Task Distance Calculations

Version 0.1.0, 2019-10-22

CONTENTS

PREFACE

This is a draft document intended for feedback.

It is based on the task distance calculations in FS, with the following differences:

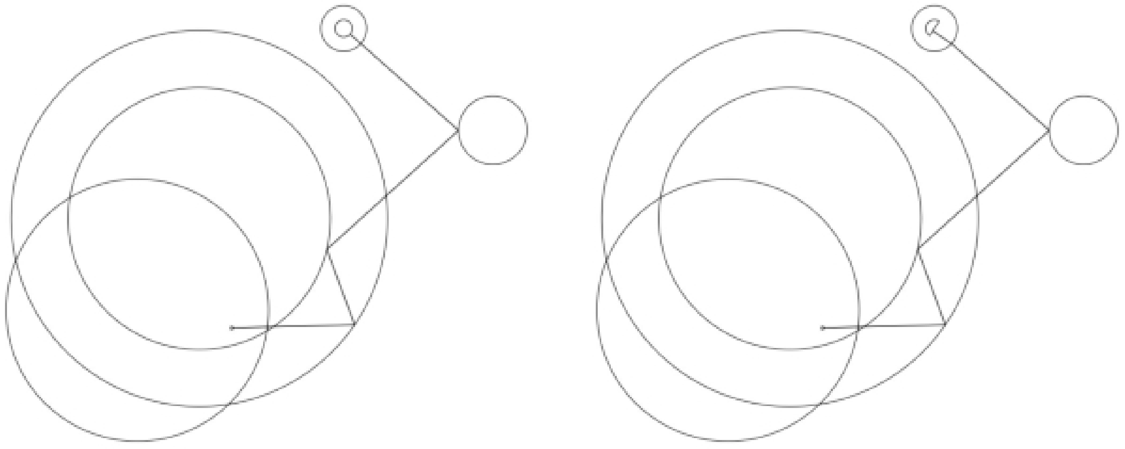
- the code is split into simple functions
- it computes the maximum number of operations, rather than using a fixed value (200)
- it ignores the radius of the first point
- it adds extra error-checking in case cylinders erroneously intersect
- it calculates the shortest path to the end of speed section and then on to goal

What this document does not yet address is:

- the transformation method from WGS84 to cartesian coordinates
- the method of transforming cartesian fix positions back to WGS84 coordinates

1. INTRODUCTION

The task distance is defined as the path of shortest distance that touches all turnpoint cylinders, from the launch point to the end of speed section and then on to goal. If the end of speed section is also the goal (or there is no speed section) then the path is measured from launch point to goal.



Shortest path to a goal cylinder (left) and a goal line (right). The effect of measuring the shortest path to the end of the speed section can be seen by the kinked leg to the goal line.

The calculations required to determine this path are best suited to plane geometry, so the WGS84 coordinates of the turnpoint cylinders (and goal line endpoints) are transformed to cartesian coordinates. The planar distance of the calculated path is not used, but the resulting position fixes on the cylinders (and goal line) are transformed back to WGS84 coordinates and used to calculate the ellipsoidal distance.

This document describes the algorithms needed to perform these calculations, which are as much about speed and ease of use as they are about acceptable accuracy.

2. CALCULATIONS

These calculations require a set of points representing the cylinders, comprising a center position in cartesian coordinates, a radius in metres and a fix position that will mark the point on the cylinder (or line) of the shortest path. This fix position is initially set as the center position.

When the calculations are used to determine the best distance flown by a pilot, the set of points is created from a track point and the remaining cylinders not reached by the pilot.

2.1. Core algorithm

The base calculation uses three sequential points (P1, P2, P3) and finds the shortest distance between P1 and P3 that touches the P2 cylinder. P1 and P3 are taken from their fix positions and the point found on the cylinder is stored as the P2 fix. The leg distance is measured between the P1 and P2 fix positions.

The algorithm sequentially steps through the remaining points and performs this calculation on each one, so the most recent P2 fix becomes the P1 fix used in the next step. On completion, all points will have a fix position on their cylinder (or line) and the shortest distance found will be the sum of the leg distances.

This algorithm is then repeated, using the points from the previous operation, until the difference between the current shortest distance and the last shortest distance is either less than a certain tolerance, or the maximum number of operations has been reached.

2.2. Required settings

In order that implementations give the same results, it is necessary to define some basic settings.

