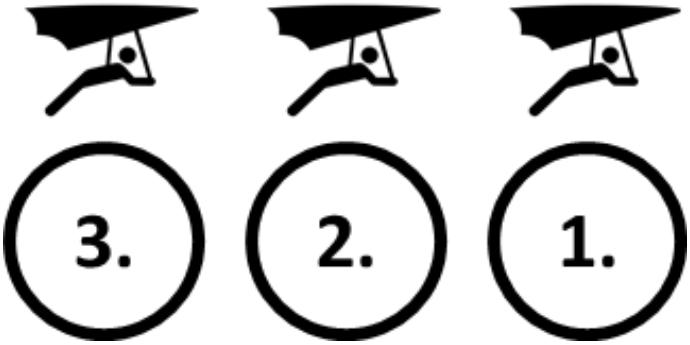
Time-based Scoring

The fair and transparent scoring formula for paragliding and hang-gliding races



Specification V2.1

Author: Jörg Ewald

Date: June 21, 2020

Editor's note: Hang-gliding and paragliding are sports in which both men and women participate, and are indistinctively considered as "pilots". Throughout this document the words "he", "him" or "his" are intended to apply equally to either sex unless it is specifically stated otherwise.

Contents

1 Introduction

This document contains all definitions required to score centralised cross-country competitions for both hang-gliding and paragliding, using time-based scoring (TBS). Its purpose is to serve as an educational tool for all parties involved in such competitions (organizers, pilots, score keepers, spectators, media), as a reference for the implementation of scoring systems, as well as a basis for future improvements and modifications.

1.1 Feedback

TBS has been tested and proofed on the field already. It is still possible to improve the current setup and we are listening to feedback and positive contributions. We are especially looking for competition organizers to work with us in fine-tuning the scoring. To get started, contact us at **tbs@yourstruly.ch**.

1.2 Changes from previous edition

As TBS is still under development, the contents of this document are expected to change frequently over time. See also the following section. Changes from the previous edition of this document are marked using left and right borders in red.

1.3 Differences between Hang-Gliding and Paragliding

TBS handles a few things differently between hang-gliding and paragliding, mainly due to the speed difference between the two.

Text marked in blue applies exclusively to hang-gliding.

Text marked in orange applies exclusively to paragliding.

1.4 History

Version	Release date	Main changes
2.0	January 22, 2020	First official release of TBS
2.1	June 21, 2020	• Adopt definitions for task distance calculations from GAP • New way to determine minimum distance • Calculate day_quality value, to be used in the WPRS for hang-gliding competitions

2 TBS explained

This section explains what TBS wants to achieve, and how this is done at a high level. The in-depth specification of the calculations behind TBS are given in the remainder of the document.

2.1 The TBS philosophy

Time-based scoring was invented by Maxime Bellemin and Jörg Ewald in response to the increasing complexity and incomprehensibility of the GAP scoring formula that has been in use in free-flying cross-country competitions for nearly two decades. Maxime and Jörg set out to create a scoring system that is:

1. Easy to understand
2. Transparent
3. Suitable for a racing sport

2.2 Guiding principles

These are the three guiding principles of TBS:

1. Performance is measured and expressed in a well-known and comparable unit: time
2. Reaching goal has priority over being fast
3. A pilot's score is only influenced by his own performance, and by better-performing pilots

2.3 Races using TBS

A race that is scored with TBS work the same way as today's races:

1. A task is set, consisting of a launch point, a start cylinder, several turnpoints, an end-of-speed-section cylinder and a goal
2. Times are given at which pilots can launch, start the race, and by when the race stops
3. Pilots fly the task, trying to reach goal as fast as possible

2.3.1 Sprints

In addition, the task committee can define some of the turnpoints from the task to also be sprint goals. Pilots are then scored by the order in which they reach those sprint goals and awarded a fraction of the task winner's time as a bonus, depending on their position at the sprint goal.

2.3.2 Score

At the core, a pilot's score is the time he takes to fly the speed section. This time is reduced by any sprint bonus the pilot may have won along the course. It is also capped at a maximum time that depends on how long the winner took to fly the task.

Pilots who do not reach goal are scored with a base time that depends on the (capped) time of the last pilot in goal (with a minimum that depends on the winner's time), plus an adjustment time that is calculated from the uncovered task distance, and an adjustment speed. This adjustment speed depends on the fastest pilot's speed and the number of pilots in goal: The faster the fastest pilot, and the more

pilots in goal, the higher the adjustment speed, and therefore the bigger the penalty for not reaching goal.

2.3.3 No pilot in goal

A task where no pilot reaches goal is converted into a distance competition: The task is adjusted such that the pilot who covered most of the task distance reaches an ad-hoc goal, by increasing the radius of the first turnpoint not reached by any pilot. Then the task is re-scored, with two slight changes from regular tasks: Sprint results are applied to all pilots' task times, and the base time for the pilots who did not reach the ad-hoc goal is based directly on the furthest pilot's task time, without any capping or applying any minimum.

2.3.4 Stopped tasks

Stopped tasks work nearly the same as in GAP, with the following differences:

1. Bonus distance is only calculated from each pilot's track point at the task stop time.
2. If applying the bonus distance takes a pilot into a sprint goal or into goal, the pilot is scored with a time for that point that depends on the task stop time, their distance from that point, and the fastest pilot's speed.

2.4 Competition score

The competition score of each pilot consist of the sum of the task scores. The pilot with the lowest total time wins the competition.

3 Scoring Process

Scoring follows a nine-step process, as depicted in Figure 1:

1. Setting the competition parameters. This happens once for each competition, at the outset, and must not be changed throughout the competition. See section 6 and 7.
2. Setting a task – this happens typically once per day on flyable days. See section 8.
3. Letting the pilots fly the task. See section 9.
4. Evaluating pilots' track logs for this task, and determining for each pilot the distance flown and, if the end of speed section was reached, in what time this happened. See section 10.
5. Evaluating the task, calculation of several values that will be used for scoring pilots. See section 11.
6. Scoring each pilot's flight, See section 12.
7. Ranking all pilots according to their score for the task results. See section 14.
8. Aggregation of task results for competition scoring and ranking. See section 15.

