2D Geometry

From HPCM Wiki

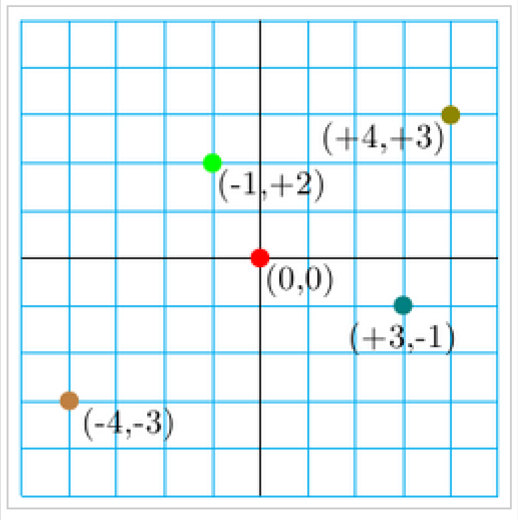
Contents

Points, Distances, and Directions

2D computational geometry concerns the xy-plane. If a point p in the xy-plane has x-coordinate px and y-coordinate py, then we represent p by the coordinate pair (px,py), and write p = (px,py). The xy-plane origin is represented by the point (0,0).

The distance between two points p1 = (p1x,p1y) and p2 = (p2x,p2y) can, by Pythagoras' Theorem, be computed by

```
dx = p2x - p1x
dy = p2y - p1y
distance ( p1, p2 )
    = sqrt ( dx * dx + dy * dy )
```



Example of Points in xy-Plane

We also find it useful to consider the direction from p1 to p2. The angle of this direction can be computed as

```
dx = p2x - p1x
dy = p2y - p1y
angle ( p1, p2 ) = atan2 ( dy, dx )
```

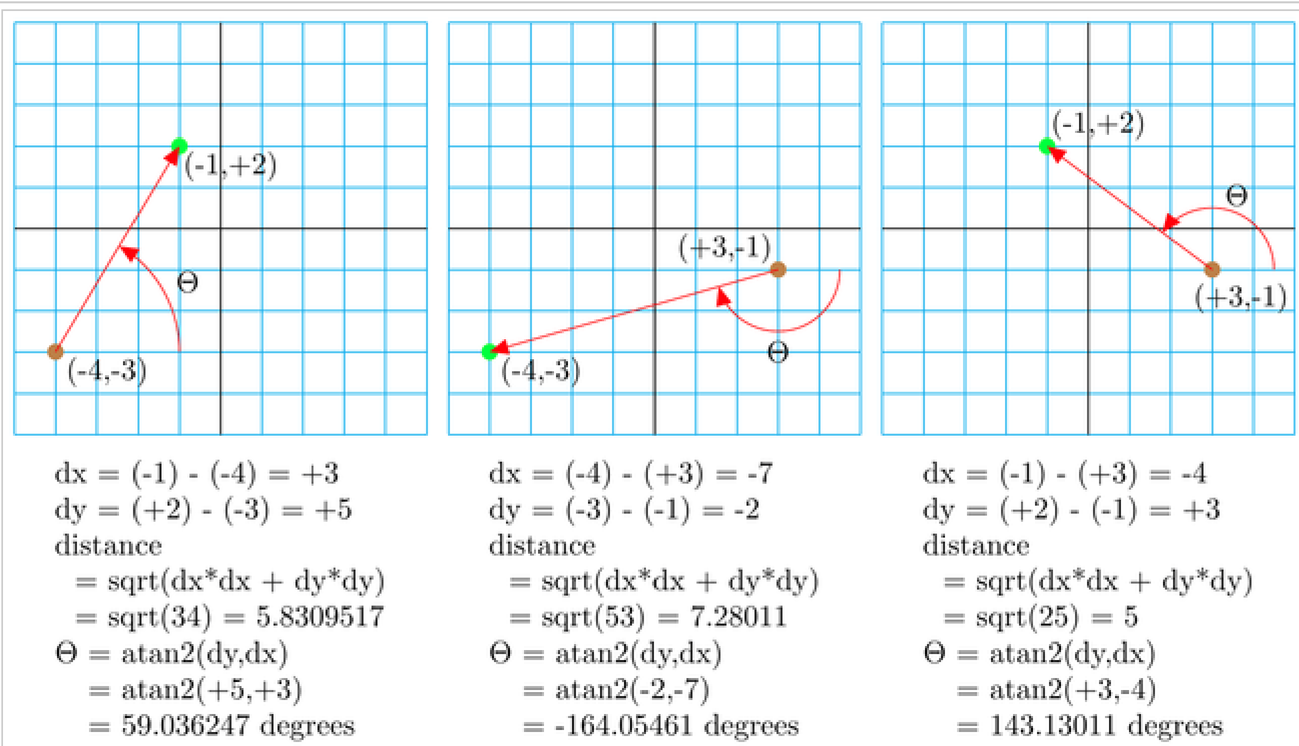
Recall that angles can be measured in either degrees or radians, and π radians = 180 degrees. Also in the xy-plane directions are measured by the angle counter-clockwise from the positive x-axis, and angles clockwise from the positive x-axis are negative. An angle X in degrees is equal to the angle X ± 360, and an angle X in radians is equal to X ± 2π, so in order to assert that two angles X and Y are equal, we write one of

```
X = Y modulo 360 degrees
X = Y modulo 2π radians
```

In order to write code using atan2 and π in different programming languages, use the following includes and names:

```
C:                      C++:                    JAVA:
  #include <float.h>      #include <cfloat>
  #include <math.h>       #include <cmath>
  M_PI                    M_PI                    Math.PI
  atan2                   atan2                   Math.atan2
```

Here atan2(y,x) returns the arc tangent of y/x in radians. It returns positive values if y > 0 and negative values if y < 0. It returns +π/2 (+90 degrees in radians) if y > 0 and x = 0, and -π/2 (-90 degrees in radians) if y < 0 and x = 0. It is undefined only if (x,y) = (0,0), meaning in our situation that angle(p1,p1) is undefined.