Tolerance

The value of the tolerance affects the final task distance. It also affects the number of operations required, which itself is dependant on the complexity of the task.

A tolerance of 1 metre is used so that the calculations run quickly. A smaller value may shorten the planar distance by a few metres for a simple configuration of cylinders (perhaps more when the task travels between inner and outer cylinders). But this will not always be reflected in the final ellipsoidal distance and will require many more operations iterating over each cylinder.

Maximum number of operations

A maximum number of operations is required in case a distance larger than the previous one is found, which may lead to an infinite loop. This could happen in complex cylinder configurations or if a cylinder erroneously intersects another. The maximum number of operations is the number of points multiplied by 10.

Start index and order of evaluation

The first point used in the evaluation must be at index 1, because the distance is measured from the center of the launch waypoint (regardless of whether it has been given a radius).

The order in which the points are evalutated is also important. While the number of operations may be reduced by evaluating odd then even indexed points, or other such tricks, the results cannot be consistently replicated across difference cylinder configurations. Evaluation order must be sequentially upwards.

Summary of required settings:

- The tolerance value is 1 metre.
- The maximum number of operations is the number of points * 10.
- Points are evaluated seqentially upwards from index 1.

3. EXAMPLES

The pseudo code used to convey the intention and flow of the calculations is written in a C-like style and presented as a series of simple functions.

3.1. Point object

In these examples the object, or struct, representing a cylinder or line is denoted as a `point`. It has the following properties, or members:

x	(float) the x coordinate
y	(float) the y coordinate
radius	(int) the radius in metres
fx	(float) the x coordinate of the fix
fy	(float) the y coordinate of the fix

On construction, or when creating a new instance, the `fx` and `fy` properties must be initialized to the `x` and `y` values.

```
function createPoint(x, y, radius = 0)
{
  // Using ECMAScript object literals to convey object creation
  return { x: x, y: y, radius: radius, fx: x, fy: y };
}

function createPointFromCenter(point)
{
  return createPoint(point,x, point.y, point.radius)
}

function createPointFromFix(point)
{
  return createPoint(point.fx, point.fy, point.radius)
}
```

3.2. Planar calculations

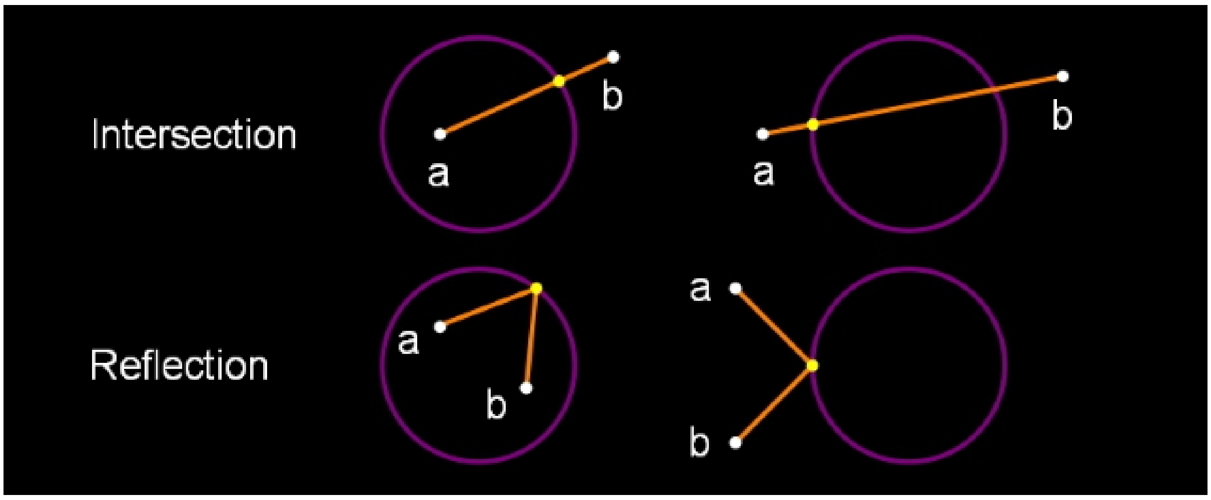
The entry point is the getShortestPath function, which initiates and keeps track of the distances found and the number of operations.

The main function is optimizePath, which is responsible for stepping through the set of points and calculating the shortest distance. It uses the following helper functions:

- getTargetPoints
- processCylinder
- processLine

3.2.1. Cylinder evaluation

In general the the solution to finding the shortest path is either one of intersection or reflection, as shown in the image below.