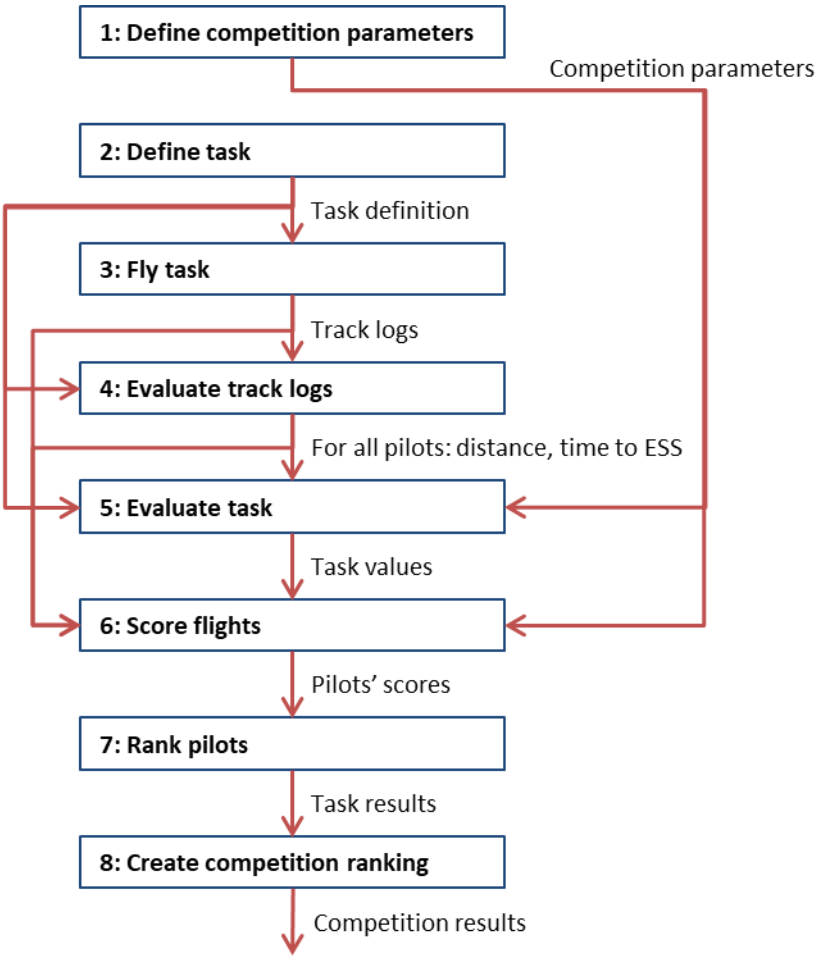


Figure 1: Scoring process

4 Definitions

To a large extent, TBS relies on the same definitions as GAP (the scoring formula currently used by most competitions). See Section 7A, section 5.2.1 of the FAI sporting code. There, the following terms are defined:

- Flight
- Free flight
- Competition task
- Competition flight
- Take-off
- Speed section
- Start of speed section (SSS)
- Turnpoint (TP)
- Control zone
- End of speed section (ESS).
- Goal
- Landing place
- Task distance
- Flown distance
- Finish point
- Race start
- Start time
- Start gate
- Window open time
- Task deadline
- Finish time
- Task time
- Landing time

5 Use of Tracklog Data

5.1 Position

Coordinates of positions, such as turn points or pilot positions, are always given as WGS84 coordinates, based on the WGS84 ellipsoid. The format in which those coordinates are used can be chosen by organisers as appropriate. We recommend DD.ddddd.

5.2 Distance

In general, task evaluation occurs in the x/y plain, therefore distance measurements are always exclusively horizontal measurements. The earth model underlying the distance calculation can be chosen as appropriate. For simplicity's sake, we suggest the FAI sphere, but acknowledge that most tools in use today, both flight recorders and scoring programs, calculate distances based on the WGS84 ellipsoid.

5.2.1 Effect of altitude in distance calculation

For the calculation of the altitude bonus in stopped tasks (13.3.3), altitude is also considered, but this does not affect distance calculations between two geographic points.

5.3 Altitude

All altitude evaluation is primarily based on barometric altitude, as recorded in the flight instrument tracklog (the International Standard Atmosphere pressure altitude QNE) and then when necessary corrected by the scoring software for the pressure conditions of the task (QNH). GNSS altitude may be taken into consideration (from the primary tracklog or a backup log) only in case of problems with barometric logging.

5.4 Time

Time evaluation is based on UTC, as given in GPS tracklogs, with a one-second resolution. For better readability, times of the day may be expressed in local time for the competition location.

6 Competition Parameters

Before the first task, the following parameters must be defined by the meet director, or another person or group as defined by the competition's local regulations:

1. Distance from launch to closest landing
2. Maximum speed

In general, the proposed default values can be used, but as TBS is still being developed, experimenting with different values is encouraged. Let us know what you find out.

6.1 Altitude difference between launch and closest landing

Altitude difference in meters between launch and the closest landing, the designated "bomb-out" landing. This is used to determine Minimum distance (see section 7.2). If several sites will be used for a single competition, either use the biggest altitude, or adjust for each task.

In towing competitions, use the expected release altitude above the launch field.

Default: 800 m

6.2 Maximum task speed

Maximum potential average speed in a task at the competition's site. If several sites will be used for a single competition, with different characteristics (e.g. predominantly one-way tasks vs. closed-circuit tasks), use the speed of the site that gives the highest average speeds.

Default for paragliding: 40 km/h

Default for hang-gliding: 60 km/h

7 Internal parameters

Several internal parameters are implicitly set when a competition is created, some of them based on the competition parameters (see section 6):

1. Nominal launch
2. Minimum distance
3. Nominal goal
4. Maximum adjustment speed
5. Minimum adjustment speed
6. Discards
7. Turnpoint tolerance
8. Score-back time for stopped tasks
9. Bonus glide ratio
10. Sprint bonus factor

7.1 Nominal launch

When pilots do not take off for safety reasons, to avoid difficult launch conditions or bad conditions in the air, Launch Validity is reduced (see section 11.1.1). Nominal Launch defines a threshold as a percentage of the pilots in a competition. Launch Validity is only reduced if fewer pilots than defined by that threshold decide to launch.

In TBS, based on experience with GAP, we use a fixed value of 95% for Nominal Launch.

