# Misc Geometric Formula:

|  |  |
| --- | --- |
| **Triangle** | Circum Radius = a\*b\*c/(4\*area)  In Radius = area/s, where s = (a+b+c)/2  length of median to side c = sqrt(2\*(a\*a+b\*b)-c\*c)/2  length of bisector of angle C = sqrt(ab[(a+b)\*(a+b)-c\*c])/(a+b) |
| **Ellipse** | Area = PI\*a\*b  Circumference = 4a \*int(0,PI/2){sqrt(1-(k\*sint)\*(k\*sint))}dt  = 2\*PI\*sqrt((a\*a+b\*b)/2) approx  where k = sqrt((a\*a-b\*b)/a)  = PI\*(3\*(r1+r2)-sqrt[(r1+3\*r2)\*(3\*r1+r2)]) |
| **Spherical cap** | V = (1/3)\*PI\*h\*h\*(3\*r-h)  Surface Area = 2\*PI\*r\*h |
| **Spherical Sector** | V = (2/3)\*PI\*r\*r\*h |
| **Spherical Segment** | V = (1/6)\*PI\*h\*(3\*a\*a+3\*b\*b+h\*h) |
| **Torus** | V = 2\*PI\*PI\*R\*r\*r |
| **Truncated Conic** | V = (1/3)\*PI\*h\*(a\*a+a\*b+b\*b)  Surface Area = PI\*(a+b)\*sqrt(h\*h+(b-a)\*(b-a))  = PI\*(a+b)\*l |
| **Pyramidal frustum** | (1/3)\*h\*(A1+A2+sqrt(A1\*A2)) |

# Misc Trigonometric Functions and Formulas:

tan A/2 = +sqrt((1-cos A)/(1+cos A))

= sin A / (1+cos A)

= (1-cos A) / sin A

= cosec A – cot A

sin 3A = 3\*sin A – 4\*sincube A cos 3A = 4\*coscube A – 3\*cos A

tan 3A = (3\*tan A-tancube A)/(1-3\*tansq A)

sin 4A = 4\*sin A\*cos A – 8\*sincube A\*cos A

cos 4A = 8\*cos4 A – 8\*cossq A + 1

[r\*(cost+i\*sint)]p = rp\*(cos pt+i\*sin pt)

**a**cos**x** + **b**sin**x = c,** x = 2nπ + α ± β, where

cosα = a / (sqrt(a^2+b^2)), cosβ = c / (sqrt(a^2+b^2));

2sinAcosB = sin(A+B) + sin(A-B)

2cosAsinB = sin(A+B) - sin(A-B)

2cosAcosB = cos(A-B) + cos(A+B)

2sinAsinB = cos(A-B) – cos(A+B)

sinC + sinD = 2sin[(C+D)/2]cos[(C-D)/2]

sinC - sinD = 2cos[(C+D)/2]sin[(C-D)/2]

cosD + cosC = 2cos[(C+D)/2]cos[(C-D)/2]

cosD - cosC = 2sin[(C+D)/2]sin[(C-D)/2]

# Misc Integration Formula:

a^x => a^x/ln(a)

1/sqrt(x\*x+a\*a) => ln(x+sqrt(x\*x+a\*a))

1/sqrt(x\*x-a\*a) => ln(x+sqrt(x\*x-a\*a))

1/(x\*sqrt(x\*x+a\*a) => -(1/a)\*ln([a+sqrt(x\*x+a\*a)]/x)

1/(x\*sqrt(a\*a-x\*x) => -(1/a)\*ln([a+sqrt(a\*a-x\*x)]/x)