The yellow dot marks the fix on the cylinder between point A and point B.

The functions to solve these problems are:

- setIntersection1
- setIntersection2
- setReflection

Exceptions to the above are when point A and point B are the same, or when one them is on the cylinder. The latter situation can only occur if a cylinder erroneously intersects another, which may result in a miscalculation or extra operations. These situations are handled by:

- projectOnCircle
- pointOnCircle

3.2.2. getShortestPath

This is the outer controlling function that iterates through a sequence of distance optimizations until the difference between the last one is less than the tolerance or the maximum number of operations has been reached.

```
// Inputs:
// points - array of point objects
// esIndex - index of the ESS point, or -1
// line - goal line endpoints, or empty array

function getShortestPath(points, esIndex, line)
{
  tolerance = 1.0;
  lastDistance = INT_MAX;
  finished = false;
  count = getArrayLength(points);

  // opsCount is the number of operations allowed
  opsCount = count * 10;

  while (!finished && opsCount-- > 0) {
    distance = optimizePath(points, count, esIndex, line);

    // See if the difference between the last distance id
    // smaller than the tolerance
    finished = lastDistance - distance < tolerance;
    lastDistance = distance;
  }
}
```

3.2.3. optimizePath

The algorithm sequentially steps through the cylinder points, taking three consecutive points at each step, with the middle one being the target of the calculation.

Each set of three points is passed to an appropriate function that finds the shortest path between the outer points and the target. The position of the fix on the target cylinder (or goal line) is set as a property of the target point. The target point subsequently becomes the first point in the next iteration step and this fix property is used to denote its position.

```
// Inputs:
// points - array of point objects
// count - number of points
// esIndex - index of the ESS point, or -1
// line - goal line endpoints, or empty array

function optimizePath(points, count, esIndex, line)
{
  distance = 0;
  hasLine = getArrayLength(line) == 2;

  for (index = 1; index < count; index++) {
    // Get the target cylinder c and its preceding and succeeding points
    c, a, b = getTargetPoints(points, count, index, esIndex);

    if (index == count - 1 && hasLine) {
      processLine(line, c, a);
    } else {
      processCylinder(c, a, b);
    }

    // Calculate the distance from A to the C fix point
    legDistance = hypot(a.x - c.fx, a.y - c.fy);
    distance += legDistance;
  }

  return distance;
}
```

3.2.4. getTargetPoints

Returns a set of three consecutive points comprising the target cylinder C, plus its preceding and succeeding points (A and B). The target point C is taken directly from the points array (ie. it is a reference, or copied by reference), while the other two are created as new point objects from either their fix or center positions.

The end of speed section is taken from its center position, so that its fix is pinned to the preceeding points, rather than any subsequent points at a different position.

```
// Inputs:
// points - array of point objects
// count - number of points
// index - index of the target cylinder (from 1 upwards)
// esIndex - index of the ESS point, or -1

function getTargetPoints(points, count, index, esIndex)
{
  // Set point C to the target cylinder
  c = points[index];

  // Create point A using the fix from the previous point
  a = createPointFromFix(points[index - 1]);

  // Create point B using the fix from the next point
  // (use point C center for the lastPoint and esIndex).

  if (index == count - 1 || index == esIndex) {
    b = createPointFromCenter(c);
  } else {
    b = createPointFromFix(points[index + 1]);
  }

  return [c, a, b];
}
```

3.2.5. processCylinder

Sets the fix on the target cylinder C from the fixes on the previous point A and the next point B.