7.2 Minimum distance

The minimum distance awarded to every pilot who takes off. It is the distance below which it is pointless to measure a pilot's performance. The minimum distance parameter is set so that pilots who are about to "bomb out" will not be tempted to fly into the next field to get past a group of pilots – they all score the same time anyway.

In TBS, based on experience with GAP, minimum distance is based on the altitude difference between launch and the closest landing (6.1), and the bonus glide ratio (7.10). The distance is always rounded to the next higher whole kilometre.

$$minDist = ceiling(\frac{altitudeDifferenceLaunchLanding*2*bonusGlideRatio}{1000})$$

7.3 Nominal goal

Goal percentage of a good task. This affects the goal percentage portion of the adjustment speed (see section 11.2), which in turn is used to calculate the task time for pilots landing short of goal. If the percentage of pilots reaching goal is below nominal goal, then the adjustment speed will be reduced. If the percentage of pilots reaching goal is at or above nominal goal, the goal percentage portion of the adjustment speed will be constant at maximum speed.

Set to 30%

7.4 Maximum adjustment speed

Used to calculate the adjustment speed for a task (see section 11.2), which in turn is used to calculate the task time for pilots landing short of goal. The maximum adjustment speed is calculated from the maximum task speed set for the competition (see section 6.2):

Maximum adjustment speed = Maximum task speed * 1.0

7.5 Minimum adjustment speed

Used to calculate the adjustment speed for a task (see section 11.2), which in turn is used to calculate the task time for pilots landing short of goal. The minimum adjustment speed is calculated from the maximum task speed set for the competition (see section 6.2):

Minimum adjustment speed = Maximum task speed * 0.3

7.6 Maximum goal time factors

Used to calculate the maximum time for which pilots reaching goal will be scored (see section 11.2). It is calculated based on the winner's time and a top and a bottom boundary factor:

MaximumGoalTimeBottomFactor, default is 1.5

MaximumGoalTimeTopFactor, default is 2.5

7.7 Discards

Defines how many tasks need to be flown until a pilot can drop their worst task result. See also section 15.2.

Default: 0 (no discards)

7.8 Turnpoint tolerance

Tolerance applied to distance calculations when determining whether a turnpoint has been reached.

In TBS, based on experience with GAP, we use a fixed value of 0.2%

7.9 Score-back time for stopped tasks

In a stopped task, this value defines the amount of time before the task stop was announced that will not be considered for scoring. The default is 5 minutes, but depending on local meteorological circumstances, it may be set to a longer period for a whole competition. See also section 13.3.1.

In TBS, based on experience with GAP, the score-back time is set to 5 minutes.

7.10 Bonus glide ratio

In a stopped task, pilots still flying at the task stop time (task stop announcement time minus score-back time) are awarded additional distance, based on their altitude over goal. The Bonus Glide Ratio defines the factor by which altitude is converted into distance. See also section 13.3.3.

The same glide ratio is also used to calculate minimum distance from the altitude difference between launch and the closest landing (7.2).

For paragliding: 4.0
For hang-gliding: 5.0

7.11 Sprint bonus factor

Percentage of fastest pilot's time that will be distributed amongst the sprint winners.

Set to 5%

8 Task Setting

8.1 Definition of a task

A task definition consists of:

1. A launch point, given as GPS coordinates
2. A number of turnpoints
3. An indication which of the turnpoint is the start (start of speed section)
4. If goal does not serve as end of speed section: An indication which of the turnpoint is the end of speed section
5. An indication which of the turnpoints serve as sprint goals. Any turnpoint after the start, up to and including the end of speed section, can serve as a sprint goal. The more sprint goals are set in a task, the smaller the available time bonus per sprint goal.
6. A launch time window
7. A start procedure, including timing
8. Optionally, a task deadline

8.2 Definition of a turnpoint

A turnpoint is defined as a cylinder given by:

1. A centre point *c*, given as GPS coordinates
2. A radius *r*, given in meters

A turnpoint cylinder is then given as the cylinder with radius *r* around the axis which cuts the x/y plain orthogonally at the cylinder's centre point *c*. For task evaluation purposes, only the cylinder's projection in the x/y plain is considered: a circle of radius *r* around *c*.

8.3 Start procedures

Competitors are free to launch any time during a given launch window, and to fly about, regardless of any turnpoint or start cylinders, up to the race start. Race start is defined as the crossing of the start cylinder for the last time before continuing to fly through the remainder of the task.

8.3.1 Race to goal

A race to goal start is defined by one or more so-called "start gates". The first – or only – start gate is given as a daytime. Subsequent start gates are given as a time interval, along with the number of start gates.

Example 1: "We have a Race to Goal task, the start gate opens at 13:00"

Example 2: "We have a Race to Goal task with 5 start gates from 13:30 at a 20 minutes interval." – the start gate times in this case are 13:30, 13:50, 14:10, 14:30, and 14.50.

Pilots are free to start any time after the first (or single) start gate. A pilot's start time is then defined as the time of the last start gate after which he started flying the speed section of the task.

Example 3: Given the start gates from Example 2 above, pilot A, crossing the start cylinder at 13:49:01, will be given a start time of 13:30. Any pilot starting after 14:50 will be given a start time of 14:50.

Starting before the first (or only) start gate is considered a failed start. Some refer to this as "jumping the gun". The two disciplines handle failed starts differently, see section 13.2.

8.3.2 Elapsed time

An elapsed time start is defined by a single "start gate", given as a daytime. Pilots are free to start any time after this start gate. A pilot's start time is then defined as the time at which he started flying the speed section of the task. Each pilot has therefore an individual start time.

Example 1: "We have an Elapsed Time task, the start gate opens at 12:30" – pilot A starting at 12:31:03 has a start time of 12:31:03, pilot B starting at 15:48:28 has a start time of 15:48:28.

