not completely written yet, sorry! Most sections will be brief.

If you want me to write it in more detail, let me know so I can prioritize it!

5.1 Polar sorting

In many geometry tasks, you'll find yourself needing to sort points by angle. But it's not a good idea to use inverse trigonometric functions to get actual angles, because this summons the evil floating-point numbers! Sometimes, two points are very nearly collinear that you won't be able to distinguish them with floats.

The solution is to be able to sort them without actually computing angles. For this, we need to compare two points and determine which one is "more clockwise" than the other, relative to the positive x axis.

But this is precisely what the cross product does! So our custom sorting comparator more-or-less just checks the sign of the cross product. If the coordinates are integers, then we stay at integers, which is nice! (You may discriminate between two points with the same angle via, say, their lengths, or squared lengths if you want to stay in integers.)

However, you need to take care of the fact that sorting by cross product isn't really *transitive*. For example, $\langle 1,1\rangle < \langle -1,1\rangle < \langle -1,-1\rangle < \langle 1,-1\rangle < \langle 1,1\rangle$. For this, you may want to sort by *quadrant* first before sorting by angle. I usually use the following "quadrant" function:

```
int quadrant(const Point& p) {
    if (p.x == 0 && p.y == 0) return 0;
    if (p.x > 0 && p.y >= 0) return 1;
    if (p.x <= 0 && p.y > 0) return 2;
    if (p.x < 0 && p.y <= 0) return 3;
    if (p.x >= 0 && p.y < 0) return 4;
}
```

Note that the origin is a special case! The above assigns quadrants this way:

```
2 2 2 2 1 1 1
2 2 2 2 1 1 1
2 2 2 2 1 1 1
3 3 3 0 1 1 1
3 3 3 4 4 4 4
3 3 3 4 4 4 4
3 3 3 4 4 4 4
```

If you find yourself being confused by the inequalities, I suggest just doing something like this instead:

```
int quadrant(const Point& p) {
    if (p.y > 0) {
        return p.x < 0 ? 4 : p.x == 0 ? 3 : 2;
    } else if (p.y == 0) {
        return p.x < 0 ? 5 : p.x == 0 ? 0 : 1;
    } else { // p.y < 0
```

```
7           return p.x < 0 ? 6 : p.x == 0 ? 7 : 8;
8       }
9   }
```

This isn't really a "quadrant" function anymore, but it serves the same purpose in sorting so it's okay. The difference is that this assigns a unique ID for the axes, so you don't have to worry about the inequality symbols too much. It now looks like this

```
4 4 4 3 2 2 2
4 4 4 3 2 2 2
4 4 4 3 2 2 2
5 5 5 0 1 1 1
6 6 6 7 8 8 8
6 6 6 7 8 8 8
6 6 6 7 8 8 8
```

I structured the code so that this diagram is reflected somewhat accurately!

Exercise 5.1. Show that if the points are strictly above the y-axis, then you don't have to sort by quadrant first; the cross-product comparison will sort things correctly.

5.2 Convex hull

The **convex hull** of a set of points is the smallest convex polygon containing the points. A polygon is **convex** if all angles are $< 180°$.

You can imagine the convex hull as a rubber band around the points.

The problem of computing the convex hull of a given set of points is a classic one. There are two basic algorithms: Graham scan and Jarvis march (a.k.a. gift-wrapping). Please read the Wikipedia pages for now. Here are brief overviews:

- **Graham scan** sorts points in counterclockwise order, then iterates through them: using a stack, it *pops* points that form an angle $\geq 180°$.
- **Jarvis march** is kinda like selection sort: it tries to get the next point in the hull by finding the most clockwise point available.

In both cases, you use cross products to compare orientations.

Graham scan runs in $\mathcal{O}(n \log n)$ and Jarvis march runs in $\mathcal{O}(nh)$, where h is the number of vertices in the convex hull. (In the worst case, $h = \Theta(n)$, so this is $\mathcal{O}(n^2)$.)