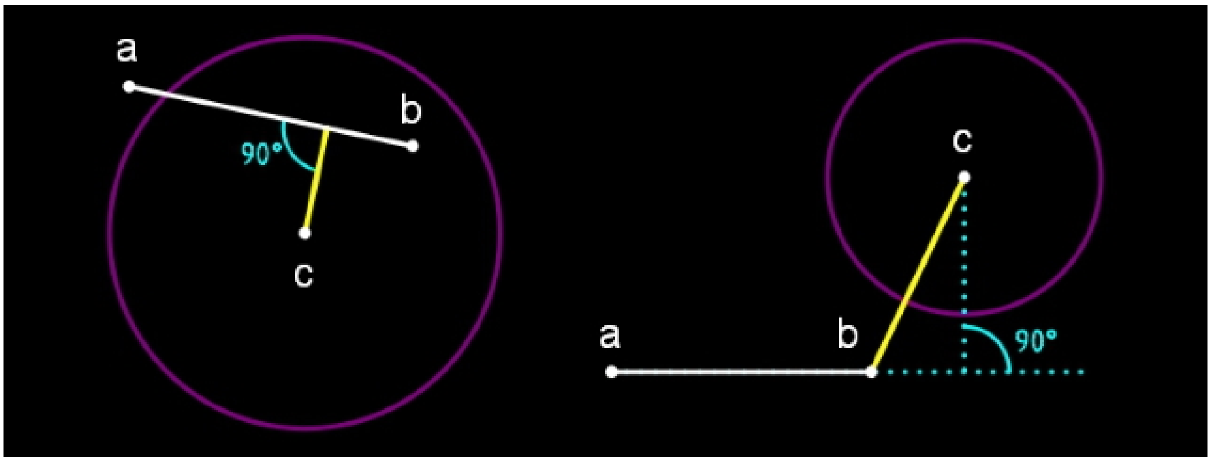
The distance from point C center to the AB line segment determines if it intersects the cylinder (distCtoAB < C radius). If it does and both points are inside the cylinder it requires the reflection solution, otherwise one of the intersection solutions. If the line does not intersect the cylinder or is tangent to it, it requires the reflection solution.

See Cylinder evaluation for more information.

```
// Inputs:
// c, a, b - target cylinder, previous point, next point

function processCylinder(c, a, b)
{
  distAC, distBC, distAB, distCtoAB = getRelativeDistances(c, a, b);

  if (distAB == 0.0) {
    // A and B are the same point: project the point on the circle
    projectOnCircle(c, a.x, a.y, distAC);

  } else if (pointOnCircle(c, a, b, distAC, distBC, distAB, distCtoAB) {
    // A or B are on the circle: the fix has been calculated
    return;

  } else if (distCtoAB < c.radius) {
    // AB segment intersects the circle, but is not tangent to it

    if (distAC < c.radius && distBC < c.radius) {
      // A and B are inside the circle
      setReflection(c, a, b);

    } else if (distAC < c.radius && distBC > c.radius ||
      (distAC > c.radius && distBC < c.radius)) {
      // One point inside, one point outside the circle
      setIntersection1(c, a, b, distAB);

    } else if (distAC > c.radius && distBC > c.radius) {
      // A and B are outside the circle
      setIntersection2(c, a, b, distAB);
    }
  } else {
    // A and B are outside the circle and the AB segment is
    // either tangent to it or or does not intersect it
    setReflection(c, a, b);
  }
}
```

3.2.6. getRelativeDistances

Returns the distances between points A, B and C, plus the distance from C to the AB line segment.

