2D Geometry

Topics

- Points and Vectors
- Transformations
- Products and angles
- Lines
- Segments
- Polygons
- Circles

Topics

- **Points and Vectors**
- Transformations
- Products and angles
- Lines
- Segments
- Polygons
- Circles

Points and Vectors

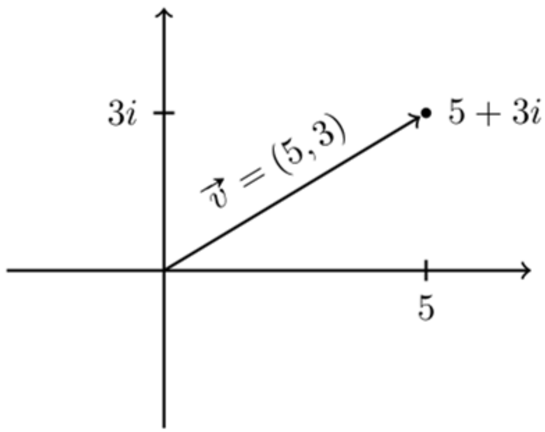
- **Complex numbers**
  - Basic operations
  - Polar form
  - Multiplication
- **Point representation**
  - With a custom structure
  - With the C++ complex structure

Points and Vectors

- **Complex numbers**
  - Basic operations
  - Polar form
  - Multiplication
- **Point representation**
  - With a custom structure
  - With the C++ complex structure

Complex numbers

- Complex numbers are an extension of the real numbers with a new unit, the imaginary unit, noted ***i***.
- A complex number is usually written as $a + bi$ (for $a, b \in R$) and we can interpret it geometrically as point *(a, b)* in the two-dimensional plane, or as a vector with components $v = (a, b)$.



Points and Vectors

- **Complex numbers**
  - **Basic operations**
  - Polar form
  - Multiplication
- **Point representation**
  - With a custom structure
  - With the C++ complex structure

Basic operations

$(a + bi) + (c + di) = (a + c) + (b + d)i$ (addition)

$(a + bi) - (c + di) = (a - c) + (b - d)i$ (subtraction)

$k(a + bi) = (ka) + (kb)i$
(multiplication by scalar)

Points and Vectors

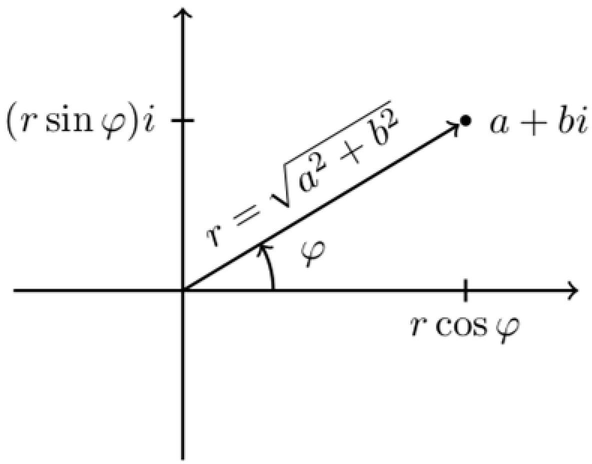
- **Complex numbers**
  - Basic operations
  - **Polar form**
  - Multiplication
- **Point representation**
  - With a custom structure
  - With the C++ complex structure

Polar form

For a given complex number $a + bi$, we can compute its polar form as

$r = |a + bi| = a^2 + b^2$

$\varphi = arg(a + bi) = atan2(b, a)$



$r \cos\varphi + (r \sin\varphi)i = r(\cos\varphi + i \sin\varphi) =: r\ cis\ \varphi$

Points and Vectors

- **Complex numbers**
  - Basic operations
  - Polar form
  - **Multiplication**
- **Point representation**
  - With a custom structure
  - With the C++ complex structure

Multiplication

Complex multiplication is easiest to understand using the polar form.

$(r_1 \text{ cis } \varphi_1) * (r_2 \text{ cis } \varphi_2) = (r_1 r_2) \text{ cis}(\varphi_1 + \varphi_2)$

$$
\begin{aligned}
(a + bi) * (c + di) &= ac + a(di) + (bi)c + (bi)(di) \\
&= ac + adi + bci + (bd)i\ 2 \\
&= ac + (ad + bc)i + (bd)(-1) \\
&= (ac - bd) + (ad + bc)i
\end{aligned}
$$

Points and Vectors

- **Complex numbers**
  - Basic operations
  - Polar form
  - Multiplication
- **Point representation**
  - With a custom structure
  - With the C++ complex structure