Other popular algorithms are:

- **Andrew's monotone chain algorithm.** This is like Graham scan but instead of polar sorting, you just sort by x coordinate, then compute the upper and lower hull separately with a stack. This is an example of a *sweep line algorithm*.[12]
- **Quickhull.** If Jarvis march is selection sort, Quickhull is quicksort: It chooses an extreme point as a "pivot", splits the points, and recursively computes the hull at both sides. (However, unlike quicksort, the usual implementation isn't randomized, so if a case triggers the worst case, it will always do so. In other words, this is hackable.)

[12] Well, technically it's a sweep-line algorithm only if you process the upper and lower hulls simultaneously, and with a single pass, say from left to right.

As you can see, there seem to be some parallels between *convex hull* algorithms and *sorting* algorithms. This is more than just a coincidence: you can reduce sorting to convex hull!

Exercise 5.2. Reduce the problem of sorting the numbers $[x_1, x_2, \ldots, x_n]$ to the problem of finding the convex hull of $(x_1, x_1^2), (x_2, x_2^2), (x_3, x_3^2), \ldots, (x_n, x_n^2)$.

This means that convex hull algorithms cannot run faster than sorting! Since we know that sorting runs in $\Omega(n \log n)$, we know that convex hull algorithms run in $\Omega(n \log n)$ as well![13]

By the way, note that in Graham scan, sorting by angle is done by choosing a pivot P, typically the lowermost-leftmost point. Then polar sorting is done relative to P. Since all points are above P (or maybe to the right), Exercise 5.1 mostly applies, so we don't have to use quadrants to sort. However, to compare two points A and B, you have to compute the cross product of $A - P$ and $B - P$.

Alternatively, subtract P from all points first before sorting. (This is the "transform-untransform" thing again!)

5.3 Farthest pair of points

A standard problem goes as follows:

Problem 5.1. Given a set of n points, find the pair of points with the largest distance between them.

Actually, it should be doable now with the tools that we have so far!

Exercise 5.3. Show that the farthest pair of points are both in the convex hull.

Thus, we may first compute the convex hull, and then find the farthest pair of points there.

At this point, we can do an $\mathcal{O}(n^2)$ algorithm; for each point, find the point in the hull farthest from it in $\mathcal{O}(n)$ time. But this is no better than brute force in the worst case!

At this point, we can proceed in two ways, using two additional insights. We now label the points in the hull $P_0, \ldots, P_{k-1}$ counterclockwise. (Note that $k = \Theta(n)$ in the worst case.) We also let $P_i = (x_i, y_i)$.

5.3.1 Divide and conquer

One thing you might guess is that for a fixed point P, as you go around the hull, the points get farther and farther, until you reach a "peak", then the points get closer and closer after that. Unfortunately, this is wrong.

Exercise 5.4. Show that this is wrong by finding a counterexample. Furthermore, show that the sequence of distances from P as you go around the hull may zig-zag an arbitrary number of times.

[13] Actually, there's a bit of technical subtlety here. The proof of the $\Omega(n \log n)$ lower bound that you've probably seen uses decision trees, but that's not appropriate when dealing with convex hulls! See the Wikipedia page for more discussion on this.

Given this, it may be surprising that the following is true:

Theorem 5.5. Let f_i be the index of the farthest point from P_i, breaking ties by choosing the "closest point to P_i counterclockwise." Formally, let f_i be the smallest index $\geq i$ such that $|P_i - P_{f_i}|$ is maximum. In case of ties, choose f_i to be the smallest one. Also, we interpret indices to be mod k, so $P_j = P_{j+k} = P_{j+2k} = \ldots$. Then f_i is nondecreasing, i.e.,

$$f_0 \leq f_1 \leq f_2 \leq \ldots.$$

We'll prove this in a bit, but using this, we can now devise a *divide and conquer* algorithm to find the farthest pair of points in the hull.

Exercise 5.6. Compute $f_0, f_1, \ldots, f_{k-1}$ in $\mathcal{O}(k \log k)$ time.

Hint: First, compute f_0 in $\mathcal{O}(k)$. Then $f_k = f_0 + k$.

