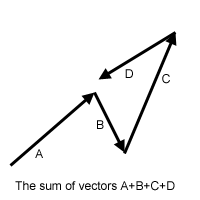
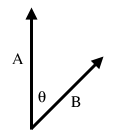
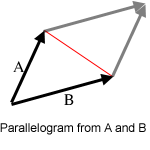
Vector Addition  
There are a number of mathematical operations that can be performed on vectors. The simplest of these is addition: you can add two vectors together and the result is a new vector. If you have two vectors (x1, y1) and (x2, y2), then the sum of the two vectors is simply (x1+x2, y1+y2). The image below shows the sum of four vectors. Note that it doesn't matter which order you add them up in - just like regular addition. Throughout these articles, we will use plus and minus signs to denote vector addition and subtraction, where each is simply the piecewise addition or subtraction of the components of the vector.



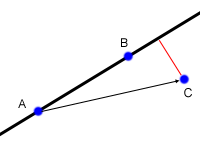
Dot Product  
The addition of vectors is relatively intuitive; a couple of less obvious vector operations are dot and cross products. The dot product of two vectors is simply the sum of the products of the corresponding elements. For example, the dot product of (x1, y1) and (x2, y2) is x1\*x2 + y1\*y2. Note that this is not a vector, but is simply a single number (called a scalar). The reason this is useful is that the dot product, A ⋅ B = |A||B|Cos(θ), where θ is the angle between the A and B. |A| is called the norm of the vector, and in a 2-D geometry problem is simply the length of the vector, sqrt(x2+y2). Therefore, we can calculate Cos(θ) = (A ⋅ B)/(|A||B|). By using the acos function, we can then find θ. It is useful to recall that Cos(90) = 0 and Cos(0) = 1, as this tells you that a dot product of 0 indicates two perpendicular lines, and that the dot product is greatest when the lines are parallel. A final note about dot products is that they are not limited to 2-D geometry. We can take dot products of vectors with any number of elements, and the above equality still holds.



Cross Product  
An even more useful operation is the cross product. The cross product of two 2-D vectors is x1\*y2 - y1\*x2 Technically, the cross product is actually a vector, and has the magnitude given above, and is directed in the +z direction. Since we're only working with 2-D geometry for now, we'll ignore this fact, and use it like a scalar. Similar to the dot product, A x B = |A||B|Sin(θ). However, θ has a slightly different meaning in this case: |θ| is the angle between the two vectors, but θ is negative or positive based on the right-hand rule. In 2-D geometry this means that if A is less than 180 degrees clockwise from B, the value is positive. Another useful fact related to the cross product is that the absolute value of |A||B|Sin(θ) is equal to the area of the parallelogram with two of its sides formed by A and B. Furthermore, the triangle formed by A, B and the red line in the diagram has half of the area of the parallelogram, so we can calculate its area from the cross product also.



Line-Point Distance  
Finding the distance from a point to a line is something that comes up often in geometry problems. Lets say that you are given 3 points, A, B, and C, and you want to find the distance from the point C to the line defined by A and B (recall that a line extends infinitely in either direction). The first step is to find the two vectors from A to B (AB) and from A to C (AC). Now, take the cross product AB x AC, and divide by |AB|. This gives you the distance (denoted by the red line) as (AB x AC)/|AB|. The reason this works comes from some basic high school level geometry. The area of a triangle is found as base\*height/2. Now, the area of the triangle formed by A, B and C is given by (AB x AC)/2. The base of the triangle is formed by AB, and the height of the triangle is the distance from the line to C. Therefore, what we have done is to find twice the area of the triangle using the cross product, and then divided by the length of the base. As always with cross products, the value may be negative, in which case the distance is the absolute value.



Things get a little bit trickier if we want to find the distance from a line segment to a point. In this case, the nearest point might be one of the endpoints of the segment, rather than the closest point on the line. In the diagram above, for example, the closest point to C on the line defined by A and B is not on the segment AB, so the point closest to C is B. While there are a few different ways to check for this special case, one way is to apply the dot product. First, check to see if the nearest point on the line AB is beyond B (as in the example above) by taking AB ⋅ BC. If this value is greater than 0, it means that the angle between AB and BC is between -90 and 90, exclusive, and therefore the nearest point on the segment AB will be B. Similarly, if BA ⋅ AC is greater than 0, the nearest point is A. If both dot products are negative, then the nearest point to C is somewhere along the segment. (There is another way to do this, which I'll discuss [here](http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=geometry3#satellites)).