8.4 Distances

8.4.1 Route optimisation algorithm

At the core of all distance calculations lies an algorithm that calculates the shortest path from a given start position through a series of turnpoint cylinders to a destination (which can be a cylinder, a goal line or a point). The algorithm used to determine this path is defined in Appendix A. The result of this algorithm is a set of points on the cylinders that represent the optimal crossing points for shortest overall distance. Distance is then calculated by iterating through those points, calculating the distance between two subsequent points (leg distance), and building the sum of those leg distances.

$$routeElement = \text{point}(lat, lon) | \text{cylinder}(lat, lon, radius) | \text{goal}(lat, lon, radius)$$
$$routeDefinition = \{routeElement_1, \dots, routeElement_n\}$$
$$path = \{point_1, \dots, point_n\} \text{where} point_x = \text{point}(lat_x, lon_x)$$
$$\text{getShortestPath}(routeDefinition) \text{returns } path$$
$$\text{pathDistance}(path) = \sum_1^{n-1} \text{distance}(path.point_i, path.point_{i+1})$$

8.4.2 Task distances

Task distance is defined as speed section distance plus any distance pilots need to cover before or after the speed section:

$$launch = \text{point}(lat_{launch}, lon_{launch})$$

$$taskPart1 = \left\{ \begin{matrix} launch, \\ preTurnpoint_1, \dots, preTurnpoint_m, \\ startOfSpeedSection, \\ turnpoint_1, \dots, turnpoint_n, \\ endOfSpeedSection \end{matrix} \right\}$$

$$taskPart1Path = \text{getShortestPath}(taskPart1)$$
$$taskPart1Distance = \text{pathDistance}(taskPart1Path)$$
$$preSpeedSectionDistance = \text{pathDistance}(taskPart1Path[0..startOfSpeedSectionIndex])$$
$$\boldsymbol{speedSectionDistance} = taskPart1Distance - preSpeedSectionDistance$$

$$postSpeedSectionDefinition = \begin{Bmatrix} taskPart1Path.point_{last}, \\ postTurnpoint_1, \dots, postTurnpoint_n, \\ goal \end{Bmatrix}$$

$$postSpeedSectionPath = \text{getShortestPath}(postSpeedSectionDefinition)$$

$$postSpeedSectionDistance = \text{pathDistance}(postSpeedSectionPath)$$

$$\boldsymbol{taskDistance} = \begin{pmatrix} preSpeedSectionDistance \\ +speedSectionDistance \\ +postSpeedSectionDistance \end{pmatrix}$$

9 Flying a task

9.1 Re-starting

In tasks that are either set as Race to goal with multiple start gates, or as Elapsed time task, a pilot may decide to return to the start and take a later start time after already having flown a portion of the start.

10 Track evaluation

From each pilot's track, task evaluation determines the distance this pilot flew along the task, the times at which he reached the individual turnpoints, and the time this pilot took to fly the speed section.

10.1 Cylinder radius tolerance

To compensate for the differences in distance calculation between different distance measurement algorithms used in pilots' navigation devices and in scoring programs, a tolerance is applied to all cylinder radius measurements. This had to be introduced so that a pilot reading the distance to the next cylinder centre from his GPS device can rely on having reached the turnpoint when the distance displayed by the instrument is smaller than the defined turnpoint cylinder radius.

The turnpoint tolerance is now set as an internal parameter, see section 7.8.

10.2 Reaching a cylinder

To determine whether a pilot reached a specific cylinder in the task, the pilot's track log must show evidence that:

1. The pilot reached the previous cylinder in the task definition
2. It contains at least one crossing for the cylinder, defined as either of the following:
    a. Existence of a single tracklog point inside the turnpoint cylinder's tolerance band – crossing time is the time at which this tracklog point was recorded
    b. A pair of consecutive tracklog points which lie on opposite sides of the turnpoint cylinder boundary – crossing time is linearly interpolated from the two tracklog points' recording times and their respective distances from the boundary
3. It contains a valid crossing, which occurred later than the valid crossing of the previous cylinder in the task definition, but no earlier than the start time, and no later than the task deadline.

$$
\begin{aligned}
&crossing_{turnpoint_i} : \exists j : (\text{distance}(center_i, trackpoint_j) >= innerRadius_i \\
&\wedge \text{distance}(center_i, trackpoint_j) <= outerRadius_i) \\
&\vee (\text{distance}(center_i, trackpoint_j) > radius_i \wedge \text{distance}(center_i, trackpoint_{j+1}) < radius_i) \\
&\vee (\text{distance}(center_i, trackpoint_j) < radius_i \wedge \text{distance}(center_i, trackpoint_{j+1}) > radius_i)
\end{aligned}
$$

The time of a crossing depends on whether it cuts across the cylinder boundary, or whether it consists of a point within the tolerance band.

$$
\begin{aligned}
&(\text{distance}(center_i, trackpoint_j) >= innerRadius_i \wedge \text{distance}(center_i, trackpoint_j) <= outerRadius_i) : \\
&crossing.time = trackpoint_{j+1}.time
\end{aligned}
$$

$$
\begin{aligned}
&(\text{distance}(center_i, trackpoint_j) < radius_i \wedge \text{distance}(center_i, trackpoint_{j+1}) > radius_i) \\
&\vee (\text{distance}(center_i, trackpoint_j) > radius_i \wedge \text{distance}(center_i, trackpoint_{j+1}) < radius_i) : \\
&crossing.time = \text{interpolateTime}(trackpoint_j, trackpoint_{j+1})
\end{aligned}
$$

The method used to interpolate the crossing time is buried in FS' code and will have to be documented at a later point.

10.2.1 Start time

In Race to Goal tasks with multiple start gates, and in Elapsed Time tasks, the last crossing of the start cylinder that is followed by a crossing of the first turnpoint cylinder (or the landing) is taken as the pilot's start, and that crossing's time is used as the pilot's start time.

10.3 Reaching goal

Verification of a pilot reaching a goal cylinder is achieved by the same method as verification of reaching a turnpoint cylinder (10.2)

10.4 Flown distance

To determine a pilot's flown distance, a first step determines which turnpoints he reached considering all timing restrictions: launch within launch time window, valid start, but only until the task deadline time. After the last turnpoint the pilot reached, for every remaining track point, the shortest distance to goal is calculated using the method described in section **Error! Reference source not found.**. The flown distance is then calculated as task distance minus the shortest distance the pilot still had to fly. Therefore, for scoring, the pilot's best distance along the course line is considered, regardless of where the pilot landed in the end.

$$\forall Pilot_p p \in PilotsLaunched \land p \notin PilotsReachedGoal:$$
$$FlownDistance_p$$
$$= TaskDistance - \min\left(\forall trackpoint_{p,i} \in trackpoints_p : shortestDistanceToGoal\left(trackpoint_{p,i}\right)\right)$$

If a pilot reaches goal, he will be scored for the task distance.

$$\forall Pilot_p : p \in PilotsReachedGoal : FlownDistance_p = TaskDistance$$

10.5 Time for speed section

The time a pilot took to fly the speed section is determined by his start time (which is influenced by the task's start procedure and the time he crossed the start of speed section cylinder) and the time when he crossed the end of speed section after reaching all previous turn points. The smallest unit for time measurement is one second.