Hint: Recall the "divide and conquer optimization" in the DP4 module.

Now, we prove Theorem 5.5.

Proof. Let's assume $k \geq 2$, otherwise the hull is trivial. With $k \geq 2$, note that $i < f_i < i + k$ for any i. (Why?)

Suppose otherwise, so $f_i > f_{i+1}$ for some i. WLOG $i = 0$ (by just renumbering the points), so we have $f_0 > f_1$. So we have the following inequalities:

$$0 < 1 < f_1 < f_0$$

and also $f_0 < k$, so these are four distinct indices around the hull. Furthermore, the quadrilateral $P_0 P_1 P_{f_1} P_{f_0}$ must be convex also.

By definition, we know that P_1 is farther from P_{f_1} than P_{f_0} (or they have the same distance), while P_0 is *strictly* closer to P_{f_1} than P_{f_0}.

To make things easier to visualize—and in fact, I suggest drawing stuff on paper to make the following easier to follow—let's perform translation, rotation and scaling so that P_{f_1} ends up at $(1, 0)$ and P_{f_0} ends up at $(-1, 0)$. (Why can we do this?)

Note that the set of points strictly closer to $(1, 0)$ than $(-1, 0)$ are precisely the points (x, y) where $x > 0$. Similarly, the points farther from $(1, 0)$ than $(-1, 0)$ are the points (x, y) where $x < 0$. Finally, the points of equal distance from them are those with $x = 0$.

Therefore, if $P_i = (x_i, y_i)$, then we have $x_0 > 0$, $x_1 \leq 0$, $x_{f_1} = 1 > 0$ and $x_{f_0} = -1 < 0$. But this means that the sequence of four x-coordinates alternate in sign, which is impossible in a convex polygon, where the x-coordinates only switch signs at most twice! (Any convex polygon intersects any line in at most two points.) □

The running time is $\mathcal{O}(n \log n)$, both for computing the convex hull and this divide and conquer algorithm.

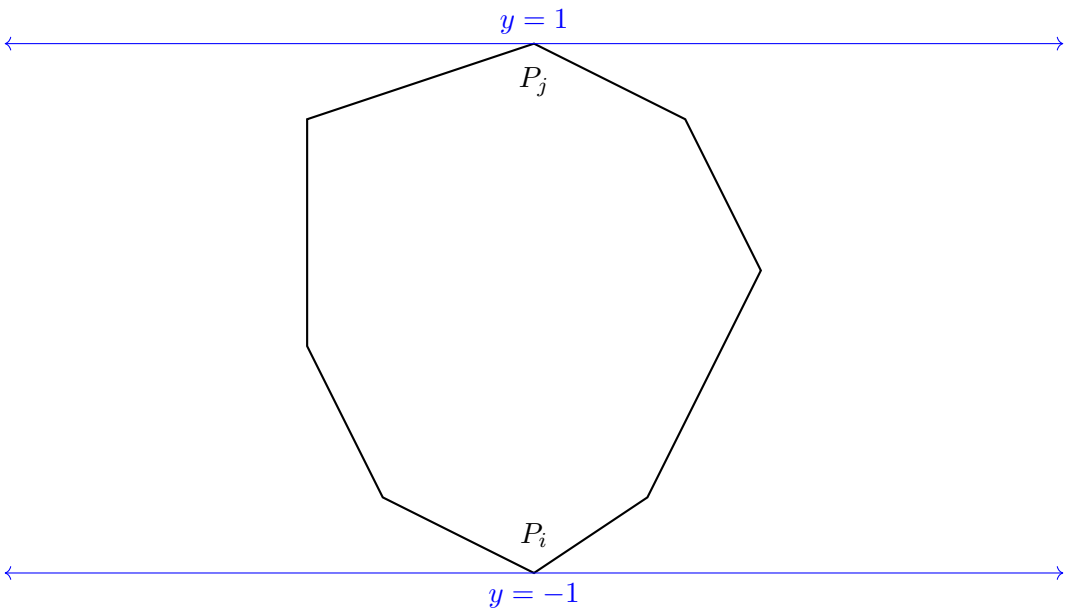
5.3.2 Rotating calipers (a.k.a., two pointers)

We can use another property of the farthest pair of points. Suppose (P_i, P_j) is the farthest pair of points. Again, to make things easier to visualize, let's perform linear transformations that preserve relative distances so that P_i ends up at $(0, -1)$ and P_j ends up at $(0, 1)$.