Example of Directions in xy-Plane

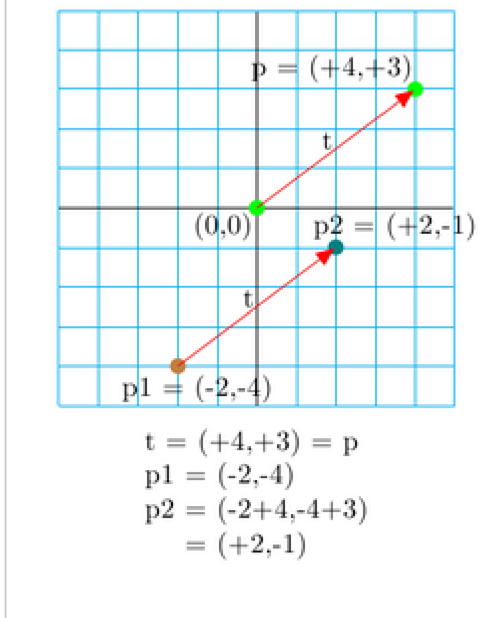
Translations

Translations are a foundational concept of computational geometry. A translation is a motion of all the points in the xy-plane such that all points move the same distance in the same direction.

Given a translation t and any point p1 = (p1x,p1y), t translates p1 to another point p2 = (p2x,p2y). Can we compute the coordinates of p2?

The answer is yes if we know the coordinates of the point that t translates the origin (0,0) to. Suppose these coordinates are (tx,ty); that is, t translates (0,0) to the point p = (tx,ty). Then

```
distance ( p1, p2 )
    = distance ( origin, p )
    = sqrt ( tx*tx + ty*ty )
angle ( p1, p2 )
    = angle ( origin, p )
    = atan2 ( ty, tx )
```



Example 2D Translation

The only way to satisfy these equations is to set

```
p2 = (p1x+tx,p1y+ty)
```

So the translation t is completely determined by (tx,ty), the coordinates of the point to which t translates the origin.

Given this it is natural to define the distance and angle of the translation t as

```
distance ( t ) = sqrt ( tx*tx + ty*ty )
angle ( t ) = atan2 ( ty, tx )
```

Next we define the sum of translations t1 = (t1x,t1y) and t2 = (t2x,t2y), denoted by t1 + t2, to be the motion that takes a point p and translates that first by t1 to a point p1, and then translates p1 by t2 to a point p2. We compute:

```
p = (px,py)
p1 = (p1x,p1y) = (px+t1x,py+t1y)
p2 = (p2x,p2y) = (p1x+t2x,p1y+t2y)
               = (px+t1x+t2x,py+t1y+t2y)
```

From these formulae we can see that the motion t1 + t2 is also a translation and

```
t1 + t2 = (t1x+t2x,t1y+t2y)
```

Lastly we define the `scalar product' of a real number s (which will be called a `scalar' below) with a translation t1 = (tx,ty) to be the translation t2 = s*t defined as follows:

```
if s > 0:
    distance ( t2 )
        = s * distance ( t1 )
    angle ( t2 )
        = angle ( t1 )
```

```
if s < 0:
    distance ( t2 )
        = (- s) * distance ( t1 )
    angle ( t2 )
        = angle ( t1 ) + 180 degrees
```

```
if s = 0:
    distance ( t2 ) = 0
```

Looking at the formulae above one can check that

```
t2 = (t2x,t2y) = (s*t1x,s*t2y)
```

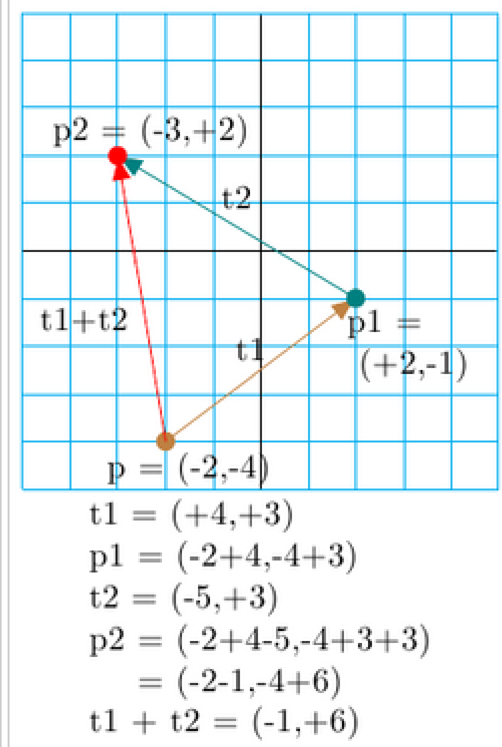
If s > 0, t2 just changes the magnitude of t1, i.e., changes the distance moved without changing the direction. If s > 1 this distance expands, while if 0 < s < 1 this distance contracts. If s == 1 then t2 = t1 and the translation does not change.

If s == -1, t2 is a `reflection' of t1. t2 moves points the same distance as t1 but in the opposite direction.

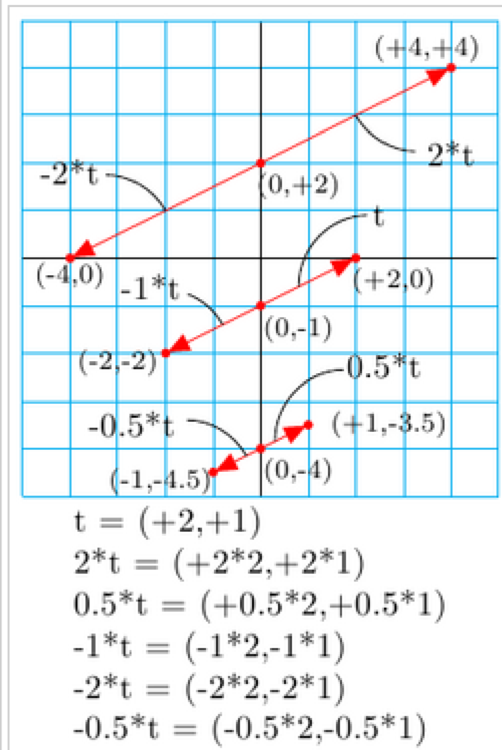
If s == 0, t2 = (0,0), the translation that does nothing to the points.