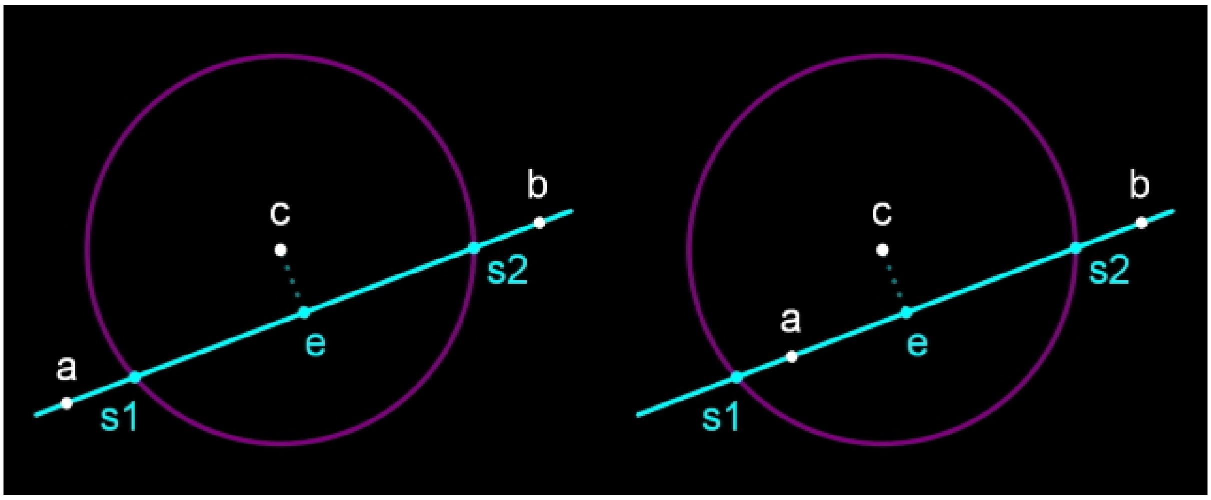


The distance from C to the AB line segment is shown by the yellow line.

```
// Inputs:
// c, a, b - target cylinder, previous point, next point

function getRelativeDistances(c, a, b)
{
  // Calculate distances AC, BC and AB
  distAC = hypot(a.x - c.x, a.y - c.y);
  distBC = hypot(b.x - c.x, b.y - c.y);
  len2 = (a.x - b.x) ** 2 + (a.y - b.y) ** 2;
  distAB = sqrt(len2);

  // Find the shortest distance from C to the AB line segment
  if (len2 == 0.0) {
    // A and B are the same point
    distCtoAB = distAC;
  } else {
    t = ((c.x - a.x) * (b.x - a.x) + (c.y - a.y) * (b.y - a.y)) / len2;

    if (t < 0.0) {
      // Beyond the A end of the AB segment
      distCtoAB = distAC;
    } else if (t > 1.0) {
      // Beyond the B end of the AB segment
      distCtoAB = distBC;
    } else {
      // On the AB segment
      cpx = t * (b.x - a.x) + a.x;
      cpy = t * (b.y - a.y) + a.y;
      distCtoAB = hypot(cpx - c.x, cpy - c.y);
    }
  }

  return [distAC, distBC, distAB, distCtoAB];
}
```

3.2.7. getIntersectionPoints

Returns the two intersection points (s1, s2) of circle C by the AB line, plus the midpoint (e) of the line between them. The intersection points will either be within the AB segment, beyond point A (s1) or beyond point B (s2).



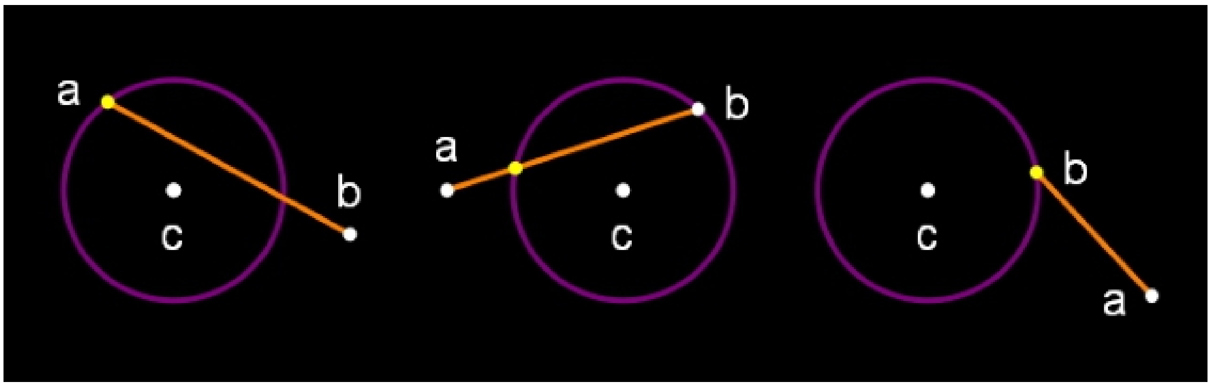
```
// Inputs:
// c, a, b - target cylinder, previous point, next point
// distAB - AB line segment length

function getIntersectionPoints(c, a, b, distAB)
{
  // Find e, which is on the AB line perpendicular to c center
  dx = (b.x - a.x) / distAB;
  dy = (b.y - a.y) / distAB;
  t2 = dx * (c.x - a.x) + dy * (c.y - a.y);

  ex = t2 * dx + a.x;
  ey = t2 * dy + a.y;

  // Calculate the intersection points, s1 and s2
  dt2 = c.radius ** 2 - (ex - c.x) ** 2 - (ey - c.y) ** 2;
  dt = dt2 > 0 ? sqrt(dt2) : 0;

  s1x = (t2 - dt) * dx + a.x;
  s1y = (t2 - dt) * dy + a.y;
  s2x = (t2 + dt) * dx + a.x;
  s2y = (t2 + dt) * dy + a.y;

  return [createPoint(s1x, s1y), createPoint(s2x, s2y), createPoint(ex, ey)];
}
```

3.2.8. pointOnCircle

Sets the C fix position if either point A or point B is on the circle C.