Now, draw horizontal lines through $(0, -1)$ and $(0, 1)$. (These are the lines $y = -1$ and $y = 1$.) Then I claim that:

Exercise 5.7. Show that the whole polygon is strictly between these two lines (except for P_i and P_j).

So our situation is now more or less like this:



In particular, this implies that as you go from i to j, the y-coordinates of the points are increasing, while as you go further from j to i, the y-coordinates decrease. In other words, the y-coordinates are bitonic.

I hope you can see now how this may allow us to possibly identify the farthest pair of points.

First, let's rotate the whole system continuously clockwise until one of the sides become horizontal. WLOG, assume that P_iP_{i+1} becomes horizontal (we can transform otherwise). Then, let's scale and translate again so that P_i and P_j end up at the lines $y = -1$ and $y = 1$ again, respectively. (Note that the x-coordinates may not be equal to 0 now because of the rotation. Also, note that P_iP_{i+1} is still horizontal.) I hope you can still see that:

Exercise 5.8. Show that after doing the above transformation, P_j is still the point with the highest y-coordinate.

Let's take a look at this in the original setup, without doing the linear transformations we did. The idea of "highest y-coordinate" is axis-dependent, and depended on the fact that P_iP_{i+1} became horizontal. We can make this axis-independent as follows:

Exercise 5.9. Let (P_i, P_j) be the farthest pair of points. Show that one of the following is true:

- P_j is the farthest point from the *line* P_iP_{i+1}.
- P_i is the farthest point from the *line* P_jP_{j+1}.

But now, finding the farthest point in a convex polygon from a line is straightforward: the distances from the line are *bitonic*! We can either use ternary search—$\mathcal{O}(k \log k)$—or binary search the sequence of distance differences—still $\mathcal{O}(k \log k)$—or use two pointers using the following:

Exercise 5.10. Let g_i be the smallest index $> i$ such that the distance from the line P_iP_{i+1} to P_{g_i} is maximum. Show that g_i is nondecreasing, i.e.,

$$g_0 \le g_1 \le g_2 \le \cdots.$$

Exercise 5.11. Use Exercise 5.10 to find the farthest pair of points in a convex polygon in $\mathcal{O}(k)$ time.

Hint: Use a "two-pointer" approach.

All in all, this is $\mathcal{O}(n \log n)$ time, dominated by computing the convex hull. (With Exercise 5.11, the rest of the steps take $\mathcal{O}(k) = \mathcal{O}(n)$ time.)

The algorithm described in Exercise 5.11 is called **rotating calipers**. It's just a fancy name for the two-pointer technique in geometry, especially if you're "rotating" in some way!

5.4 Closest pair of points

Another standard problem goes as follows:

Problem 5.2. Given a set of n points, find the pair of points with the smallest distance between them.

There's a standard divide and conquer algorithm for this! It runs in $\mathcal{O}(n \log n)$ time. Please read these notes[14] for now.

This may seem like a solution for a very specific problem, but it's actually more of a "paradigm". The key takeaway is that there are only "a few points nearby" every point. This is a useful insight if you're dealing with nearby points in the problem somehow! It usually allows you to reduce the number of pairs you need to consider from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n)$ or even $\mathcal{O}(n)$.

For example, a similar idea allows you to compute the "Manhattan spanning tree" of a set of points in $\mathcal{O}(n \log n)$ time. (The Euclidean spanning tree is a bit harder; it involves a somewhat advanced algorithm called Delaunay triangulation.) Also, since we're using Manhattan distance here, segment trees are enough to find the "nearby" points! You don't need Euclidean-distance-based structures (such as kd-trees).

The following exercises demonstrate some nontrivial observations related to bounded distances.