It can be proved from the above equations for computing s*t and t1+t2 that for any scalars s, s1, s2 and any translations t, t1, t2:

```
s*(t1+t2) = (s*t1) + (s*t2)
s1*(s2*t) = (s1*s2)*t
```



Example 2D Sum of Translations



Example 2D Scalar Products of Translation

Vectors

A 2D `vector' is a mathematician's abstraction. We can represent a vector v by its x and y coordinates like a point: v = (vx,vy).

Vectors can be added:

```
(vx,vy) + (wx,wy) = (vx+wx,vy+wy)
```

subtracted:

```
(vx,vy) - (wx,wy) = (vx-wx,vy-wy)
```

and multiplied by a real number:

```
s * (vx,vy) = (s*vx,s*vy)
```

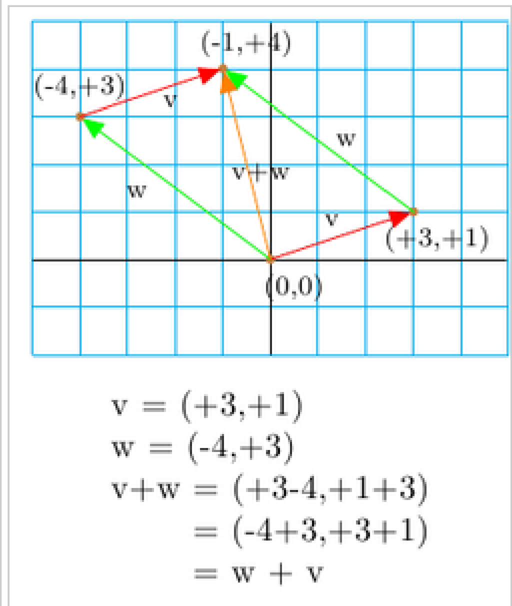
Real numbers such as s are called `scalars' to distinguish them from vectors.

A vector v = (vx,vy) has a length

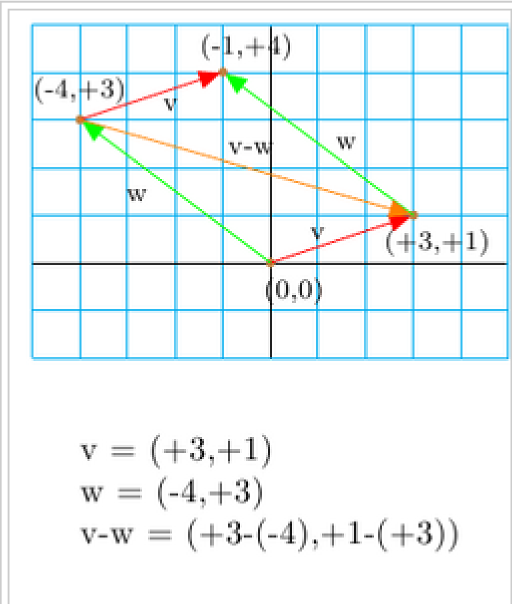
```
||v|| = sqrt ( vx*vx + vy*vy )
```

and direction angle

```
angle ( v ) = atan2 ( vy, vx )
```



Example 2D Vector Addition



Example 2D Vector Subtraction

Vectors can be used to represent points and to represent translations. You compute with vectors, but some represent points and others represent translations.

So for example, given two vectors p1 and p2 representing points, then p2 - p1 represents the translation that moves p1 to p2. Also || p2 - p1 || is the distance between p1 and p2, and angle(p2-p1) is the angle(p1,p2), the direction of p2 when viewed from p1.

To take another example, given a vector p representing a point and another vector t representing a translation, t moves p to p + t. Also ||t|| is the length of t and angle(t) its direction.

Note that vector addition and scalar multiplication obey laws of associativity, commutativity, distributivity, and negation:

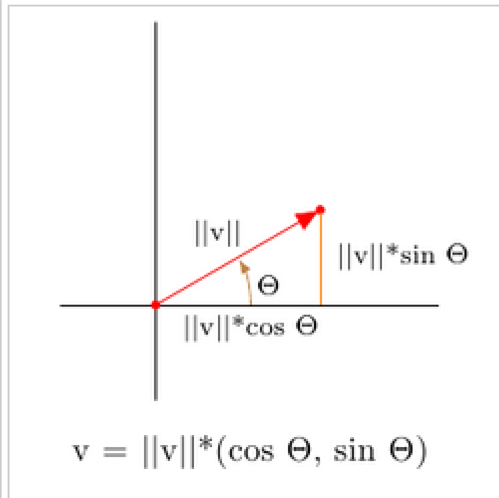
```
v1 + (v2 + v3) = (v1 + v2) + v3
v1 + v2 = v2 + v1
s1*(s2*v) = (s1*s2)*v
s*(v1+v2) = s*v1 + s*v2
v1 - v2 = v1 + (-1)*v2
```

You can easily check these equations using the above definitions. For example,

```
v1 + (-1)*v2 = (v1x,v1y) + (-1)*(v2x,v2y)
             = (v1x,v1y) + ((-1)*v2x,(-1)*v2y)
             = (v1x+(-1)*v2x,v1y+(-1)*v2y)
             = (v1x-v2x,v1y-v2y)
             = v1 - v2
```

It is also sometimes useful to represent a vector in **polar coordinates**, in which the length and angle of the vector are given. Above we give the equations for computing the polar coordinates from the xy coordinates of a vector. To go from polar coordinates to xy coordinates use:

```
v = ||v||*(cos θ, sin θ)
where ||v|| is the length of v
and θ = angle ( v ) is the angle of v
```



2D Vector in Polar Coordinates

Rotations

Rotations are another fundamental concept of 2D computational geometry. The good news is that only one rotation, the left rotation by 90 degrees, needs to be used very often. Nevertheless, understanding rotations is the easy way to understand other concepts such as the scalar product of vectors defined below.