- If point A is on the circle, the fix is taken from point A.
- If point B is on the circle, the fix is either taken from the intersection point closest to point A (if the AB segment intersects the circle and point A is outside it), or from point B.

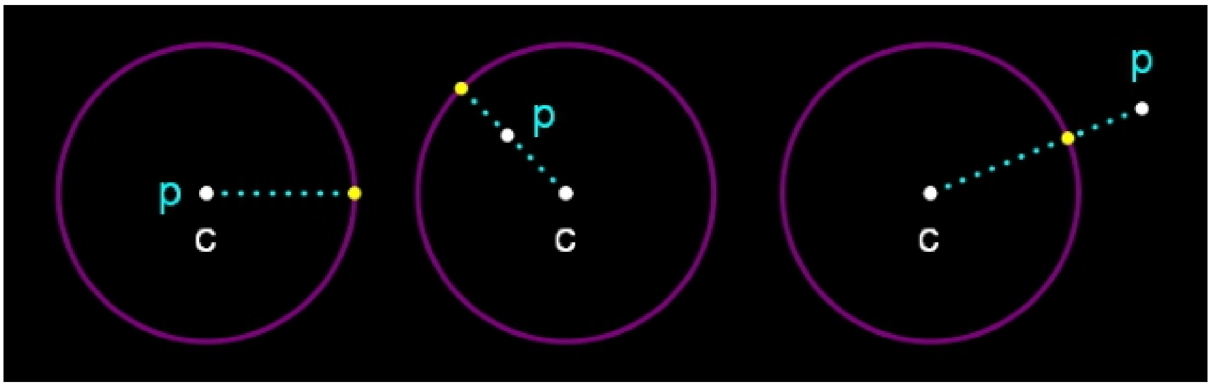
Returns `false` if neither point is on the circle, otherwise `true`.



```
// Inputs:
// c, a, b - target cylinder, previous point, next point
// Distances between the points

function pointOnCircle(c, a, b, distAC, distBC, distAB, distCtoAB)
{
  if (fabs(distAC - c.radius) < 0.0001) {
    // A on the circle (perhaps B as well): use A position
    c.fx = a.x;
    c.fy = a.y;
    return true;
  }

  if (fabs(distBC - c.radius) < 0.0001) {
    // B on the circle

    if (distCtoAB < c.radius && distAC > c.radius) {
      // AB segment intersects the circle and A is outside it
      setIntersection2(c, a, b, distAB);
    } else {
      // Use B position
      c.fx = b.x;
      c.fy = b.y;
    }
    return true;
  }

  return false;
}
```

3.2.9. projectOnCircle

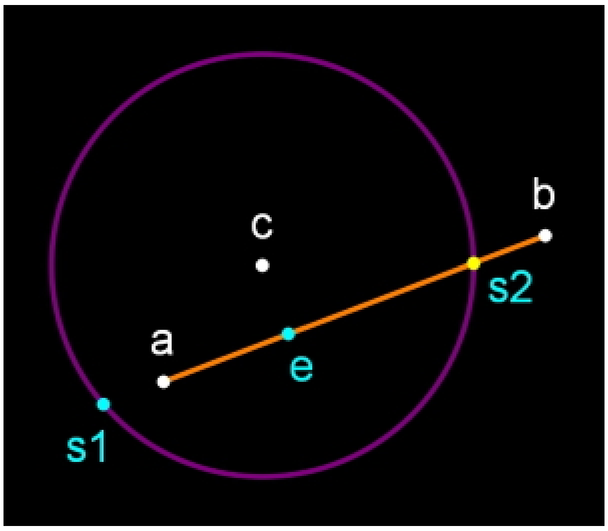
Projects a point (P) on the circle C, at the intersection of the circle by the CP line. The result is stored in the C fix position.



```
// Inputs:
// c - the circle
// x, y - coordinates of the point to project
// len - line segment length, from c to the point

function projectOnCircle(c, x, y, len)
{
  if (len == 0.0) {
    // The default direction is eastwards (90 degrees)
    c.fx = c.radius + c.x;
    c.fy = c.y;
  } else {
    c.fx = c.radius * (x - c.x) / len + c.x;
    c.fy = c.radius * (y - c.y) / len + c.y;
  }
}
```

3.2.10. setIntersection1

Sets the intersection of circle C by the AB line segment when one point is inside and the other is outside the circle (ie. there is only one intersection point on the AB line segment). The result is stored in the C fix position.