# Misc Differentiation Formula:

asin x => 1/sqrt(1-x\*x) acos x => -1/sqrt(1-x\*x)

atan x => 1/(1+x\*x) acot x => -1/(1+x\*x)

asec x => 1/[x\*sqrt(x\*x-1)] acosec x => -1/[x\*sqrt(x\*x-1)]

a^x => a^x\*ln(x) cot x => -cosecsq x

sec x => sec x \* tan x cosec x => -cosec x \* cot x

# Mirror point(mx,my) of a point(x,y) w.r.to a line(ax+by+c=0):

void mirrorPoint(double a,double b,double c,double x,double y,double &mx,double &my) {

mx = - x\*(a\*a-b\*b) - 2.0\*a\*b\*y - 2.0\*a\*c; mx /= (a\*a+b\*b);

my = y\*(a\*a-b\*b) - 2.0\*a\*b\*x - 2.0\*b\*c; my /= (a\*a+b\*b);

}

# Circum Circle:

R = abc / (4\*area)

//measuring the Circum\_center M(x,y):

k1 = A.x\*A.x - B.x\*B.x + A.y\*A.y - B.y\*B.y;

k2 = A.x\*A.x - C.x\*C.x + A.y\*A.y - C.y\*C.y;

k3 = (A.x\*C.y + B.x\*A.y + C.x\*B.y) - (C.x\*A.y + A.x\*B.y + B.x\*C.y);

M.x = (k2\*(A.y-B.y) - k1\*(A.y-C.y))/(2.\*k3);

M.y = (k1\*(A.x-C.x) - k2\*(A.x-B.x))/(2.\*k3);

# In Circle:

// The triangle consists of points A, B and C

r = area / s

I.x = (A.x\*a + B.x\*b + C.x \* c) / (a+b+c)

I.y = (A.y\*a + B.y\*b + C.y \* c) / (a+b+c)

# Great circle Distance Between 2 points given in Longitude/latitude format [Radius = R]

haversine(x) = ( 1 – cos(x) )/2.0;

a = haversine(lat2 - lat1)

b = cos(lat1) \* cos(lat2) \* haversine(lon2 - lon1)

c = 2 \* atan2(sqrt(a + b), sqrt(1 - a - b))

d = R \* c

# Determining if a point lies on the interior of a 3D convex polygon:

// To determine whether a point is on the interior of a convex polygon in 3D, one

// might be tempted to first determine whether the point is on the plane, then

// determine its interior status. Both of these can be accomplished at once by

// computing the sum of the angles between the test point (q below) and every pair of

// edge points p[i]->p[i+1]. This sum will only be twopi if both the point is on the

// plane of the polygon AND on the interior. The angle sum will tend to 0 the further

// away from the polygon point q becomes. The following code snippet returns the angle

// sum between the test point q and all the vertex pairs. The angle sum is in radians.

#define EPSILON 0.0000001

#define MODULUS(p) (sqrt(p.x\*p.x + p.y\*p.y + p.z\*p.z))

const double TWOPI = 6.283185307179586476925287, RTOD = 57.2957795;

double CalcAngleSum( point3D q, point3D \*p, int n ) {

double m1,m2,anglesum=0,costheta;

point3D p1, p2;

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

p1.x = p[i].x - q.x; p1.y = p[i].y - q.y; p1.z = p[i].z - q.z;

p2.x = p[(i+1)%n].x - q.x;

p2.y = p[(i+1)%n].y - q.y;

p2.z = p[(i+1)%n].z - q.z;

m1 = MODULUS(p1), m2 = MODULUS(p2);

if(m1\*m2 <= EPSILON) return(TWOPI); // We are on a node, consider this inside

else costheta = (p1.x\*p2.x + p1.y\*p2.y + p1.z\*p2.z) / (m1\*m2);

anglesum += acos(costheta);

}

return(anglesum);

}

# Misc Geometry:

const double eps = 1e-11, pi = 2 \* acos( 0.0 );

struct point { // Creates normal 2D point

double x, y;

point() {}

point( double xx, double yy ) { x = xx, y = yy; }

};

struct point3D { // Creates normal 3D point

double x, y, z;