Pilots who do not reach goal do not get a time at this stage.

$$\forall p : p \in PilotsReachingESS : time_p = timeAtESS_p - startTime_p$$

SpeedSectionTime(p) = time(p)

11 Task evaluation

11.1 Validity

The task validity is a value between 0 and 1 and measures how suitable a competition task is to evaluate pilots' skills. It is calculated for each task after the task has been flown.

TaskValidity = LaunchValidity

11.1.1 Launch Validity

Launch validity depends on nominal launch and the percentage of pilots present at take-off who launched. If the percentage of pilots on take-off who launch is equal to nominal launch, or higher, then launch validity is 100%. If, for example, only 20% of the pilots present at take-off launch, launch validity goes down to about 17%.

The reasoning behind launch validity: Launch conditions may be dangerous, or otherwise unfavourable. If a significant number of pilots at launch think that the day is not worth the risk of launching, then the gung-ho pilots who did go will not get so many points. This is a safety mechanism.

'Pilots present' are pilots arriving on take-off, with their gear, with the intention of flying. For scoring purposes, 'Pilots present' are all pilots not in the 'Absent' status (ABS): Pilots who took off, plus pilots present who did not fly (DNF). DNFs need to be attributed carefully. A pilot who does not launch due to illness, for instance, is not a DNF, but an ABS.

$$MinLaunchPercentage = 1 - NominalLaunch$$

$$LaunchPercentage = \frac{NumberOfPilotsFlying}{NumberOfPilotsPresent}$$

$$LaunchValidity = Min(1, Max\left(0, \frac{LaunchPercentage}{2 * NominalLaunch - 1} - \frac{1 - NominalLaunch}{NominalLaunch}\right))$$

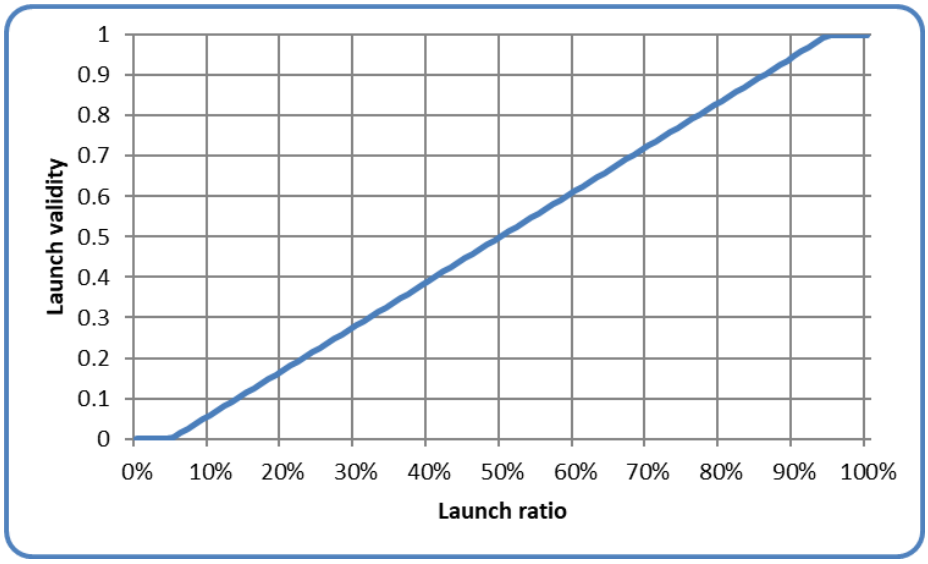


Figure 2: Launch validity curve

11.2 Adjustment speed and adjustment base time

Adjustment speed and adjustment base time are calculated from the pilots' distances and times:

GoalRatio = NumberOfPilotsInGoal / NumberOfPilotsLaunched

WinnerTime = min(for all p: p element of PilotsReachingGoal: SpeedSectionTime(p)) [in seconds]

GoalRatio = 0: WinnerSpeed = 0 km/h
GoalRatio > 0: WinnerSpeed = TaskDistance / WinnerTime [in km/h]

***AdjustmentSpeed** = WinnerAdjustmentSpeed + GoalRatioAdjustmentSpeed*

WinnerAdjustmentSpeed = max(min(MaxAdjSpeed, (MaxAdjSpeed + MinAdjSpeed – WinnerSpeed)), MinAdjSpeed)

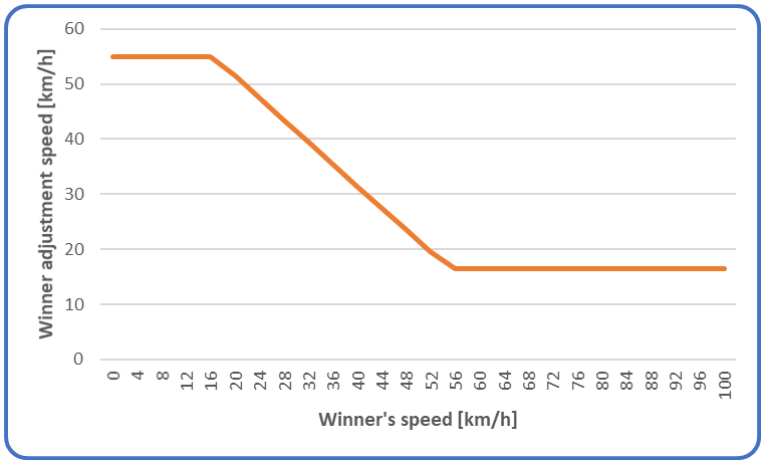


Figure 3: Winner adjustment speed, for maximum task speed = 55 km/h

GoalRatioAdjustmentSpeed = max(MinAdjSpeed, GRAS(GoalRatio))