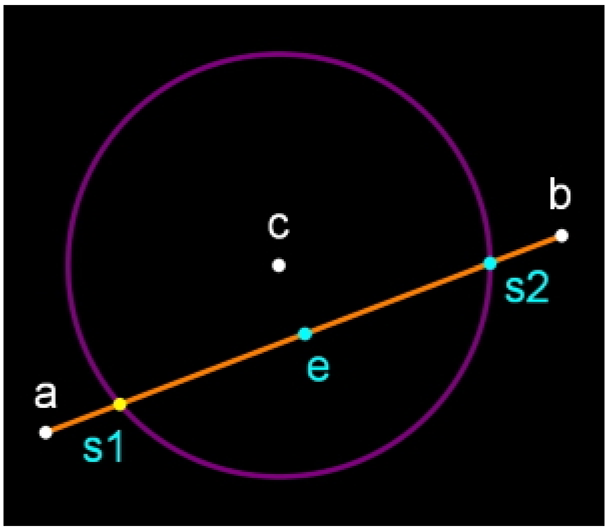


```
// Inputs:
// c, a, b - target cylinder, previous point, next point
// distAB - AB line segment length

function setIntersection1(c, a, b, distAB)
{
  // Get the intersection points (s1, s2)
  s1, s2, e = getIntersectionPoints(c, a, b, distAB);

  as1 = hypot(a.x - s1.x, a.y - s1.y);
  bs1 = hypot(b.x - s1.x, b.y - s1.y);

  // Find the intersection lying between points a and b
  if (fabs(as1 + bs1 - distAB) < 0.0001) {
    c.fx = s1.x;
    c.fy = s1.y;
  } else {
    c.fx = s2.x;
    c.fy = s2.y;
  }
}
```

3.2.11. setIntersection2

Sets the intersection of circle C by the AB line segment when both points are outside the circle (ie. there are two intersection points on the AB line segment).

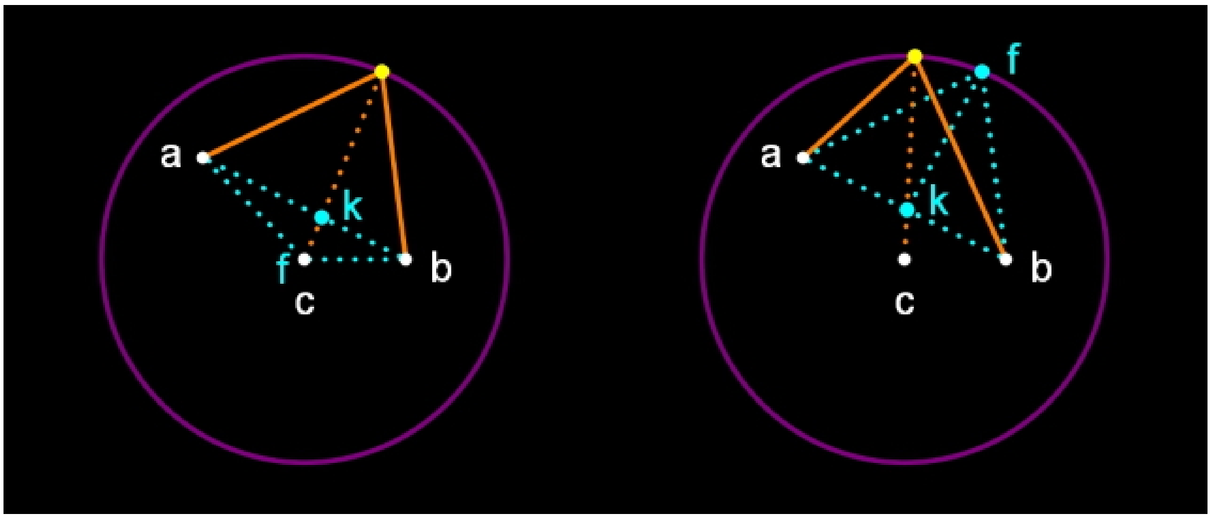
The result is stored in the C fix position and is the intersection that is closest to point A. This will lie between point A and point E (the midpoint of the line between the intersection points).



```
// Inputs:
// c, a, b - target cylinder, previous point, next point
// distAB - AB line segment length

function setIntersection2(c, a, b, distAB)
{
  // Get the intersection points (s1, s2) and midpoint (e)
  s1, s2, e = getIntersectionPoints(c, a, b, distAB);

  as1 = hypot(a.x - s1.x, a.y - s1.y);
  es1 = hypot(e.x - s1.x, e.y - s1.y);
  ae = hypot(a.x - e.x, a.y - e.y);

  // Find the intersection between points a and e
  if (fabs(as1 + es1 - ae) < 0.0001) {
    c.fx = s1.x;
    c.fy = s1.y;
  } else {
    c.fx = s2.x;
    c.fy = s2.y;
  }
}
```