};

struct line { // Creates a line with equation ax + by + c = 0

double a, b, c;

line() {}

line( point p1,point p2 ) {

a = p1.y - p2.y;

b = p2.x - p1.x;

c = p1.x \* p2.y - p2.x \* p1.y;

}

};

struct circle { // Creates a circle with point 'center' as center and r as radius

point center;

double r;

circle() {}

circle( point P, double rr ) { center = P; r = rr; }

};

struct segment { // Creates a segment with two end points -> A, B

point A, B;

segment() {}

segment( point P1, point P2 ) { A = P1, B = P2; }

};

inline bool eq(double a, double b) { return fabs( a - b ) < eps; } //two numbers are equal

# Distance - Point, Point:

inline double Distance( point a, point b ) {

return sqrt( ( a.x - b.x ) \* ( a.x - b.x ) + ( a.y - b.y ) \* ( a.y - b.y ) );

}

# Distance^2 - Point, Point:

inline double sq\_Distance( point a, point b ) {

return ( a.x - b.x ) \* ( a.x - b.x ) + ( a.y - b.y ) \* ( a.y - b.y );

}

# Distance - Point, Line:

inline double Distance( point P, line L ) {

return fabs( L.a \* P.x + L.b \* P.y + L.c ) / sqrt( L.a \* L.a + L.b \* L.b );

}

# Is left Function:

inline double isleft( point p0, point p1, point p2 ) {

return( ( p1.x - p0.x ) \* ( p2.y - p0.y ) - ( p2.x - p0.x ) \* ( p1.y - p0.y ) );

}

# Intersection - Line, Line:

inline bool intersection( line L1, line L2, point &p ) {

double det = L1.a \* L2.b - L1.b \* L2.a;

if( eq ( det, 0 ) ) return false;

p.x = ( L1.b \* L2.c - L2.b \* L1.c ) / det;

p.y = ( L1.c \* L2.a - L2.c \* L1.a ) / det;

return true;

}

# Intersection - Segment, Segment:

inline bool intersection( segment L1, segment L2, point &p ) {

if( !intersection( line( L1.A, L1.B ), line( L2.A, L2.B ), p) ) {

return false; // can lie on another, just check their equations, and check overlap

}

return(eq(Distance(L1.A,p)+Distance(L1.B,p),Distance(L1.A,L1.B)) &&

eq(Distance(L2.A,p)+Distance(L2.B,p),Distance(L2.A,L2.B)));

}

# Perpendicular Line of a Given Line Through a Point:

inline line findPerpendicularLine( line L, point P ) {

line res; //line perpendicular to L, and intersects with P

res.a = L.b, res.b = -L.a;

res.c = -res.a \* P.x - res.b \* P.y;

return res;

}

# Distance - Point, Segment:

inline double Distance( point P, segment S ) {

line L1 = line(S.A,S.B), L2; point P1;

L2 = findPerpendicularLine( L1, P );

if( intersection( L1, L2, P1 ) )

if( eq ( Distance( S.A, P1 ) + Distance( S.B, P1 ), Distance( S.A, S.B ) ) )

return Distance(P,L1);

return min ( Distance( S.A, P), Distance( S.B, P) );

}

# Area of a 2D Polygon:

double areaPolygon( point P[], int n ) {

double area = 0;

for( int i = 0, j = n - 1; i < n; j = i++ ) area += P[j].x \* P[i].y - P[j].y \* P[i].x;

return fabs(area)/2;

}

# Point Inside Polygon:

bool insidePoly( point &p, point P[], int n ) {

bool inside = false;

for( int i = 0, j = n - 1; i < n; j = i++ )

if( (( P[i].x < p.x ) ^ ( P[j].x < p.x )) &&

(P[i].y - P[j].y) \* abs(p.x - P[j].x) < (p.y - P[j].y) \* abs(P[i].x - P[j].x) )

inside = !inside;

return inside;