Let R(θ) denote the motion that rotates points about the origin (0,0) by the angle θ in the **counter-clockwise** direction. Note that if θ < 0 then the rotation will be by the angle -θ in the **clockwise** direction. Let v be a vector that represents a point, and R(θ)v represent the point the rotation R(θ) rotates v to. Then

```
angle ( R(θ)v ) = angle ( v ) + θ
||R(θ)v|| = ||v||
```

That is, R(θ) adds θ to the angle of v without changing the length of v.

We can conclude from this that if v is a vector representing a point and s is a scalar,

```
R(θ)(s*v) = s*(R(θ)v)
```

For if s > 0 it multiplies the length of both v and R(θ)v by the same amount, and if s < 0 it does the same but also adds 180 degrees to the angles of both v and R(θ)v.

Now let v1 and v2 be two vectors that represent points. R(θ) moves v1 to R(θ)v1 and similarly v2 to R(θ)v2 without changing the length of either vector and without changing the angle between the vectors. So

```
||R(θ)v1|| = ||v1||
||R(θ)v2|| = ||v2||
angle ( R(θ)v1 ) - angle ( R(θ)v2 )
    = angle ( v1 ) - angle ( v2 )
      modulo 360 degrees
```

Given this and a bit of geometric thinking, we can conclude that

```
R(θ)v2 - R(θ)v1
    = R(θ)(v2-v1)
```

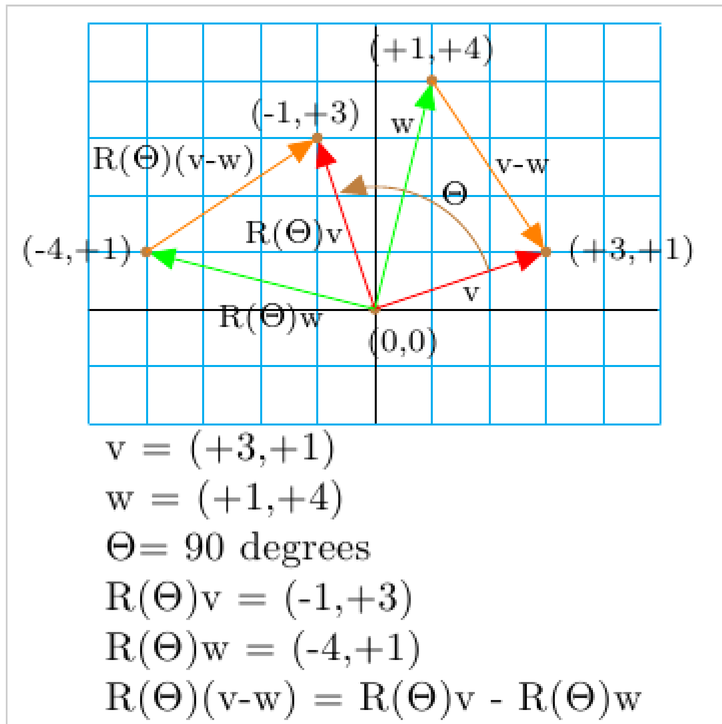
That is, the displacement between the rotated points equals the rotation of the displacement between the unrotated points.

Next observe that if we write v2 = v1 + v3 the above equation becomes

```
R(θ)(v1+v3) = R(θ)v1 + R(θ)v3
```

To summarize what we have learned so far, if s is a scalar and v, v1, v2 are vectors,

```
angle ( R(θ)v ) = angle ( v ) +
||R(θ)v|| = ||v||
R(θ)(s*v) = s*(R(θ)v)
R(θ)(v1+v2) = R(θ)v1 + R(θ)v2
R(θ)(v1-v2) = R(θ)v1 - R(θ)v2
```



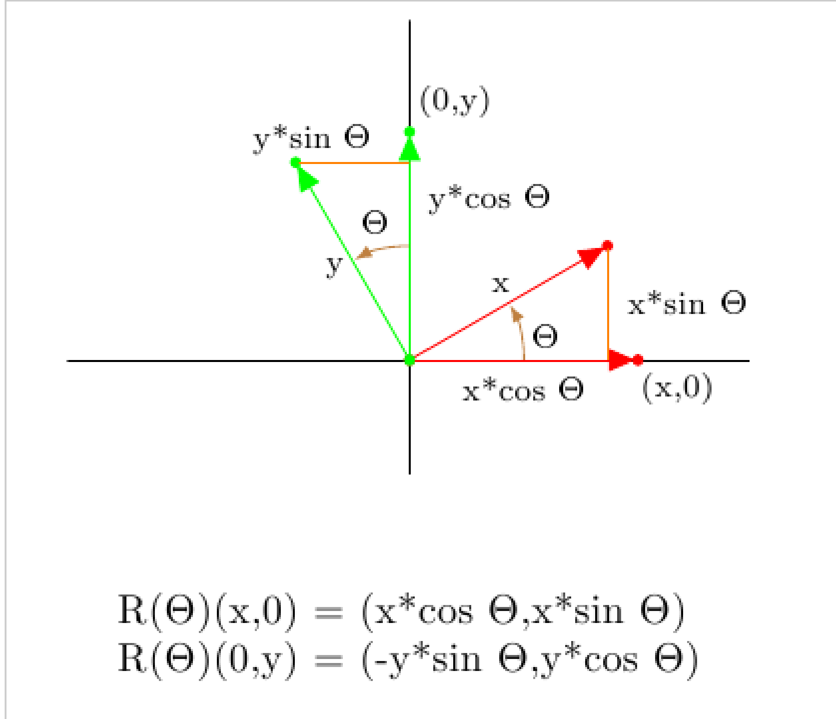
Example 2D Vector Rotation

So now how to compute R(θ)v precisely? A little trigonometry suffices to show that

```
R(θ)(x,0)
  = (cos(θ)x,sin(θ)x)
R(θ)(0,y)
  = (-sin(θ)y,cos(θ)y)
```