3.2.12. setReflection

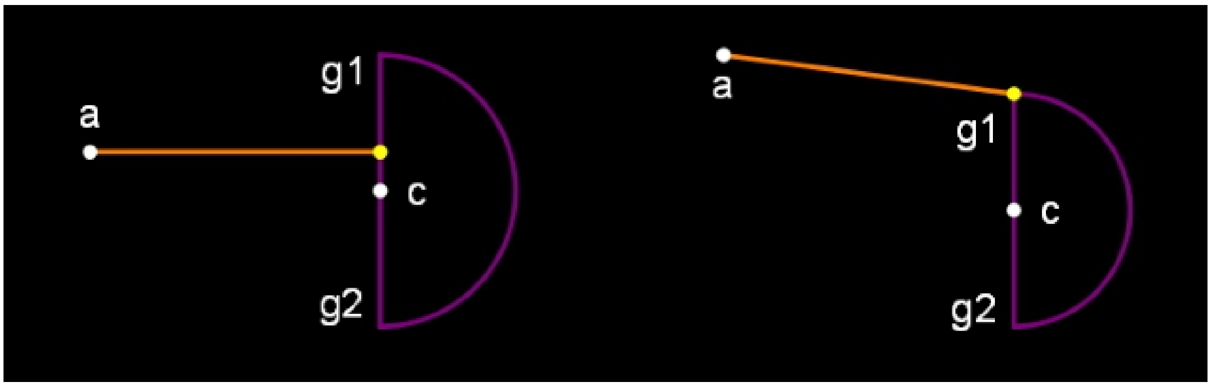
Sets the reflection point of two external or internal points (A and B) on the circle C. This uses the triangle AFB (where F is the current C fix position) to find the point (K) on the AB line segment where it is cut by the AFB angle bisector. The reflected point is found by projecting K on the circle and is stored as the latest C fix position.



```
// Inputs:
// c, a, b - target circle, previous point, next point

function setReflection(c, a, b)
{
  // The lengths of the adjacent triangle sides (af, bf) are
  // proportional to the lengths of the cut AB segments (ak, bk)
  af = hypot(a.x - c.fx, a.y - c.fy);
  bf = hypot(b.x - c.fx, b.y - c.fy);
  t = af / (af + bf);

  // Calculate point k on the AB segment
  kx = t * (b.x - a.x) + a.x;
  ky = t * (b.y - a.y) + a.y;
  kc = hypot(kx - c.x, ky - c.y);

  // Project k on to the radius of c
  projectOnCircle(c, kx, ky, kc);
}
```

3.2.13. processLine

Finds the closest point on the goal line (g1, g2) from point A, storing the result in the C fix position. This will either be on the line itself, or at one of its endpoints.



```
// Inputs:
// line - array of goal line endpoints
// c, a - target (goal), previous point

function processLine(line, c, a)
{
  g1 = line[0], g2 = line[1];
  len2 = (g1.x - g2.x) ** 2 + (g1.y - g2.y) ** 2;

  if (len2 == 0.0) {
    // Error trapping: g1 and g2 are the same point
    c.fx = g1.x;
    c.fy = g1.y;
  } else {
    t = ((a.x - g1.x) * (g2.x - g1.x) + (a.y - g1.y) * (g2.y - g1.y)) / len2;

    if (t < 0.0) {
      // Beyond the g1 end of the line segment
      c.fx = g1.x;
      c.fy = g1.y;
    } else if (t > 1.0) {
      // Beyond the g2 end of the line segment
      c.fx = g2.x;
      c.fy = g2.y;
    } else {
      // Projection falls on the line segment
      c.fx = t * (g2.x - g1.x) + g1.x;
      c.fy = t * (g2.y - g1.y) + g1.y;
    }
  }
}
```