With a custom structure

```
typedef double T;
template <class Y> int sgn(Y x) {
return (Y(0) < x) - (x < Y(0));
}
struct pt{
    T x,y;
    pt operator+(pt p) {return {x+p.x, y+p.y};}
    pt operator-(pt p) {return {x-p.x, y-p.y};}
    pt operator*(T d) {return {x*d, y*d};}
    pt operator/(T d) {return {x/d, y/d};} // only for floating-
    bool operator==(pt a) {return a.x == x && a.y == y;}
    bool operator!=(pt a) {return a.x != x || a.y != y;}
    T sq(pt p) {return p.x*p.x + p.y*p.y;}
    double abs(pt p) {return sqrt(sq(p));}
};
```

With the C++ complex structure

```
typedef double T;
typedef complex<T> pt;
#define x real()
#define y imag()
int main(){
    pt p{3,-4}, q= {6,9};
    cout << p.x << " " << p.y << "\n"; // 3 -4
    // Can be printed out of the box
    cout << p << "\n"; // (3,-4)
    pt p2{-3,2};
    //p2.x = 1; // doesn't compile
    p2 = {1,2}; // correct
    cout<<p2<<"\n";
    pt a{3,1}, b{1,-2};
    a += 2.0*b; // a = (5,-3)
    cout<<a<<"\n";
    cout << a*b << " " << a/-b << "\n"; // (-1,-13) (-2.2,-1.4)
    pt p3{4,3};
    // Get the absolute value and argument of point (in [-pi,pi])
    cout << abs(p3) << " " << arg(p3) << "\n"; // 5 0.643501
    // Make a point from polar coordinates
    cout << polar(2.0, -M_PI/4.0) << "\n"; // (1.41421,-1.41421)
    cout<<M_PI<<"\n";
    cout<<norm(complex<double>(2.0,1.0))<<"\n";
    cout<<(norm(complex<double>(2.0,1.0)) == 5.0)<<"\n";
    cout<<(norm(complex<double>(2.0,1.0)) == 5)<<"\n";
    cout<<(5==5.0)<<"\n";
}
```

Topics

- Points and Vectors
- **Transformations**
- Products and angles
- Lines
- Segments
- Polygons
- Circles

Transformations

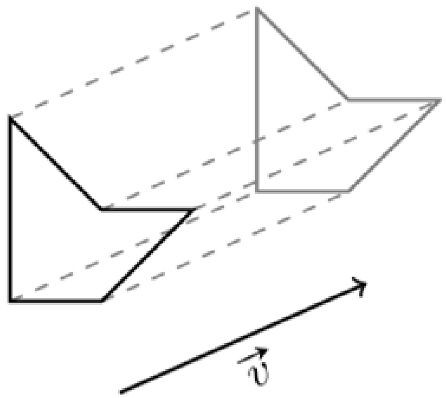
- ✔ Translation
- ✔ Scaling
- ✔ Rotation

Transformations

- ✔ **Translation**
- ✔ Scaling
- ✔ Rotation

Translation

To translate an object by a vector *v* , we simply need to add *v* to every point in the object. The corresponding function is $f(p) = p + v$ with $v \in C$.



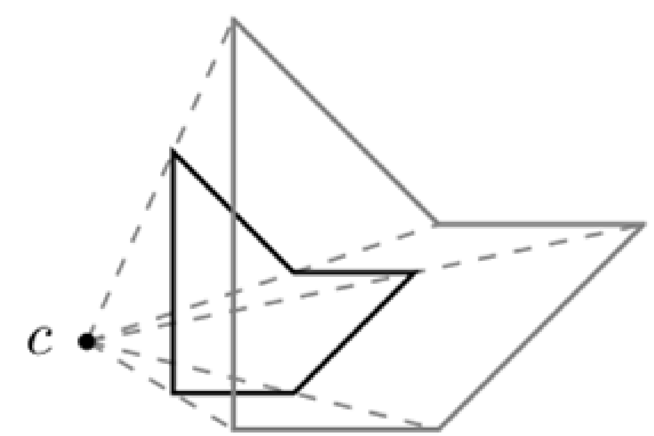

```
pt translate(pt v, pt p) {return p+v;}
```

Transformations

- ✔ Translation
- ✔ **Scaling**
- ✔ Rotation

Scaling

- To scale an object by a certain ratio *α* around a center *c*, we need to shorten or lengthen the vector from *c* to every point by a factor *α*, while conserving the direction.
- The corresponding function is $f(p) = c + \alpha(p - c)$ (α is a real here, so this is a scalar multiplication)




```
pt scale(pt c, double factor, pt p) {return c + (p-c)*factor;}
```

Transformations

- ✔ Translation
- ✔ Scaling
- ✔ **Rotation**

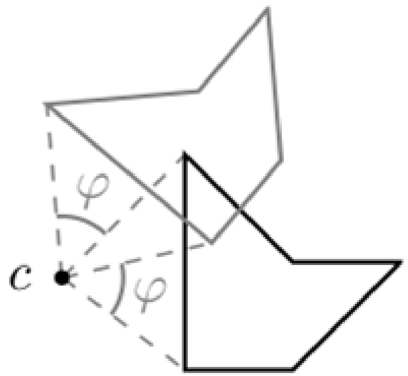
Rotation

To rotate an object by a certain angle φ around center *c*, we need to rotate the vector from *c* to every point by φ.