GRAS(GoalRatio) = MaxAdjSpeed - (GoalRatio(MaxAdjSpeed – MinAdjSpeed)/NominalGoal)*

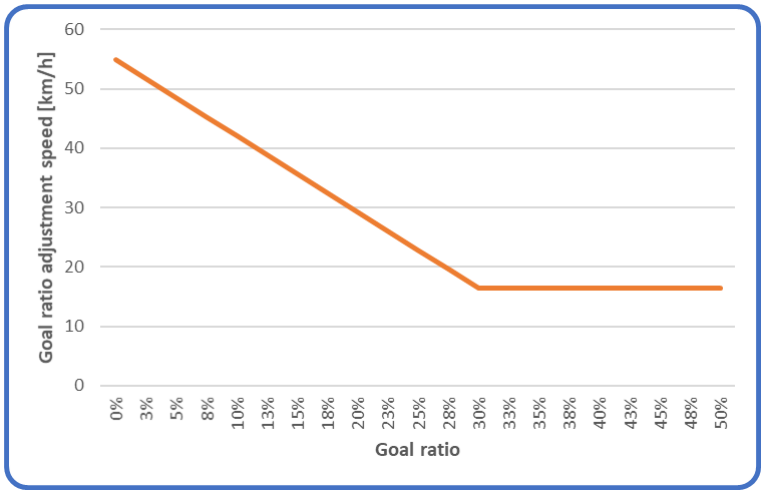


Figure 4: Goal ratio adjustment speed, for maximum task speed = 55 km/h and nominal goal = 30%

AdjustmentBaseTime *= min(MaxGoalTime, LastGoalTime)*

*MaxGoalTime = max(LowerMaximumGoalTimeBoundary, UpperMaximumGoalTimeBoundary - (WinnerTime/3600)/2)*WinnerTime [in seconds]*

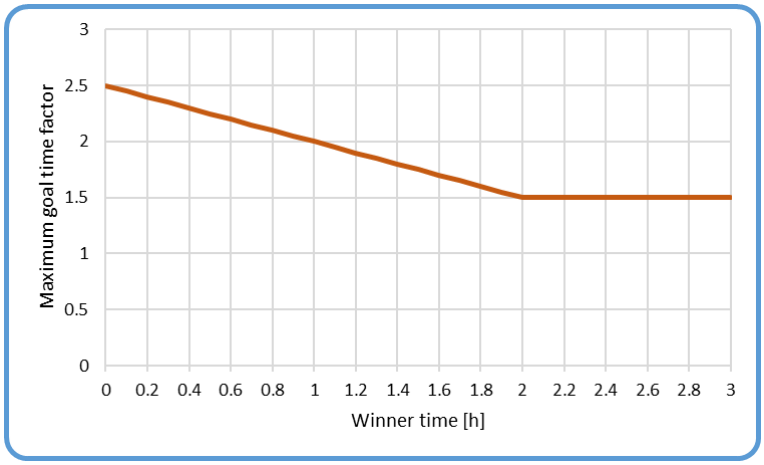


Figure 5: Maximum goal time factor

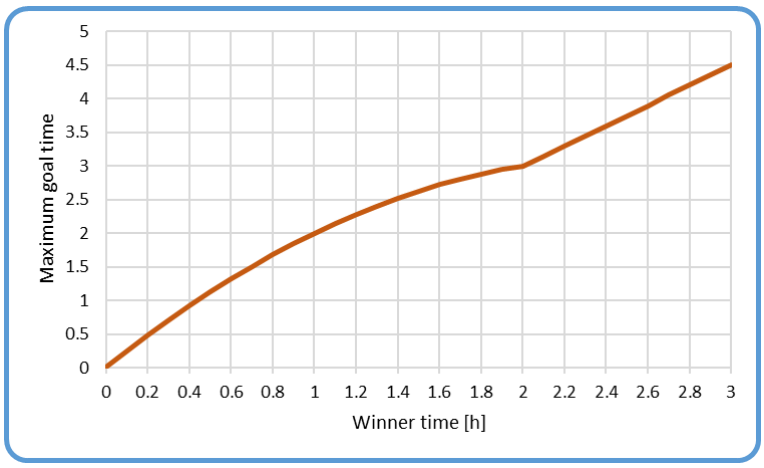


Figure 6: Maximum goal time

LastGoalTime = max(for all p: p element of PilotsReachingGoal: SpeedSectionTime(p)) [in seconds]

What this means:

1. First in goal slow => high winner adjustment speed => smaller penalty for landing out
2. First in goal fast => low winner adjustment speed => bigger penalty for landing out
3. Low goal percentage => high goal percentage adjustment speed => smaller penalty for landing out
4. High goal percentage => low goal percentage adjustment speed => biger penalty for landing out
5. Base time for landing out if some pilots reached goal:
    a. Time of slowest pilot in goal, if this time is less than MaxGoalTime
    b. MaxGoalTime if slowest pilot in goal took longer than MaxGoalTime

12 Pilot score

Each pilot's score is expressed as a time, with a resolution of one second. For pilots who reach goal, this time corresponds with the pilot's time to fly ESS, minus any sprint bonus the pilot won, corrected by the task validity.

For pilots landing short of goal, the time is calculated based on the portion of the task distance they did not cover. If a pilot flies less than minimum distance, he will be scored for this minimum distance. This also applies to pilots who are not able to produce a valid GPS tracklog, but for whom launch officials confirm launch within the launch window.

Pilots who do not launch (DNF) or who are absent from the task (ABS) are scored with flown distance 0.

$$\forall Pilot_p : p \in PilotsReachedGoal : TaskTime_p = \min\left(SpeedSectionTime_p - TotalSprintBonus_p, MaxGoalTime\right) * TaskValidity \text{ [in seconds]}$$

$$\forall Pilot_p p \in PilotsLaunched \wedge p \notin PilotsReachedGoal:$$
$$MissingDistance_p = TaskDistance - \max\left(FlownDistance_p, MinimumDistance\right)$$

$$\forall Pilot_p p \notin PilotsLaunched:$$
$$MissingDistance_p = TaskDistance$$

$$TimeAdjustment_p = \frac{MissingDistance_p}{AdjustmentSpeed}$$
$$SpeedSectionTime_p = AdjustmentBaseTime + TimeAdjustment_p$$
$$TaskTime_p = SpeedSectionTime_p * TaskValidity \text{ [in seconds]}$$

12.1 Sprints

Each turnpoint after the start up to and including ESS is a potential sprint goal. The actual sprint goals must be defined as part of task setting. Pilots are then ranked in the order in which they reach each sprint goal.