//Compute the dot product AB ⋅ BC

int dot(int[] A, int[] B, int[] C){

AB = new int[2];

BC = new int[2];

AB[0] = B[0]-A[0];

AB[1] = B[1]-A[1];

BC[0] = C[0]-B[0];

BC[1] = C[1]-B[1];

int dot = AB[0] \* BC[0] + AB[1] \* BC[1];

return dot;

}

//Compute the cross product AB x AC

int cross(int[] A, int[] B, int[] C){

AB = new int[2];

AC = new int[2];

AB[0] = B[0]-A[0];

AB[1] = B[1]-A[1];

AC[0] = C[0]-A[0];

AC[1] = C[1]-A[1];

int cross = AB[0] \* AC[1] - AB[1] \* AC[0];

return cross;

}

//Compute the distance from A to B

double distance(int[] A, int[] B){

int d1 = A[0] - B[0];

int d2 = A[1] - B[1];

return sqrt(d1\*d1+d2\*d2);

}

//Compute the distance from AB to C

//if isSegment is true, AB is a segment, not a line.

double linePointDist(int[] A, int[] B, int[] C, boolean isSegment){

double dist = cross(A,B,C) / distance(A,B);

if(isSegment){

int dot1 = dot(A,B,C);

if(dot1 > 0)return distance(B,C);

int dot2 = dot(B,A,C);

if(dot2 > 0)return distance(A,C);

}

return abs(dist);

}

That probably seems like a lot of code, but lets see the same thing with a point class and some operator overloading in C++ or C#. The \* operator is the dot product, while ^ is cross product, while + and - do what you would expect.

//Compute the distance from AB to C

//if isSegment is true, AB is a segment, not a line.

double linePointDist(point A, point B, point C, bool isSegment){

double dist = ((B-A)^(C-A)) / sqrt((B-A)\*(B-A));

if(isSegment){

int dot1 = (C-B)\*(B-A);

if(dot1 > 0)return sqrt((B-C)\*(B-C));

int dot2 = (C-A)\*(A-B);

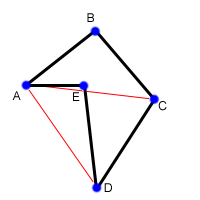
if(dot2 > 0)return sqrt((A-C)\*(A-C));

}

return abs(dist);

}

Operator overloading is beyond the scope of this article, but I suggest that you look up how to do it if you are a C# or C++ coder, and write your own 2-D point class with some handy operator overloading. It will make a lot of geometry problems a lot simpler.  
  
Polygon Area  
Another common task is to find the area of a polygon, given the points around its perimeter. Consider the non-convex polygon below, with 5 points. To find its area we are going to start by triangulating it. That is, we are going to divide it up into a number of triangles. In this polygon, the triangles are ABC, ACD, and ADE. But wait, you protest, not all of those triangles are part of the polygon! We are going to take advantage of the signed area given by the cross product, which will make everything work out nicely. First, we'll take the cross product of AB x AC to find the area of ABC. This will give us a negative value, because of the way in which A, B and C are oriented. However, we're still going to add this to our sum, as a negative number. Similarly, we will take the cross product AC x AD to find the area of triangle ACD, and we will again get a negative number. Finally, we will take the cross product AD x AE and since these three points are oriented in the opposite direction, we will get a positive number. Adding these three numbers (two negatives and a positive) we will end up with a negative number, so will take the absolute value, and that will be area of the polygon.



The reason this works is that the positive and negative number cancel each other out by exactly the right amount. The area of ABC and ACD ended up contributing positively to the final area, while the area of ADE contributed negatively. Looking at the original polygon, it is obvious that the area of the polygon is the area of ABCD (which is the same as ABC + ABD) minus the area of ADE. One final note, if the total area we end up with is negative, it means that the points in the polygon were given to us in clockwise order. Now, just to make this a little more concrete, lets write a little bit of code to find the area of a polygon, given the coordinates as a 2-D array, p.

int area = 0;

int N = lengthof(p);

//We will triangulate the polygon

//into triangles with points p[0],p[i],p[i+1]

for(int i = 1; i+1<N; i++){

int x1 = p[i][0] - p[0][0];

int y1 = p[i][1] - p[0][1];

int x2 = p[i+1][0] - p[0][0];

int y2 = p[i+1][1] - p[0][1];

int cross = x1\*y2 - x2\*y1;

area += cross;

}

return abs(cross/2.0);

Notice that if the coordinates are all integers, then the final area of the polygon is one half of an integer.