Since (x,y) = (x,0) + (0,y) we can combine the above to get

```
R(θ)(x,y)
  = R(θ)(x,0) + R(θ)(0,y)
  =   ( cos(θ)x,sin(θ)y)
    + (-sin(θ)y,cos(θ)y)
  = ( cos(θ)x - sin(θ)y,
      sin(θ)x + cos(θ)y )
```



Example 2D Rotation Coordinates

Just for the fun of it, let us use what we have learned so far to derive some not very obvious trigonometric identities. Consider the rotation by R(θ) of a unit vector u with angle Φ. Then R(θ)u is a unit vector with angle θ+Φ, and we have

```
  u = (cos Φ,sin Φ)
  R(θ)u = (cos (θ+Φ), sin (θ
        = ( cos(θ)*cos(Φ) -
            sin(θ)*cos(Φ) +
```

Looking at just the x coordinates or just the y coordinates we get the following trigonometric identities:

```
  cos (θ+Φ) = cos(θ)*cos(Φ) - sin(θ)sin(Φ)
  sin (θ+Φ) = sin(θ)*cos(Φ) + cos(θ)sin(Φ)
```

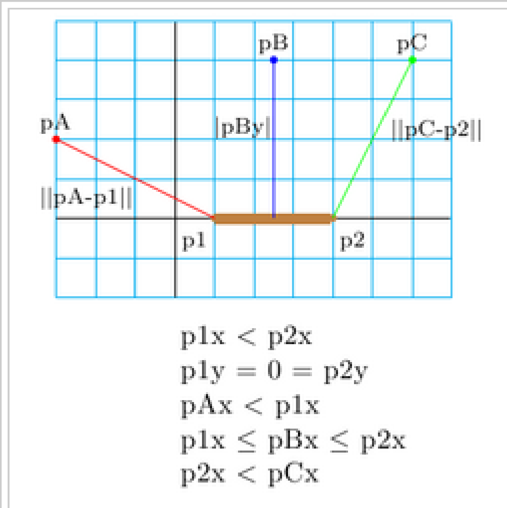
Because it is so useful, let us compute R(+90 degrees):

```
  R(+90 degrees)(x,0) = (0,x)
  R(+90 degrees)(0,y) = (-y,0)
  R(+90 degrees)(x,y) = (-y,x)
```

Coordinate Changes

Many problems in 2D computational geometry can be made fairly easy by changing coordinates. The idea is to make one of the geometric objects involved have easy to manage coordinates. For example, suppose you have been asked to find the distance between a finite line segment and a point. If the line segment lies on the x-axis and has end points p1 = (x1,0) and p2 = (x2,0) with x1 < x2, and if the point is p = (x,y), the answer is:

```
if x < x1:          distance ( p, p1 )
if x1 ≤ x ≤ x2:     |y|
if x > x2:          distance ( p, p2 )
```



Example 2D Distance from a Point to a Line Segment

So if we are given a line segment which is **not** on an axis, how can we change coordinates so it **is** on the x-axis.

Its actually computationally easy, but its harder to see why the easy way works, so we will tread slowly and carefully for a while.

Let p1 = (p1x,p1y) and p2 = (p2x,p2y) not necessarily be on any axis and let p = (x,y) be any point.

First, we can move p1 to the origin by using the translation coordinate change

```
(x',y')     = (x,y)     - (p1x,p1y)
(p1x',p1y') = (p1x,p1y) - (p1x,p1y) = (0,0
(p2x',p2y') = (p2x,p2y) - (p1x,p1y)
```

Now we rotate by minus the angle (p2x',p2y'). Why minus? Because we want to rotate the point (p2x',p2y') so it is on the x-axis. So we get:

```
θ = angle (p2x',p2y')
```

```
(x",y") = R(-θ)(x',y')
        = ( cos(-θ)x' - sin(-θ)y',
            sin(-θ)x' + cos(-θ)y' )
```

```
(p2x",p2y") = R(-θ)(p2x',p2y')
            = ( cos(-θ)p2x' - sin(-θ)p2y',
                sin(-θ)p2x' + cos(-θ)p2y' )
```

This looks a bit messy, but wait, we also have

```
p2' = (p2x',p2y') = ||p2'||*(cos(θ),sin(θ))
```

and

```
cos(-θ) =  cos(θ)
sin(-θ) = -sin(θ)
```

so

```
p2x' = cos(θ)*||p2'|| =   cos(-θ)*||p2'||
p2y' = sin(θ)*||p2'|| = - sin(-θ)*||p2'||
```

```
(x",y") = (p2x'*x'+p2y'*y',-p2y'*x'+p2x'*y') / ||p2'||
```

```
(p2x",p2y") = (p2x'*p2x'+p2y'*p2y',-p2y'*p2x'+p2x'*p2y') / ||p2'||
            = (||p2'||**2,0) / ||p2'||
            = (||p2'||,0)
```

Here (p2x",p2y") = (||p2'||,0) indicates that p2 has been moved to the x-axis, as desired.

The equation for (x",y") looks messy, but in the next sections we will make it look simple by using `scalar products'.

Scalar Products

Coordinate change computations can be managed better with help from the `vector scalar product'. This is defined for two vectors v1 = (v1x,v1y) and v2 = (v2x,v2y) as:

```
v1*v2 = v1x*v2x + v1y*v2y
```

First notice that for any scalar s and vectors v1, v2, v3:

```
v1*v2 = v2*v1
(v1+v2)*v3 = v1*v3 + v2*v3
(s*v1)*v2 = s*(v1*v2)
v1*(v2+v3) = v1*v2 + v1*v3
v1*(s*v2) = s*(v1*v2)
```

Next note that for any vectors v, v1, v2:

```
||v||² = v*v
||v1+v2||² = v1*v1 + 2*v1*v2 + v2*v2
||v1-v2||² = v1*v1 - 2*v1*v2 + v2*v2
v1*v2 = ( ||v1+v2||² - ||v1-v2||² ) / 4
```

Now if R(θ) is a rotation, for any vectors v, v1, v2 we proved above that:

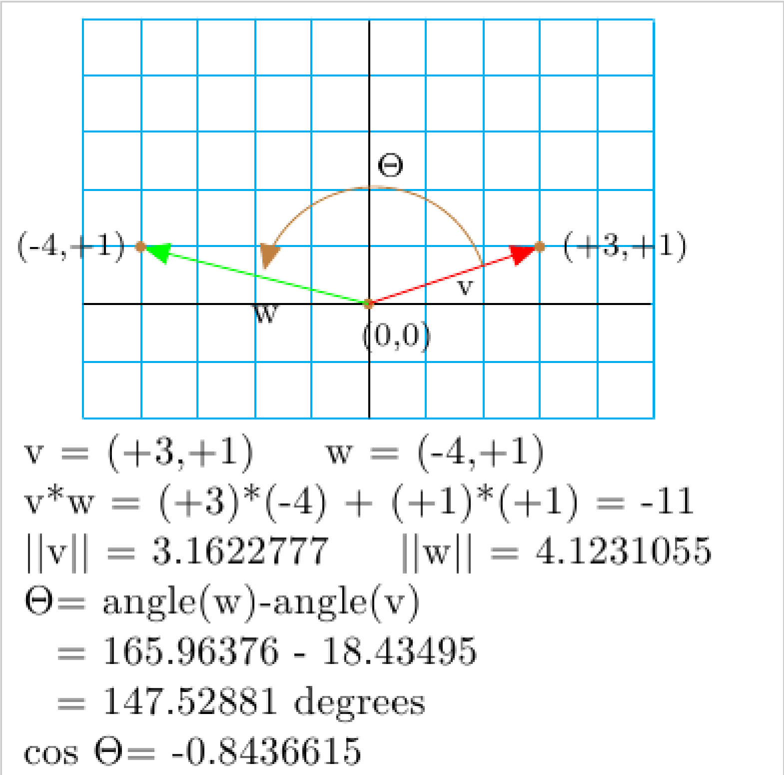
```
||R(θ)v|| = ||v||
R(θ)(v1+v2) = R(θ)v1 + R(θ)v2
R(θ)(v1-v2) = R(θ)v1 - R(θ)v2
```

This means that rotations preserve lengths, sums, and differences of vectors. It follows that rotations also preserve scalar products:

```
R(θ)v1 * R(θ)v1
  = ( ||R(θ)v1 + R(θ)v2||² - ||R(θ)v1 - R(θ)v2||² ) / 4
  = ( ||R(θ)(v1 + v2)||² - ||R(θ)(v1 - v2)||² ) / 4
  = ( ||(v1 + v2)||² - ||(v1 - v2)||² ) / 4
  = v1 * v2
```

We can use the fact that scalar products are not changed by rotations to find a purely geometric definition of the scalar product which makes no reference to a coordinate system. Consider v1 * v2. Now this scalar product does not change if we rotate v1 and v2 so v1 is on the x-axis. So let R(Φ) be a rotation so that v1' = R(Φ)v1 is on the x-axis. Let θ be the angle *between* v1 and v2. Then θ is also the angle between v1' and v2' = R(Φ)v2, and since v1' is on the x-axis, θ is also angle(v2') so:

```
||v1'|| = ||v1||
||v2'|| = ||v2||
θ = angle ( v2' )
v1' = (||v1'||,0)
v2' = ||v2'||(cos θ,sin θ)
v1' * v2'
    = ||v1'|| * ||v2'||
              * cos θ
v1 * v2
    = ||v1|| * ||v2||
             * cos θ
```



Example 2D Vector Scalar Product

So v1 * v2 is the product of the length of v1, the length of v2, and the cosine of the angle between v1 and v2.

Coordinate Change and Scalar Product

Now lets go back and redo the coordinate change we did above using the scalar product as an aid.

We start with a line segment with end-points p1 and p2 and another point p. Think of these as vectors. Then the first coordinate change was translation by -p1 so:

```
   p'  = p  - p1
   p1' = p1 - p1 = (0,0)
   p2' = p2 - p1
```

Now let u be the unit length vector in the same direction as p2', so

```
   u = p2' / ||p2'||
   u = (cos(θ),sin(θ))
   θ = angle(u)
```

We next rotate by - angle(u), and looking back at the equations we developed in the Coordinate Changes section above we find that

```
   p"  = (p'*u,p'*R(90 degrees)u)
   p1" = (0,0)
   p2" = (||p2'||,0)
```

Given this we see that a coordinate system is specified by its origin point p1, the direction of its x-axis u, and the direction of its y-axis R(90 degrees)u.

What happens if we leave out the translation which changes the origin?

Then the new coordinates of p are

```
p" = (p*u,p*R(90 degrees)u)
```

where u specifies the same direction as before, and in the new coordinates that direction is parallel to the new x-axis. If we re-examine the problem of finding the distance between a point p an a line segment with ends p1 and p2, we find that in the new coordinates the line segment is still parallel to the x-axis but not generally on the x-axis.