Exercise 5.12. Suppose there are n distinct points in an $r \times c$ rectangle, and the distance between every pair of distinct points is at least d. Show that $n = \mathcal{O}(rc/d^2 + 1)$.

Hint: Reduce to the special case $d = 1$.

Hint: Gridify the rectangle into $(d/2) \times (d/2)$ cells. How many points are in each cell?

Exercise 5.13. Suppose there are n points and there are m ordered pairs of such points with distance $\le d$. (Formally, there are m pairs (i, j), $1 \le i, j \le n$ such that $|P_i - P_j| \le d$.) Show that there are $\mathcal{O}(m)$ ordered pairs of points with distance $2d$.

[14] https://www.cs.cmu.edu/~15451-s20/lectures/lec23-closest-pair.pdf

Hint: Gridify into $d \times d$ cells. Can you bound the number of pairs of points that belong to the same cell?

Note: The "2" in "$2d$" is arbitrary. By repeating this argument, the result holds for any such fixed constant $c > 0$. (The constant behind "$\mathcal{O}(m)$" would depend on c.)

5.5 Line sweep

Line sweep is basically the technique of moving a line (typically a vertical line) across the input to solve the problem. You've probably seen an example of this in the context of data structures (with minimal geometry involved):

Problem 5.3. Given n points $P_1, P_2, \ldots, P_n$ and m points $Q_1, Q_2, \ldots, Q_m$, find the number of pairs (P_i, Q_j) such that P_i is southwest of Q_j.

On the surface, this seems like a problem that requires 2D segment trees: add the points P_i to a 2D segment tree then query the tree for each Q_j. This runs in $\mathcal{O}(n \log n + m \log^2 n)$ which is somewhat slow and hard to implement.

However, you don't need a 2D structure! We can use a *sweep line* instead, by sweeping a vertical line from left to right. (In effect, we're turning the x coordinate into a time coordinate.) Upon encountering a P_i point, we insert it into a 1D structure (we only need its y coordinate), and upon encountering a Q_i point, we query the structure. This runs in $\mathcal{O}(n \log n + m \log m)$ time but is much simpler to code!

It's not hard to imagine that sweep lines are useful in geometry in general. There's a sweep line algorithm to compute convex hulls, and to find the closest pair of points, and even to solve the following problem:

Problem 5.4 (Segment intersection)**.** Given n line segments, determine whether or not there's a pair of intersecting segments.

Sweep lines allow you to solve all the above problems in $\mathcal{O}(n \log n)$ time (or maybe $\mathcal{O}(n \log^k n)$ in general).

Exercise 5.14. Using standard data structure techniques plus sweep line, solve a special case of Problem 5.4 where each segment is either vertical or horizontal.

Note that line sweep doesn't have to be from left to right or top to bottom: you can have a *polar sweep line* too, where a line rotates about some point!

6 Miscellaneous

6.1 Why you can't add two points

Here, I'll try to address the following:

> *Why use vectors at all instead of just points? Aren't vectors and points the same thing—a pair of coordinates?*

It is true that both vectors and points are represented as a pair (x, y) of coordinates. However, that doesn't mean we should conflate the two! Conceptually, they represent different things, just like a pair (x, y) of coordinates is not considered the same as a random pair (i, j) of numbers. It's just an accident of computer representation. We're not computers! It's useful for us as humans to treat points and vectors as separate things. This will reduce the number of bugs in our code, because although the computer is perfectly happy to add two "points" $\mathbf{p}$ and $\mathbf{q}$, it's more likely a mistake, since it doesn't make sense to add two points! It's a good habit to keep in mind; it's not just the literal data types that matter, but also its "meaning".