The corresponding function is $f(p) = c + cis\ \varphi * (p - c)$.

$(x + yi) * cis\ \varphi = (x + yi) * (\cos\varphi + i \sin\varphi)$

$= (x \cos\varphi - y \sin\varphi) + (x \sin\varphi + y \cos\varphi)i$




```
pt rot(pt p, double a) {
    return {p.x*cos(a) - p.y*sin(a), p.x*sin(a) + p.y*cos(a)};
}
pt rotation(pt p, double a) {return p * polar(1.0, a);}
```

Rotation

And among those, we will use the rotation by 90° quite often:

$$(x + yi) * cis(90^\circ) = (x + yi) * (cos(90^\circ) + i\, sin(90^\circ))$$

$$= (x + yi) * i = -y + xi$$

It works fine with integer coordinates, which is very useful:

```
pt perp(pt p) {return {-p.y, p.x};}
```

Topics

- Points and Vectors
- Transformations
- **Products and angles**
- Lines
- Segments
- Polygons
- Circles

Products and angles

- ✔ **dot product**
- ✔ cross product
- ✔ orientation

dot product

The dot product $v \cdot w$ of two vectors v and w can be seen as a measure of how similar their directions are. It is defined as

$$\vec{v} \cdot \vec{w} = ||\vec{v}|| ||\vec{w}|| \cos\theta$$

```
T dot(pt v, pt w) {return v.x*w.x + v.y*w.y;}

bool isPerp(pt v, pt w) {return dot(v,w) == 0;}

double angle(pt v, pt w) {
    double cosTheta = dot(v,w) / abs(v) / abs(w);
    return acos(max(-1.0, min(1.0, cosTheta)));
}
```

Products and angles

- ✔ dot product
- ✔ **cross product**
- ✔ orientation

Cross product

The cross product *v* × *w* of two vectors *v* and *w* can be seen as a measure of how perpendicular they are. It is defined in 2D as

$$\vec{v} \times \vec{w} = \|\vec{v}\| \|\vec{w}\| \sin\theta$$

```
T cross(pt v, pt w) {return v.x*w.y - v.y*w.x;}
```

Products and angles

- ✔ dot product
- ✔ cross product
- ✔ **orientation**

orientation

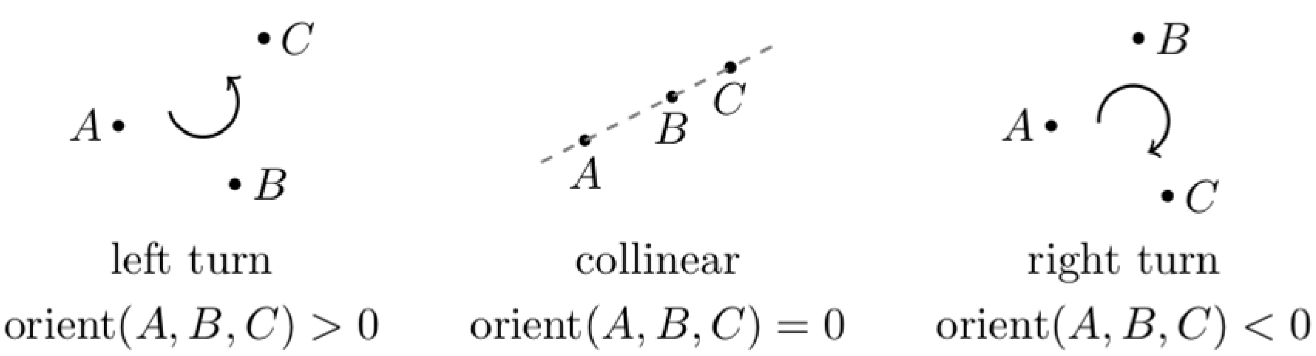
One of the main uses of cross product is in determining the relative position

of points and other objects.