And it is still true that p1"x < p2"x, as

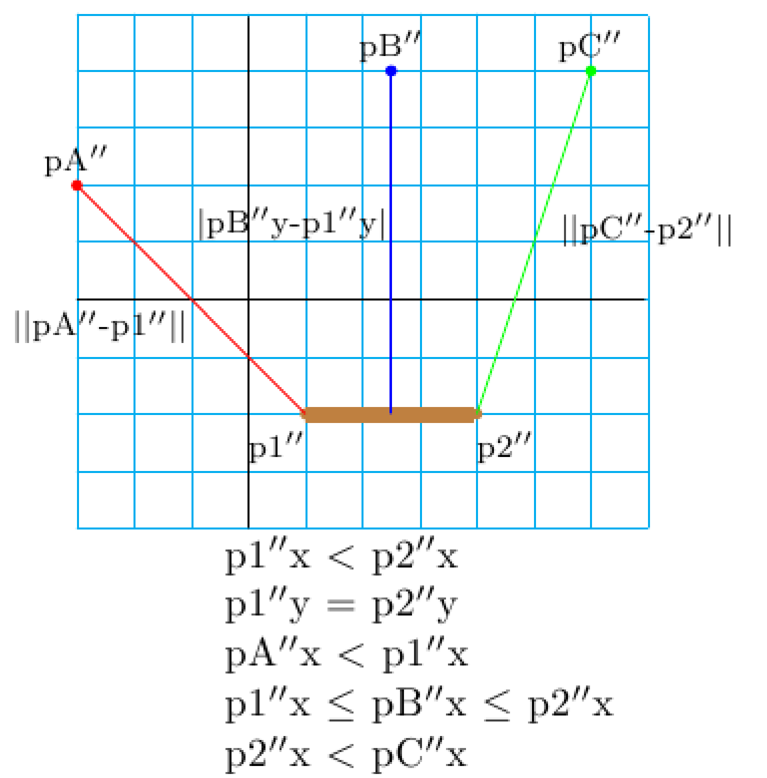
```
(p2"x - p1"x)
  = (p2*u - p1*u)
  = (p2 - p1) * u
  =   (p2 - p1) * (p2 - p1)
    / ||p2 - p1||
  = ||p2 - p1||
  > 0
```

So we need to modify our equations thusly:

```
  if p"x < p1"x:             dist
  if p1"x ≤ p"x ≤ p2"x:      |p"y
  if p"x > p2"x:             dist
```

where

```
  u = ( p2 - p1 ) / ||p2 - p1|
  p1"y = p2"y
  p1"x < p2"x
```



Example 2D Distance from a Point to a Line Segment

What happens if instead of using a unit vector u we use a vector v of arbitrary length?

Then the new coordinates of p are

```
  p" = (p*v,p*R(90 degrees)v)
  distance" = ||v||*distance
```

That is, things are as before **but** distances in the new coordinate systems are ||v|| times distances in the old coordinate systems because we have **not** divided the new coordinates by ||v||, which we would have done if we wanted to use a unit length vector u = v/||v||| in place of v.

So now our equations become:

```
  if p"x < p1"x:             distance ( p", p1" ) / ||v||
  if p1"x ≤ p"x ≤ p2"x:      |p"y - p1"y| / ||v||
  if p"x > p2"x:             distance ( p", p2" ) / ||v||
```

where

```
v = p2 - p1
p1"y = p2"y
p1"x < p2"x
```

Applications

Distance Between Point and Infinite Line

Let p be a point and consider the infinite line through the two points p1 and p2. What is the distance from p to the line?

Changing coordinates so that v = p2 - p1 becomes parallel to the x-axis gives:

```
v = p2 - p1
p' = ( p*v, p*R(90 degrees)v )
distance' = ||v||*distance
p1'y = p2'y
distance ( p, line ) = |p'y - p1'y| / ||v||
                     = | (p' - p1)*R(90 degrees)v | / ||v||
```

Distance Between Point and Finite Line Segment

Let p be a point and consider the finite line segment with endpoints p1 and p2. What is the distance from p to the line segment?

Changing coordinates so that v = p2 - p1 becomes parallel to the x-axis gives:

```
v = p2 - p1
p' = ( p*v, p*R(90 degrees)v )
distance' = ||v||*distance
p1'y = p2'y
p1'x < p2'x
distance ( p, line segment ) =
    if p'x < p1'x:           distance ( p', p1' ) / ||v||
    if p1'x ≤ p'x ≤ p2'x:    |p'y - p1'y| / ||v||
    if p2'x < p'x:           distance ( p', p2' ) / ||v||
```

Sides of an Infinite Line

Let p be a point and consider the infinite **directed** line from p1 to p2. Is p to the left of, on, or to the right of the line?

Changing coordinates so that v = p2 - p1 becomes parallel to the x-axis gives:

```
v = p2 - p1
p' = ( p*v, p*R(90 degrees)v )
distance' = ||v||*distance
p1'y = p2'y = p1*R(90 degrees)v
if p'y > p1'y:    p is to the left of the line
if p'y = p1'y:    p is on the line
if p'y < p1'y:    p is to the right of the line
```

The advice given below about Testing Equality applies.

Testing Equality

There can be a problem in testing = of numbers. Point coordinates are usually input as multiples of some number U. Then the scalar products and changed coordinates (if you do **not** divide by ||v||) are multiples of U^2. If you have two numbers X and Y that are very close approximations to multiples of U^2, then:

```
to test if X > Y:     test if X > Y + U²/2
to test if X = Y:     test if Y - U²/2 < X < Y + U²/2
to test if X < Y:     test if X < Y - U²/2
```

So, for example, if U = 0.01 and X and Y are very close approximations to multiples of U^2 = 0.0001, these test are:

```
to test if X > Y:     test if X > Y + 0.00005
to test if X = Y:     test if Y - 0.00005 < X < Y + 0.00005
to test if X < Y:     test if X < Y - 0.00005
```

Note that its important to **not** divide by ||v|| before making these comparisons - if you did the numbers would be multiples of $U^2/||v||$. Also note that if U = 1, then the coordinates and scalar products are all integers, and this extra twist to testing inequalities is unnecessary (note that using `double's to store and compute with integers works precisely as long as you only add, subtract, multiply, and compare, and no integer has more than 15 decimal digits).

Convex Hull

Find the convex hull of a set S of points.

By definition, the convex hull is the smallest convex set containing S. Its perimeter is a polygon whose vertices are in S, and if p1 and p2 are successive vertices on a counter-clockwise trip around this polygon, all the points of S are to the left of or on the infinite line from p1 to p2. We can use this to find the perimeter of the hull.

First pick a leftmost point of S, and if there are several, pick the bottom-most. Call it p(0); it is a vertex of the perimeter of the hull. Then for each i, beginning with i = 0, pick a point p(i+1) of S such that all the points of S are to the left of or on the infinite line from p(i) to p(i+1). Eventually for some N we will have p(N) = p(0) and we are done.

This is not a very efficient way of computing the convex hull perimeter, so if S is very large some more efficient way should be used.