Anyway, intuitively, we should think of points as "absolute locations" in space, while vectors as representing "relationships between points". With this intuition, we can somewhat infer which operations are valid and which are not:

- Adding two vectors $\mathbf{v}$ and $\mathbf{w}$ should make sense: It represents going along $\mathbf{v}$ then along $\mathbf{w}$. It should result in another *vector*.
- Adding a vector $\mathbf{v}$ to a point $\mathbf{p}$ also makes sense: It represents going along $\mathbf{v}$ starting from $\mathbf{p}$. It should result in a *point*.
- Adding two points $\mathbf{p}$ and $\mathbf{q}$ doesn't make sense! The points represent absolute locations in space; the point $(0, 0)$ is not particularly special here since it was just chosen arbitrarily. Since $\mathbf{p} + \mathbf{q}$ depends on this arbitrary meaningless choice, it's also meaningless.
- Similarly, negating a point $\mathbf{p}$ doesn't make sense; it's heavily reliant on the choice of origin $(0, 0)$.
- However, negating a vector $\mathbf{v}$ makes sense: it represents going along $\mathbf{v}$ but backwards. It results in a *vector*.
- Subtracting vector $\mathbf{v}$ from another vector $\mathbf{w}$ should make sense, since $-\mathbf{v}$ makes sense. It results in a *vector*.
- Subtracting vector $\mathbf{v}$ from a point $\mathbf{p}$ should make sense, since $-\mathbf{v}$ makes sense. It results in a *point*.
- Subtracting a point $\mathbf{p}$ from another point $\mathbf{q}$ should make sense! $\mathbf{q} - \mathbf{p}$ represents a relationship between the two points. It is a *vector*.
- Subtracting a point $\mathbf{p}$ from a vector $\mathbf{v}$ doesn't make sense! (Can you see why?)

You've probably seen instances of this elsewhere! For example, in physics, "potential energy" doesn't really make sense on its own; the thing we call "0 potential energy" was basically just arbitrarily chosen. It only makes sense once you *subtract* two of them; you get a quantity that's still in units of energy, but now has physical meaning![15]

[15] If you're curious, the mathematical object formalizing this kind of structure—where there are "absolute" objects and there are "relative" objects that look very much alike but are interpreted differently—is called a *torsor*. Think of "torsion"/"distortion"; I don't actually know if these are etymologically related to "torsor", but the imagery fits, e.g., you can think of vectors as sort of "distortions" of the plane (i.e., the points).

7 Problems

- **Polygon Inside A Circle:** UVa 10432
- **Freckles:** UVa 10034
- **Rope Crisis in Ropeland!:** UVa 10180
- **Useless Tile Packers:** UVa 10065
- **How Far?:** UVa 10466
- **Triangle Containment:** https://projecteuler.net/problem=102
- **Triangles Containing the Origin:** https://projecteuler.net/problem=184
- **Pythagorean Tree:** https://projecteuler.net/problem=395
- **Triangles Containing the Origin II:** https://projecteuler.net/problem=456
- **Armistice Area Apportionment:** https://codeforces.com/problemset/problem/645/G
- **Chef and Polygons:** https://www.codechef.com/problems/CHPLGNS
- **Defend the Recipe:** https://www.codechef.com/problems/ALLPOLY
- **Path by a Castle:** https://www.codechef.com/problems/KGP16B
- **A Simple Polygon:** https://www.codechef.com/problems/SIMPPOLY
- **Two Roads:** https://www.codechef.com/problems/TWOROADS
- **Hull Sum:** https://www.codechef.com/problems/HULLSUM
- **Chef and Balanced Polygons:** https://www.codechef.com/problems/BALANPOL
- **Basketball Game:** https://www.hackerrank.com/contests/noi-ph-2018-postselection/challenges/basketball-game
- **How Long Will it Rain?:** https://www.hackerrank.com/contests/noi-ph-2018-postselection/challenges/how-long-will-it-rain
- **Cubes and Cylinders:** https://www.hackerrank.com/contests/noi-ph-2018-postselection/challenges/cubes-and-cylinders

References

[1] Victor Lecomte. *Handbook of geometry for competitive programmers*. URL: https://victorlecomte.com/cp-geo.pdf.

[2] Grant Sanderson. *Essence of linear algebra*. 2016. URL: https://www.youtube.com/playlist?list=PL0-GT3co4r2y2YErbmuJw2L5tW4Ew2O5B.