Line-Line Intersection  
One of the most common tasks you will find in geometry problems is line intersection. Despite the fact that it is so common, a lot of coders still have trouble with it. The first question is, what form are we given our lines in, and what form would we like them in? Ideally, each of our lines will be in the form Ax+By=C, where A, B and C are the numbers which define the line. However, we are rarely given lines in this format, but we can easily generate such an equation from two points. Say we are given two different points, (x1, y1) and (x2, y2), and want to find A, B and C for the equation above. We can do so by setting   
A = y2-y1  
B = x1-x2  
C = A\*x1+B\*y1  
Regardless of how the lines are specified, you should be able to generate two different points along the line, and then generate A, B and C. Now, lets say that you have lines, given by the equations:  
A1x + B1y = C1  
A2x + B2y = C2  
To find the point at which the two lines intersect, we simply need to solve the two equations for the two unknowns, x and y.

double det = A1\*B2 - A2\*B1

if(det == 0){

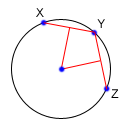
//Lines are parallel

}else{

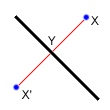
double x = (B2\*C1 - B1\*C2)/det

double y = (A1\*C2 - A2\*C1)/det }

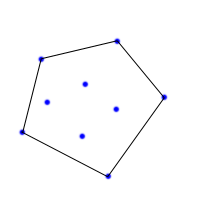
To see where this comes from, consider multiplying the top equation by B2, and the bottom equation by B1. This gives you  
A1B2x + B1B2y = B2C1  
A2B1x + B1B2y = B1C2  
Now, subtract the bottom equation from the top equation to get  
A1B2x - A2B1x = B2C1 - B1C2  
Finally, divide both sides by A1B2 - A2B1, and you get the equation for x. The equation for y can be derived similarly.   
  
This gives you the location of the intersection of two lines, but what if you have line segments, not lines. In this case, you need to make sure that the point you found is on both of the line segments. If your line segment goes from (x1,y1) to (x2,y2), then to check if (x,y) is on that segment, you just need to check that min(x1,x2) ≤ x ≤ max(x1,x2), and do the same thing for y. You must be careful about double precision issues though. If your point is right on the edge of the segment, or if the segment is horizontal or vertical, a simple comparison might be problematic. In these cases, you can either do your comparisons with some tolerance, or else use a fraction class.  
  
Finding a Circle From 3 Points  
Given 3 points which are not colinear (all on the same line) those three points uniquely define a circle. But, how do you find the center and radius of that circle? This task turns out to be a simple application of line intersection. We want to find the perpendicular bisectors of XY and YZ, and then find the intersection of those two bisectors. This gives us the center of the circle.



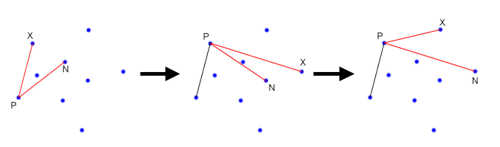
To find the perpendicular bisector of XY, find the line from X to Y, in the form Ax+By=C. A line perpendicular to this line will be given by the equation -Bx+Ay=D, for some D. To find D for the particular line we are interested in, find the midpoint between X and Y by taking the midpoint of the x and y components independently. Then, substitute those values into the equation to find D. If we do the same thing for Y and Z, we end up with two equations for two lines, and we can find their intersections as described above.  
  
Reflection  
Reflecting a point across a line requires the same techniques as finding a circle from 3 points. First, notice that the distance from X to the line of reflection is the same as the distance from X' to the line of reflection. Also note that the line between X and X' is perpendicular to the line of reflection. Now, if the line of reflection is given as Ax+By=C, then we already know how to find a line perpendicular to it: -Bx+Ay=D. To find D, we simply plug in the coordinates for X. Now, we can find the intersection of the two lines at Y, and then find X' = Y - (X - Y).