Does a Finite Line Segment Intersect an Infinite Line

Let p3 and p4 be the ends of a finite line segment and consider the infinite line through p1 and p2. Does the finite line segment intersect the infinite line segment?

Consider the **directed** infinite line from p1 to p2. The answer is yes if and only if either p3 or p4 is on this infinite line, or one is to its left and the other to its right. See Sides of an Infinite Line to see how to determine whether a point is to the left or right of a directed infinite line.

Changing coordinates so that v = p2 - p1 becomes parallel to the x-axis gives:

```
v = p2 - p1
p1'y = p2'y = p1*R(90 degrees)v
if (p3'y - p1'y) * (p4'y - p1'y) ≤ 0:    yes
else:                                    no
```

The advice given above about Testing Equality applies. If initial coordinates are integers, you may want to replace

```
(p3'y - p1'y) * (p4'y - p1'y) ≤ 0
```

by

```
not ( (p3'y < p1'y and p4'y < p1'y) or (p3'y > p1'y and p4'y > p1'y) )
```

as the product may overflow.

Where Does a Finite Line Segment Intersect an Infinite Line

Let p3 and p4 be the ends of a finite line segment and consider the infinite line through p1 and p2. Does the finite line segment intersect the infinite line segment, and if so, where?

The points p of the finite line segment can be represented by linear combinations of p3 and p4, thus:

```
p = t*p3 + (1-t)*p4
0 ≤ t ≤ 1
```

So the problem is to find if there exist p and t satisfying these equations such that p is on the infinite line. Now if w is any vector, the equation for p gives

```
p*w = t*(p3*w) + (1-t)*(p4*w)
```

and we use this with w = R(90 degrees)v below to determine t.

Let p be the intersection point, if it exists, and change coordinates so that v = p2 - p1 becomes parallel to the x-axis:

```
v = p2 - p1
distance' = ||v||*distance
w = R(90 degrees)v
p'y = p1'y = p2'y = p*w = p1*w = p1*w
p3'y = p3*w
p4'y = p4*w
t*p3'y + (1-t)*p4'y = p'y = p1'y
(p3'y - p4'y)*t = p1'y - p4'y
```

We can solve this for t, except in some cases. Also if we get a solution with t < 0 or t > 1, the intersection is off the end of the finite line segment, and therefore the finite line segment does **not** intersect the infinite line. More specifically,

```
if p3'y == p4'y:
    // the line segment is parallel to the infinite line
    if p1'y = p4'y:
         // the line segment lies ON the infinite line
         ALL t are solutions; pick any t, say t = 0
    else if p1'y ≠ p4'y:
         // the line segment does NOT lie on the infinite line
         there are NO solutions
else if p3'y ≠ p4'y:
    let t = ( p1'y - p4'y ) / ( p3'y - p4'y )
    if 0 ≤ t ≤ 1:
        there is an intersection at p = t*p3 + (1-t)*p4
    else:
        there is no intersection
        // p is off the end of the finite line segment
```

The advice given above about Testing Equality applies. It may be safer to test 0 ≤ t ≤ 1 by avoiding the division and testing if either:

```
0 ≤ p1'y - p4'y ≤ p3'y - p4'y
```

or

```
0 ≥ p1'y - p4'y ≥ p3'y - p4'y
```

Do Two Finite Lines Intersect

Given a finite line with end points p1 and p2 and another finite line with end points p3 and p4, to these lines intersect?

First, note that if one of the finite lines does NOT intersect the infinite line that contains the other finite line, then the two finite lines do NOT intersect. See Does a Finite Line Segment Intersect an Infinite Line.

This is almost a complete algorithm for determining when two finite lines do NOT intersect, and therefore, determining when they DO intersect. But there is a special case when both finite lines are on the same infinite line.

In this special case, change coordinates so one of the finite lines is horizontal. Then the other is horizontal with the same y coordinate. Let the x coordinates of the points in the changed coordinate system be p1x, p2x, p3x, p4x and swapping endpoints of a finite line if necessary make it such that p1x < p2x and p3x < p4x. Then the two lines do NOT intersect if and only if either p2x < p3x or p4x < p1x.

Find the Shortest Path Outside Convex Polygons

Given a set of disjoint convex polygons and two points outside any polygon, what is the shortest path between the two points that does not enter the interior of any polygon.

The answer is a sequence of line segments connecting points that are either one of the two points outside the polygons or polygon vertices. The problem is to figure out which of these line segments do not enter the interior of any polygon, and then the problem becomes a search for a shortest path through an undirected graph (which is dealt with in Search and Shortest Paths).

So we want to take the set of all line segments whose end points are each one of the two points outside polygons or polygon vertices, and exclude those whose interiors intersect the interior of some polygon. Actually, we can go farther and exclude those whose interiors contain a polygon vertex, because the vertex can be used to split the line in two, and the two lines can replace the longer line in the path.

Therefore it suffices to exclude just those line segments whose interiors intersect a polygon edge, plus those line segments both of whose endpoints are non-adjacent in the same polygon (as the polygons are convex - the case of a non-convex polygon is harder).

So how do we figure whether the interior of a line segment L intersects a polygon edge E. We use a modification of the method of Do Finite Lines Intersect. We check that E intersects the infinite line that extends L, and then we check that the **interior** of L intersects the infinite line that extends E. We perform these checks using the method of Does a Finite Line Intersect an Infinite Line, but in the second use we change the equation

```
(p3'y - p1'y) * (p4'y - p1'y) ≤ 0
```

to

```
(p3'y - p1'y) * (p4'y - p1'y) < 0
```

in order to require that the **interior** of L intersects the infinite line extending E.

Other Applications

Do a circle and an infinite line intersect? If so, where?

Do a circle and a finite line segment intersect? If so where?

Do two circles intersect? If so where?

Given a point and a circle, find the infinite lines through the point that are tangent to the circle. Find where these intersect the circle.

Given two circles, find the infinite lines tangent to both. Find where these intersect the circle.

Given a set of circles and two points outside any circle, what is the shortest path between the two points that does not enter the interior of any circle.