The bonus awarded per sprint goal depends on the winner's time and the number of sprint goals:

$$TotalSprintBonus = WinnerTime * SprintFactor \text{ [in seconds]}$$

$$\forall SprintGoal\ s : SprintGoalBonus_s = \frac{TotalSprintBonus}{NumberOfSprintGoals} \text{ [in seconds]}$$

The number of pilots who receive a sprint bonus is determined by the number of pilots flying a task, and the nominal goal parameter:

$$N = PilotsFlying * NominalGoal$$

The sprint bonus for each pilot is calculated based on their rank for that sprint goal:

$$\forall SprintGoal\ s : \forall Pilot\ p \in PilotsReachedSpringGoal_s:$$

$$rank(s,p) \leq \frac{N}{4} : Bonus_{s,p} = \left(1 - \frac{2}{N} * (rank(s,p) - 1)\right) * SprintGoalBonus_s$$

$$\frac{N}{4} < rank(s,p) \leq \frac{N}{2} : Bonus_{s,p} = \left(0.75 - \frac{1}{N} * (rank(s,p) - 1)\right) * SprintGoalBonus_s$$

$$\frac{N}{2} < rank(s,p) \leq \frac{3 * N}{4} : Bonus_{s,p} = \left(0.55 - \frac{0.6}{N} * (rank(s,p) - 1)\right) * SprintGoalBonus_s$$

$$\frac{3 * N}{4} < rank(s,p) : Bonus_{s,p} = \max\left(1, \left(0.4 - \frac{0.4}{N} * (rank(s,p) - 1)\right) * SprintGoalBonus_s\right)$$

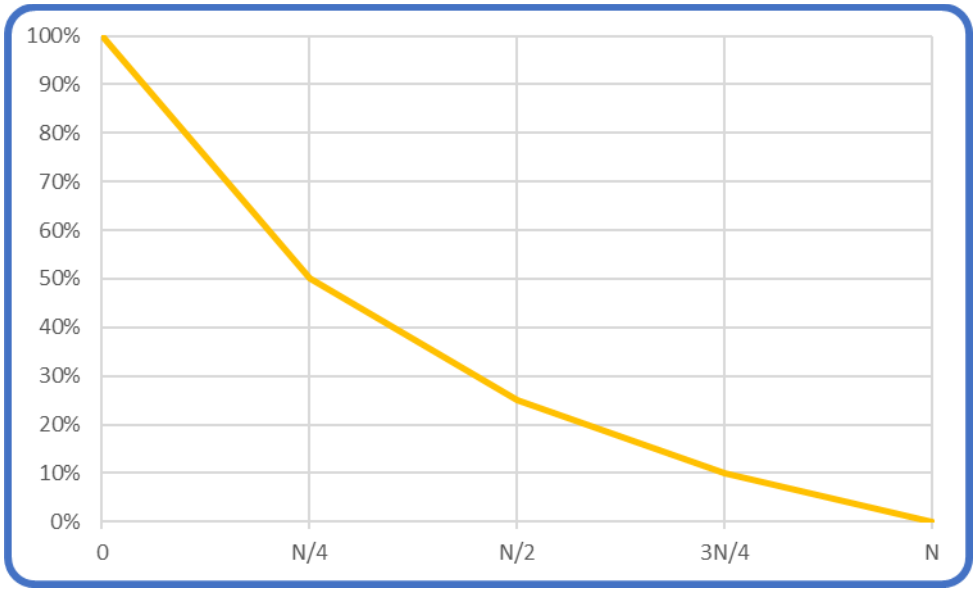


Figure 7: Sprint bonus distribution

A pilot's sprint bonus for a task is the sum of that pilot's bonus times for each sprint goal.

$$TotalSprintBonus_p = \sum_{SprintGoal\ s} Bonus_{s,p} \text{ [in seconds]}$$

13 Special cases

13.1 No pilot in goal

If no pilot reaches goal, an ad-hoc version of the task is created and scored:

1. The first turnpoint not reached by any pilot becomes the ad-hoc goal and implicitly ESS.
2. The ad-hoc goal radius is adjusted so that the pilot who flew the closest to it reaches it.
3. Then this ad-hoc task is scored as described above, with a modified formula for adjustment base time:
   AdjustmentBaseTime = WinnerTime
4. Sprint bonus is applied to all pilots' scores, whether they reached goal or not.

$$\forall Pilot_p : p \in PilotsReachedGoal : TaskTime_p = \left(SpeedSectionTime_p - TotalSprintBonus_p\right) * TaskValidity \text{ [in seconds]}$$

$$\forall Pilot_p p \in PilotsLaunched \wedge p \notin PilotsReachedGoal:$$
$$MissingDistance_p = TaskDistance - \max\left(FlownDistance_p, MinimumDistance\right)$$

$$\forall Pilot_p p \notin PilotsLaunched:$$
$$MissingDistance_p = TaskDistance$$

$$TimeAdjustment_p = \frac{MissingDistance_p}{AdjustmentSpeed}$$
$$SpeedSectionTime_p = AdjustmentBaseTime + TimeAdjustment_p$$
$$TaskTime_p = (SpeedSectionTime_p - TotalSprintBonus_p) * TaskValidity \text{ [in seconds]}$$

13.2 Early start

An early start occurs if a pilot's last SSS cylinder crossing occurred before the first (or only) start gate time.

In paragliding, pilots who perform an early start are only scored for the distance between the launch point and the SSS cylinder, as calculated when determining the complete task distance (see **Error! Reference source not found.**).

In hang-gliding, the so-called "Jump the Gun"-rule applies: If the early start occurred within 5 minutes or less before the first (or only) start gate time, the pilot is scored for his complete flight, but a penalty of 10 minutes is then applied to his total score.
If the early start occurred more than 5 minutes before the first (or only) start gate, then the pilot is only scored for the distance between the launch point and the SSS cylinder, as calculated when determining the task distance (see **Error! Reference source not found.**).
Not yet implemented in TBS for FS!