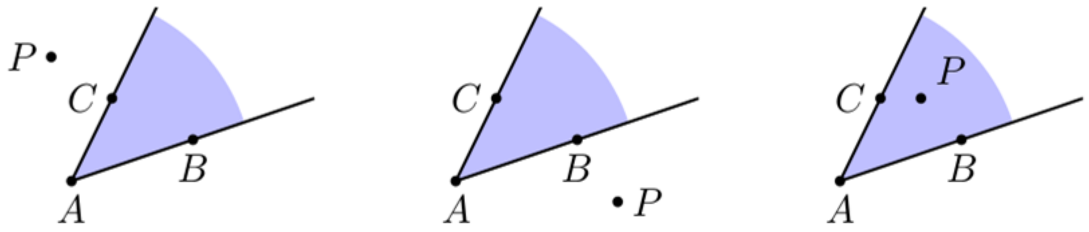
For this, we define the function orient(A,B,C)=AB×AC. It is positive if C is on the left side of AB, negative on the right side, and zero if C is on the line containing AB.

It is straightforward to implement:

```
T orient(pt a, pt b, pt c) {return cross(b-a,c-a);}
```



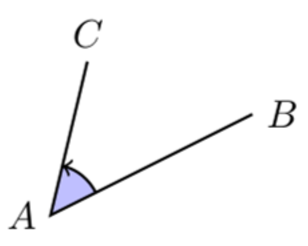
inAngle



$\text{orient}(A,B,P) \geq 0$	$\text{orient}(A,B,P) < 0$	$\text{orient}(A,B,P) \geq 0$
$\text{orient}(A,C,P) > 0$	$\text{orient}(A,C,P) \leq 0$	$\text{orient}(A,C,P) \leq 0$
KO	KO	OK

```
bool inAngle(pt a, pt b, pt c, pt p) {
    assert(orient(a,b,c) != 0);
    if (orient(a,b,c) < 0) swap(b,c);
    return orient(a,b,p) >= 0 && orient(a,c,p) <= 0;
}
```

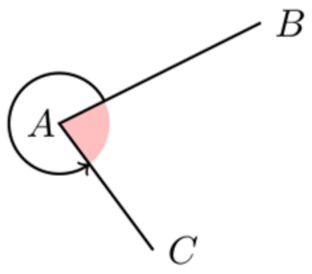
Oriented angle



$\mathrm{orient}(A, B, C) > 0$

`angle()` $= 50°$

`orientedAngle()` $= 50°$



$\mathrm{orient}(A, B, C) < 0$

`angle()` $= 80°$

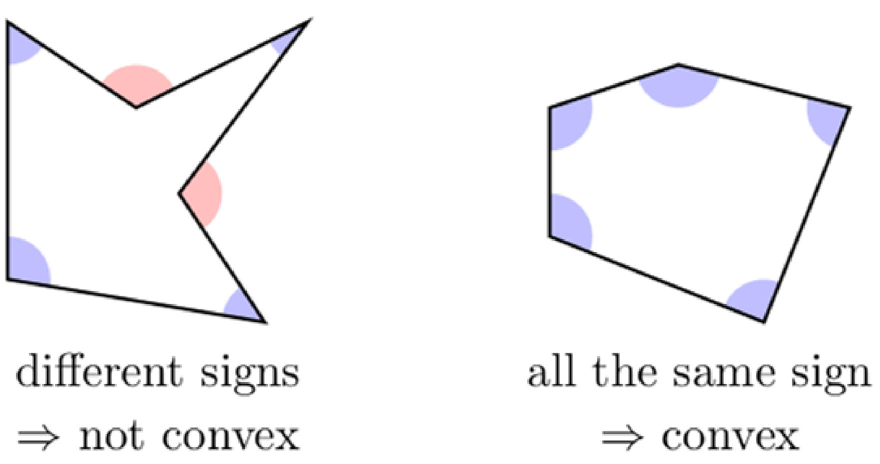
`orientedAngle()` $= 280°$

```
double orientedAngle(pt a, pt b, pt c) {
    if (orient(a,b,c) >= 0)
        return angle(b-a, c-a);
    else
        return 2*M_PI - angle(b-a, c-a);
}
```

Convex

Yet another use case is checking if a polygon $P_1 \cdots P_n$ is convex: we compute the n orientations of three consecutive vertices orient(P_i, P_{i+1}, P_{i+2}), wrapping around from n to 1 when necessary.
The polygon is convex if they are all ≥ 0 or all ≤ 0, depending on the order in which the vertice

```
bool isConvex(vector<pt> p) {
    bool hasPos=false, hasNeg=false;
    for (int i=0, n=p.size(); i<n; i++) {
        int o = orient(p[i], p[(i+1)%n], p[(i+2)%n]);
        if (o > 0) hasPos = true;
        if (o < 0) hasNeg = true;
    }
    return !(hasPos && hasNeg);
}
```



Topics

- Points and Vectors
- Transformations
- Products and angles
- **Lines**
- Segments
- Polygons
- Circles