}

# Intersection - Circle, Line:

inline bool intersection(circle C,line L,point &p1,point &p2) {

if( Distance( C.center, L ) > C.r + eps ) return false;

double a, b, c, d, x = C.center.x, y = C.center.y;

d = C.r\*C.r - x\*x - y\*y;

if( eq( L.a, 0) ) {

p1.y = p2.y = -L.c / L.b;

a = 1;

b = 2 \* x;

c = p1.y \* p1.y - 2 \* p1.y \* y - d;

d = b \* b - 4 \* a \* c;

d = sqrt( fabs (d) );

p1.x = ( b + d ) / ( 2 \* a );

p2.x = ( b - d ) / ( 2 \* a );

}

else {

a = L.a \*L.a + L.b \* L.b;

b = 2 \* ( L.a \* L.a \* y - L.b \* L.c - L.a \* L.b \* x);

c = L.c \* L.c + 2 \* L.a \* L.c \* x - L.a \* L.a \* d;

d = b \* b - 4 \* a \* c;

d = sqrt( fabs(d) );

p1.y = ( b + d ) / ( 2 \* a );

p2.y = ( b - d ) / ( 2 \* a );

p1.x = ( -L.b \* p1.y -L.c ) / L.a;

p2.x = ( -L.b \* p2.y -L.c ) / L.a;

}

return true;

}

# Find Points that are r1 unit away from A, and r2 unit away from B:

inline bool findpointAr1Br2(point A,double r1,point B, double r2,point &p1,point &p2) {

line L;

circle C;

L.a = 2 \* (B.x - A.x );

L.b = 2 \* (B.y - A.y );

L.c = A.x \* A.x + A.y \* A.y - B.x \* B.x - B.y \* B.y + r2 \* r2 - r1 \* r1;

C.center = A;

C.r = r1;

return intersection( C, L, p1, p2 );

}

# Intersection Area between Two Circles:

inline double intersectionArea2C( circle C1, circle C2 ) {

C2.center.x = Distance( C1.center, C2.center );

C1.center.x = C1.center.y = C2.center.y = 0;

if( C1.r < C2.center.x - C2.r + eps ) return 0;

if( -C1.r + eps > C2.center.x - C2.r ) return pi \* C1.r \* C1.r;

if( C1.r + eps > C2.center.x + C2.r ) return pi \* C2.r \* C2.r;

double c, CAD, CBD, res;

c = C2.center.x;

CAD = 2 \* acos( (C1.r \* C1.r + c \* c - C2.r \* C2.r) / (2 \* C1.r \* c) );

CBD = 2 \* acos( (C2.r \* C2.r + c \* c - C1.r \* C1.r) / (2 \* C2.r \* c) );

res=C1.r \* C1.r \* ( CAD - sin( CAD ) ) + C2.r \* C2.r \* ( CBD - sin ( CBD ) );

return .5 \* res;

}

# Circle Through Thee Points:

circle CircleThrough3points( point A, point B, point C) {

double den; circle c;

den = 2.0 \*((B.x-A.x)\*(C.y-A.y) - (B.y-A.y)\*(C.x-A.x));

c.center.x =( (C.y-A.y)\*(B.x\*B.x+B.y\*B.y-A.x\*A.x-A.y\*A.y) –

(B.y-A.y)\*(C.x\*C.x+C.y\*C.y-A.x\*A.x-A.y\*A.y) );

c.center.x /= den;

c.center.y =( (B.x-A.x)\*(C.x\*C.x+C.y\*C.y-A.x\*A.x-A.y\*A.y) –

(C.x-A.x)\*(B.x\*B.x+B.y\*B.y-A.x\*A.x-A.y\*A.y) );

c.center.y /= den;

c.r = Distance( c.center, A );

return c;

}

# Rotating a Point anticlockwise by 'theta' radian w.r.t Origin:

inline point rotate2D( double theta, point P ) {

point Q;

Q.x = P.x \* cos( theta ) - P.y \* sin( theta );

Q.y = P.x \* sin( theta ) + P.y \* cos( theta );

return Q;

}