Rotation  
Rotation doesn't really fit in with line intersection, but I felt that it would be good to group it with reflection. In fact, another way to find the reflected point is to rotate the original point 180 degrees about Y.  
  
Imagine that we want to rotate one point around another, counterclockwise by θ degrees. For simplicity, lets assume that we are rotating about the origin. In this case, we can find that x' = x Cos(θ) - y Sin(θ) and y' = x Sin(θ) + y Cos(θ). If we are rotating about a point other than the origin, we can account for this by shifting our coordinate system so that the origin is at the point of rotation, doing the rotation with the above formulas, and then shifting the coordinate system back to where it started.  
  
Convex Hull  
A convex hull of a set of points is the smallest convex polygon that contains every one of the points. It is defined by a subset of all the points in the original set. One way to think about a convex hull is to imagine that each of the points is a peg sticking up out of a board. Take a rubber band and stretch it around all of the points. The polygon formed by the rubber band is a convex hull. There are many different algorithms that can be used to find the convex hull of a set of points. In this article, I'm just going to describe one of them, which is fast enough for most purposes, but is quite slow compared to some of the other algorithms.



First, loop through all of your points and find the leftmost point. If there is a tie, pick the highest point. You know for certain that this point will be on the convex hull, so we'll start with it. From here, we are going to move clockwise around the edge of the hull, picking the points on the hull, one at a time. Eventually, we will get back to the start point. In order to find the next point around the hull, we will make use of [cross products](http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=geometry1#cross_product). First, we will pick an unused point, and set the next point, N, to that point. Next, we will iterate through each unused points, X, and if (X-P) x (N-P) (where P is the previous point) is negative, we will set N to X. After we have iterated through each point, we will end up with the next point on the convex hull. See the diagram below for an illustration of how the algorithm works. We start with P as the leftmost point. Now, say that we have N and X as shown in the leftmost frame. In this case the cross product will be negative, so we will set N = X, and there will be no other unused points that make the cross product negative, and hence we will advance, setting P = N. Now, in the next frame, we will end up setting N = X again, since the cross product here will be negative. However, we aren't done yet because there is still another point that will make the cross product negative, as shown in the final frame.



The basic idea here is that we are using the cross product to find the point which is furthest counterclockwise from our current position at P. While this may seem fairly straightforward, it becomes a little bit tricky when dealing with colinear points. If you have no colinear points on the hull, then the code is very straightforward.

convexHull(point[] X){

int N = lengthof(X);

int p = 0;

//First find the leftmost point

for(int i = 1; i<N; i++){

if(X[i] < X[p])

p = i;

}

int start = p;

do{

int n = -1;

for(int i = 0; i<N; i++){

//Don't go back to the same point you came from

if(i == p)continue;

//If there is no N yet, set it to i

if(n == -1)n = i;

int cross = (X[i] - X[p]) x (X[n] - X[p]);

if(cross < 0){

//As described above, set N=X

n = i;

}

}

p = n;

}while(start!=p);

}

Once we start to deal with colinear points, things get trickier. Right away we have to change our method signature to take a boolean specifying whether to include all of the colinear points, or only the necessary ones.

//If onEdge is true, use as many points as possible for

//the convex hull, otherwise as few as possible.

convexHull(point[] X, boolean onEdge){

int N = lengthof(X);

int p = 0;

boolean[] used = new boolean[N];

//First find the leftmost point

for(int i = 1; i<N; i++){

if(X[i] < X[p])

p = i;

}

int start = p;

do{

int n = -1;

int dist = onEdge?INF:0;

for(int i = 0; i<N; i++){

//X[i] is the X in the discussion

//Don't go back to the same point you came from

if(i==p)continue;

//Don't go to a visited point

if(used[i])continue;

//If there is no N yet, set it to X

if(n == -1)n = i;

int cross = (X[i] - X[p]) x (X[n] - X[p]);

//d is the distance from P to X

int d = (X[i] - X[p]) ⋅ (X[i] - X[p]);

if(cross < 0){

//As described above, set N=X

n = i;

dist = d;

}else if(cross == 0){

//In this case, both N and X are in the

//same direction. If onEdge is true, pick the

//closest one, otherwise pick the farthest one.

if(onEdge && d < dist){

dist = d;

n = i;

}else if(!onEdge && d > dist){

dist = d;

n = i;

}

}

}

p = n;

used[p] = true;

}while(start!=p);

}