Lines

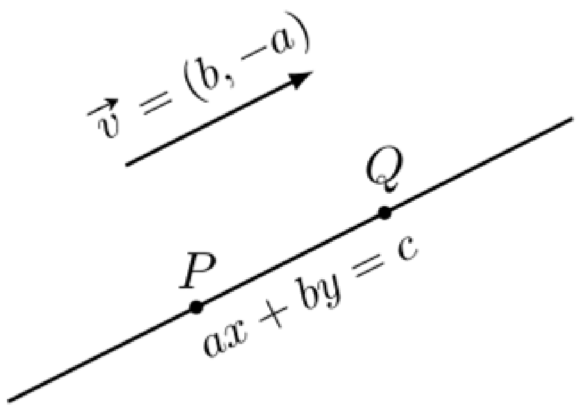
- ✔ Line representation
- ✔ Side and distance
- ✔ Perpendicular throught a point
- ✔ Translating a line
- ✔ Line intersection
- ✔ Orthogonal projection and reflection

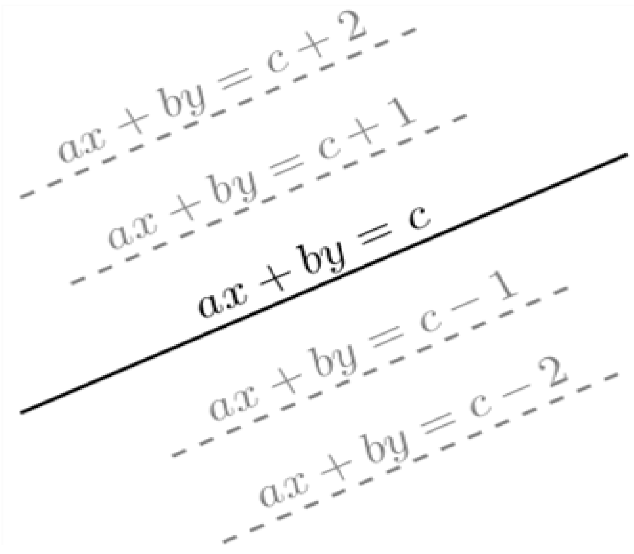
Lines

- ✔ **Line representation**
- ✔ Side and distance
- ✔ Perpendicular throught a point
- ✔ Translating a line
- ✔ Line intersection
- ✔ Orthogonal projection and reflection

Line representation

finding the equation of a line going through two points *P* and *Q* is easy: define the dire






```
struct line{
    pt v;
    T c;
    // From direction vector v and offset c
    line(pt v, T c) : v(v), c(c) {}
    // From equation ax+by=c
    line(T a, T b, T c) : v({b,-a}), c(c) {}
    // From points P and Q
    line(pt p, pt q) : v(q-p), c(cross(v,p)) {}
};
```

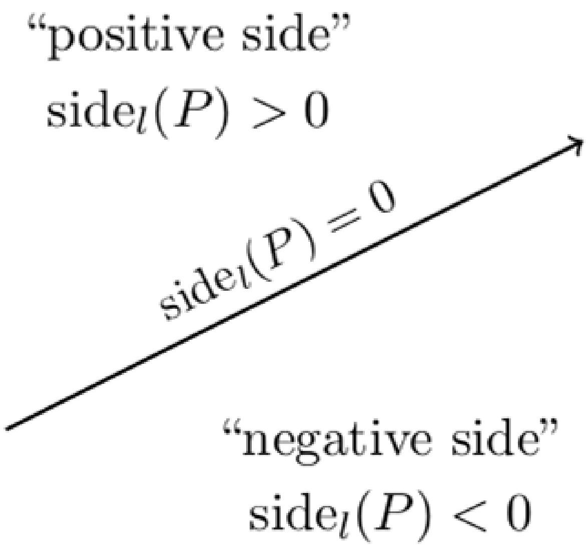
Lines

- ✔ Line representation
- ✔ **Side and distance**
- ✔ Perpendicular throught a point
- ✔ Translating a line
- ✔ Line intersection
- ✔ Orthogonal projection and reflection

Side and distance

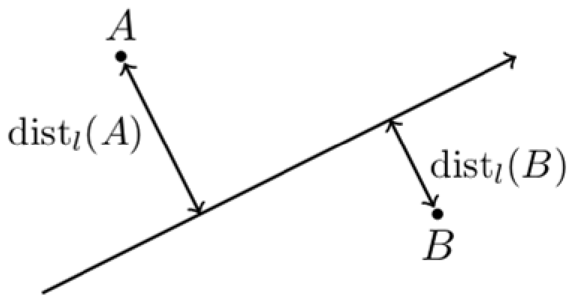
One interesting operation on lines is to find the value of *ax* + *by* − *c* for a given point *(x, y)*. For line *l* and point *P*=*(x, y)*, we will denote this operation as

$$\text{side}_l(P) := ax + by - c = \vec{v} \times P - c.$$




```
T side(pt p) {return cross(v,p)-c;}
```

Side and distance



This gives an easy implementation of distance:

```
double dist(pt p) {return abs(side(p)) / abs(v);}
double sqDist(pt p) {return side(p)*side(p) / (double)sq(v);}
```

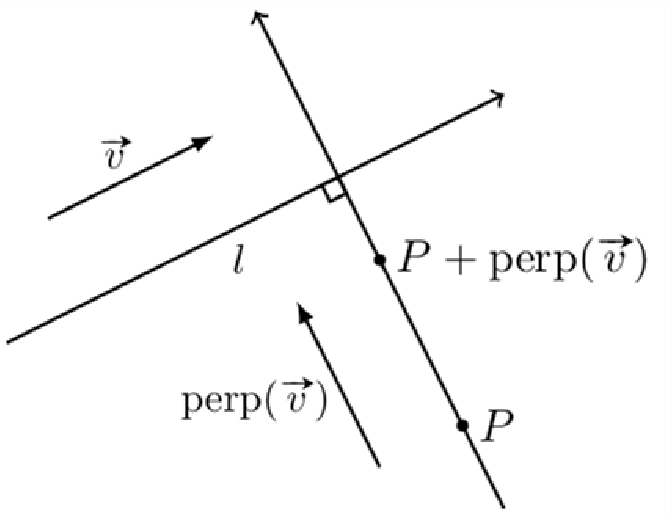
Lines

- ✔ Line representation
- ✔ Side and distance
- ✔ **Perpendicular throught a point**
- ✔ Translating a line
- ✔ Line intersection
- ✔ Orthogonal projection and reflection

Perpendicular throught the point

Two lines are perpendicular if and only if their direction vectors are perpendicular. Let's say we have a line l of direction vector *v* . To find a line perpendicular to line *l* and which goes through a certain point *P.*

it's simpler to just compute it as the line from *P* to *P* + perp(*v*).



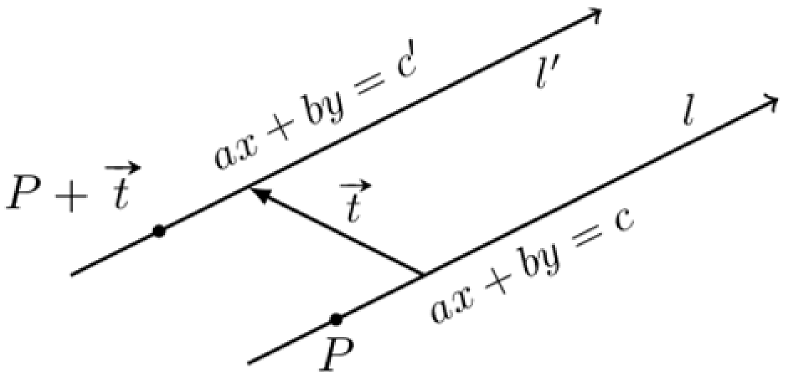

```
line perpThrough(pt p) {return {p, p + perp(v)};}
```

Lines

- ✔ Line representation
- ✔ Side and distance
- ✔ Perpendicular throught a point
- ✔ **Translating a line**
- ✔ Line intersection
- ✔ Orthogonal projection and reflection

Translating a line

If we want to translate a line *l* by vector *t* , the direction vector *v* remains the same but we have to adapt *c*.



which allows us to find c':

$c' = v \times (P + t) = v \times P + v \times t = c + v \times t$

```
line translate(pt t) {return {v, c + cross(v,t)};}
```

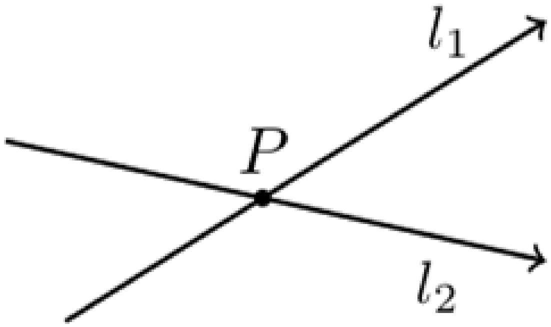
Lines

- ✔ Line representation
- ✔ Side and distance
- ✔ Perpendicular throught a point
- ✔ Translating a line
- ✔ **Line intersection**
- ✔ Orthogonal projection and reflection

Line intersection

There is a unique intersection point between two lines l_1 and l_2 if and only if $v_{l1} \times v_{l2}$!= 0.

$$P = \frac{c_{l_1}\overrightarrow{v_{l_2}} - c_{l_2}\overrightarrow{v_{l_1}}}{\overrightarrow{v_{l_1}} \times \overrightarrow{v_{l_2}}}$$




```
bool inter(line l1, line l2, pt &out) {
    T d = cross(l1.v, l2.v);
    if (d == 0) return false;
    out = (l2.v*l1.c - l1.v*l2.c) / d;
    return true;
}
```

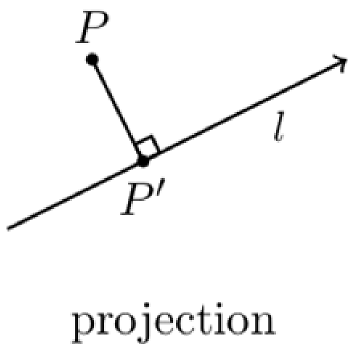
Lines

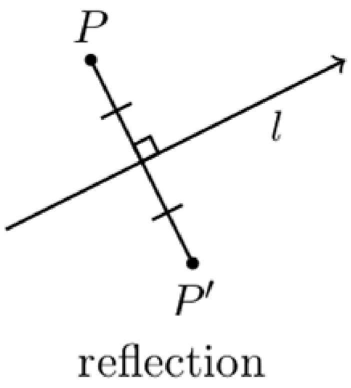
- ✔ Line representation
- ✔ Side and distance
- ✔ Perpendicular throught a point
- ✔ Translating a line
- ✔ Line intersection
- ✔ **Orthogonal projection and reflection**

Orthogonal projection and reflection

The orthogonal projection of a point *P* on a line *l* is the point on *l* that is closest to *P.*

The reflection of point *P* by line *l* is the point on the other side of *l* that is at the same distance and has the same orthogonal projection.






```
pt proj(pt p) {return p - perp(v)*side(p)/sq(v);}
pt refl(pt p) {return p - perp(v)*2*side(p)/sq(v);}
```

Topics

- Points and Vectors
- Transformations
- Products and angles
- Lines
- **Segments**
- Polygons
- Circles

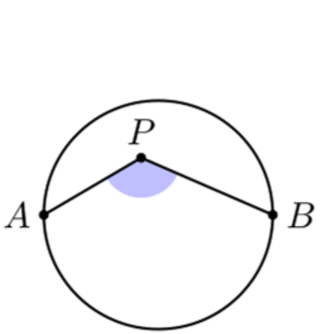
Segments

- ✔ Point on Segment
- ✔ Segment-point distance
- ✔ Segment-segment distance

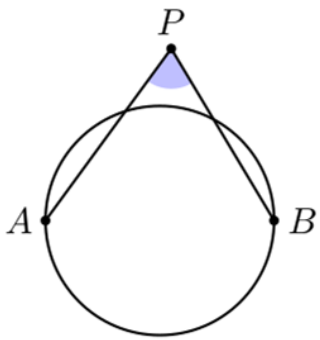
Segments

- ✔ **Point on Segment**
- ✔ Segment-point distance
- ✔ Segment-segment distance

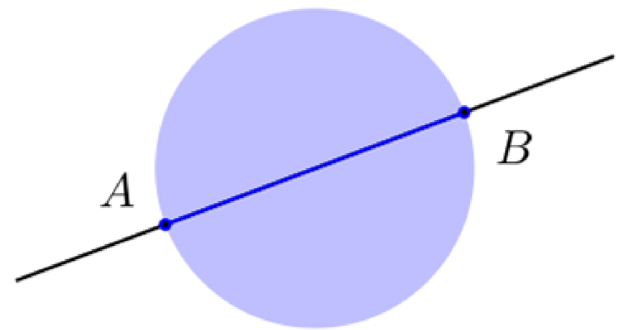
Point on segment



$\overrightarrow{PA} \cdot \overrightarrow{PB} \leq 0$
in disk



$\overrightarrow{PA} \cdot \overrightarrow{PB} > 0$
out of disk



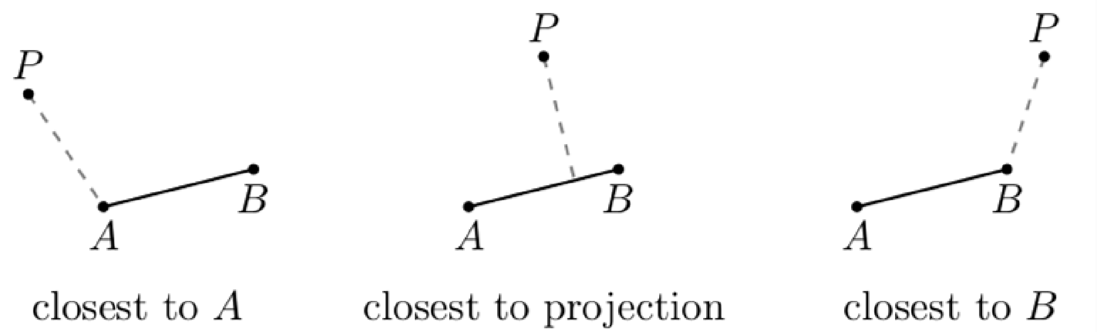
intersection of line and disk = segment

```
bool inDisk(pt a, pt b, pt p) {return dot(a-p, b-p) <= 0;}
bool onSegment(pt a, pt b, pt p) {
    return orient(a,b,p) == 0 && inDisk(a,b,p);
}
```

Segments

- ✔ Point on Segment
- ✔ **Segment-point distance**
- ✔ Segment-segment distance

Segment-point distance



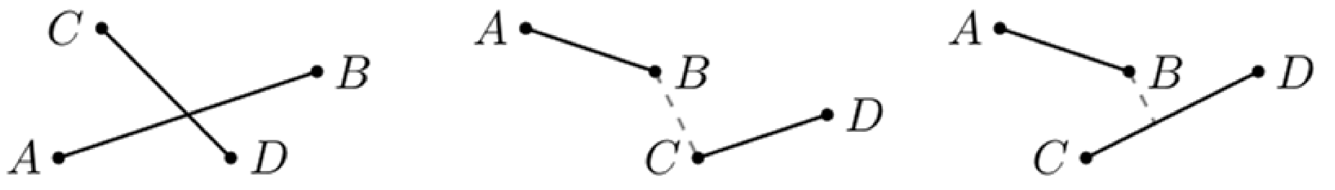
To check this, we can use the cmpProj() method in line .

```
bool cmpProj(pt p, pt q) {return dot(v,p) < dot(v,q);}
double segPoint(pt a, pt b, pt p) {
    if (a != b) {
        line l(a,b);
        if (l.cmpProj(a,p) && l.cmpProj(p,b)) // if closest to projection
        return l.dist(p); // output distance to line
    }
    return min(abs(p-a), abs(p-b)); // otherwise distance to A or B
}
```

Segments

- ✔ Point on Segment
- ✔ Segment-point distance
- ✔ **Segment-segment distance**

Segment-segment distance



proper intersection | between points | between point and line

```
double segSeg(pt a, pt b, pt c, pt d) {
    pt dummy;
    if (properInter(a,b,c,d,dummy))
        return 0;
    return min({segPoint(a,b,c), segPoint(a,b,d),
        segPoint(c,d,a), segPoint(c,d,b)});
}
```

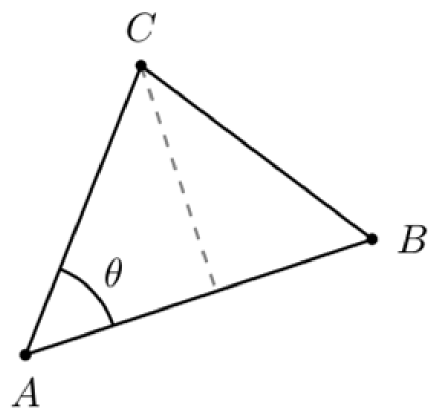
Topics

- Points and Vectors
- Transformations
- Products and angles
- Lines
- Segments
- **Polygons**
- Circles

Triangle area

To compute the area of a polygon, it is useful to first consider the area of a triangle ABC.

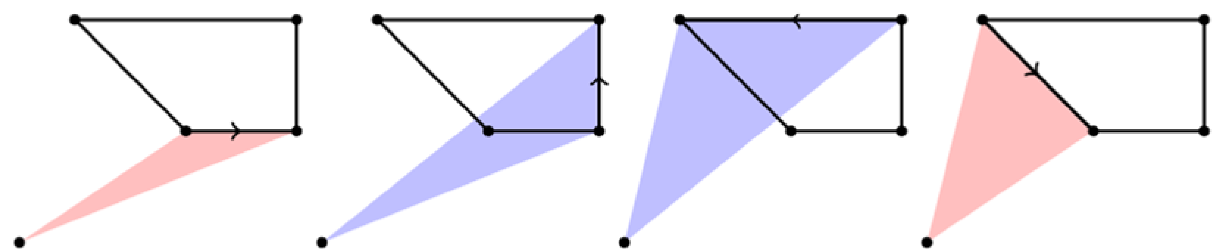
$(|AB||AC|\sin\theta)/2 = (|AB \times AC|)/2$




```
double areaTriangle(pt a, pt b, pt c) {return abs(cross(b-a, c-a)) / 2.0;}
```

Polygon area

Let’s take an arbitrary reference point *O*. Let’s consider the vertices of *ABCD* in order,



```
double areaPolygon(vector<pt> p) {
    double area = 0.0;
    for (int i = 0, n = p.size(); i < n; i++) {
        area += cross(p[i], p[(i+1)%n]); // wrap back to 0 if i == n-1
    }
    return abs(area) / 2.0;
}
```