13.3 Stopped tasks

13.3.1 Stop task time

A task can be stopped at any time by the meet director. The time when a stop was announced for the first time is the "task stop announcement time". This time must be recorded to score the task

appropriately. For scoring purposes, a "task stop" time is calculated. This is the time which determines whether a task will be scored at all. Pilots' flights will only be scored up to this task stop time.

The score-back time is set as part of the competition parameters (see section 0).

$$taskStopTime = taskStopAnnouncementTime - competitionScoreBackTime$$

13.3.2 Scored time window

For stopped Race to Goal tasks with a single start gate, scoring considers the same time window for all pilots: The time between the race start and the task stop time.

$$typeOfTask = RaceToGoal \wedge numberOfStartGates = 1:$$
$$\forall p : p \in StartedPilots : scoreTimeWindow_p = (startTime, taskStopTime)$$

Stopped Race to Goal tasks with multiple start gates, as well as Elapsed Time races, must be treated slightly differently: Only the time window available to the last pilot started is considered for scoring. This time window is defined as the amount of time *t* between the last pilot's start and the task stop time. For all pilots, only this time t after their respective start is considered for scoring.

$$typeOfTask \neq RaceToGoal \vee numberOfStartGates > 1:$$
$$scoreTime = taskStopTime - \max(\forall p : p \in StartedPilots : startTime_p)$$
$$\forall p : p \in StartedPilots : scoreTimeWindow_p = (startTime_p, startTime_p + scoreTime)$$

This means that if the last pilot started and then flew for, for example, 75 minutes until the task was stopped, all tracks are only scored for the first 75 minutes each pilot flew after taking their respective start.

13.3.3 Altitude bonus

To compensate for altitude differences at the time when a task is stopped, a bonus distance is calculated for each pilot's position at task stop time, based on that point's altitude above goal. This bonus distance is added to the distance achieved at that point.

All altitude values used for this calculation are GPS altitude values, as received from the pilots' GPS devices (no compensation for different earth models applied by those devices). For all distance calculations, these new stopped distance values are being used to determine the pilots' missing distances.

$$\forall Pilot_p p \in PilotsLaunched \wedge p \notin PilotsReachedGoal:$$

$$StopDistance_p = TaskDistance - \begin{pmatrix} shortestDistanceToGoal(trackpoint_{p,TaskStopTime}) \\ +BonusGlideRatio * (trackpoint_{p,TaskStopTime}.Altitude - Goal.Altitude) \end{pmatrix}$$

$$MissingDistance_p = TaskDistance - \max(StopDistance_p, FlownDistance_p, MinimumDistance)$$

13.3.4 Goal and sprint goals

Application of the altitude bonus may make pilots reach sprint goals or goal who had not yet reached those turnpoints at task stop time. These pilots will be scored as having reached those turnpoints, with a crossing time that is based on the task stop time, their distance from the turnpoints at task stop time, and the winner's speed – or Maximum Task Speed if no pilot was in goal at task stop time.

$$PilotsReachedGoal = \{\ \}: CrossingSpeed = MaximumTaskSpeed$$
$$PilotsReachedGoal\ \neq\ \{\ \}: CrossingSpeed = WinnerSpeed$$
$$\forall Pilot_p p \in PilotsLaunched \wedge p \notin PilotsReachedGoal:$$
$$\forall Turnpoints\ tp \in (TurnpointsReached(StopDistance_p) - TurnpoinsReached(FlownDistance_p):$$
$$CrossingTime_{tp} = TaskStopTime + \frac{Distance(trackpoint_{p,TaskStopTime}, tp)}{CrossingSpeed}$$

Sprint bonus time is applied to the scores of all pilots in goal, and to the scores of pilots still flying at the task stop time.

13.4 Penalties

Penalties for various actions are defined Section 7A. These penalties are either expressed as an absolute number (e.g. "100 points") or as a percentage (e.g. "10% of the pilot's score in the task where he performed the punishable action").

Penalties defined in Section 7A are converted into time penalties as follows:

$$1\ point = 10\ seconds$$
$$1\% = \frac{TaskTime_p}{100}$$

The resulting time penalty is then added to the punished pilot's task time to calculate his final score.

$$FinalTaskTime_p = TaskTime_p + TimePenalty_p$$

These penalties are completely independent of any "Jump the Gun"-Penalty a pilot may have incurred.

The penalty mechanism can also be used to award bonus points to a pilot for some actions like helping a pilot in distress. In that case the penalty must be given as a negative time.

Any rounding up of scores is to be done after the application of penalties. The worst task time a pilot can attain in a task, regardless of any incurred penalties, is equal to the task time awarded to absent pilots. The best task time a pilot can attain in a task, regardless of any awarded bonus, is equal to the winner's task time.

14 Task ranking

The task ranking is created by ranking pilots in ascending order of their task times. If two or more pilots score identical task times, they are ranked in the same place.

14.1 Sprint scoring

In tasks where sprint goals are being set, a task sprint ranking is created by ranking pilots in descending order of their total sprint bonus times. If two or more pilots score identical sprint bonus times, they are ranked in the same place.

15 Competition ranking

Each pilot's competition score consists of the sum of their task times.

$$\forall Pilot\ p \in PilotsInCompetition: CompetitionScore_p = \sum_{Task\ t} TaskTime_{p,t}$$

Pilots are then ranked in ascending order of their competition score. If two or more pilots achieve the same score, they are ranked in the same place.

15.1 Sprints scoring

In competitions where sprint goals are being set, a competition sprint ranking is created:

$$\forall Pilot\ p \in PilotsInCompetition: CompetitionSprintScore_p = \sum_{Task\ t} TotalSprintBonus_{p,t}$$

Pilots are then ranked in descending order of their competition sprint score. If two or more pilots achieve the same score, they are ranked in the same place.

15.2 Discards

Discards are based on the difference in task time between a pilot and the task winner in all the tasks. Discards are never applied to the task taking place on the planned final day of a competition.

If discards are applied, pilots drop those tasks where the difference between their task time and the winner's Task Time is the biggest. Dropping means that for competition scoring, the winner's task time is used instead of the pilot's actual task time.

16 Team scoring

In competitions where teams are scored, the organizers must decide on the following up front:

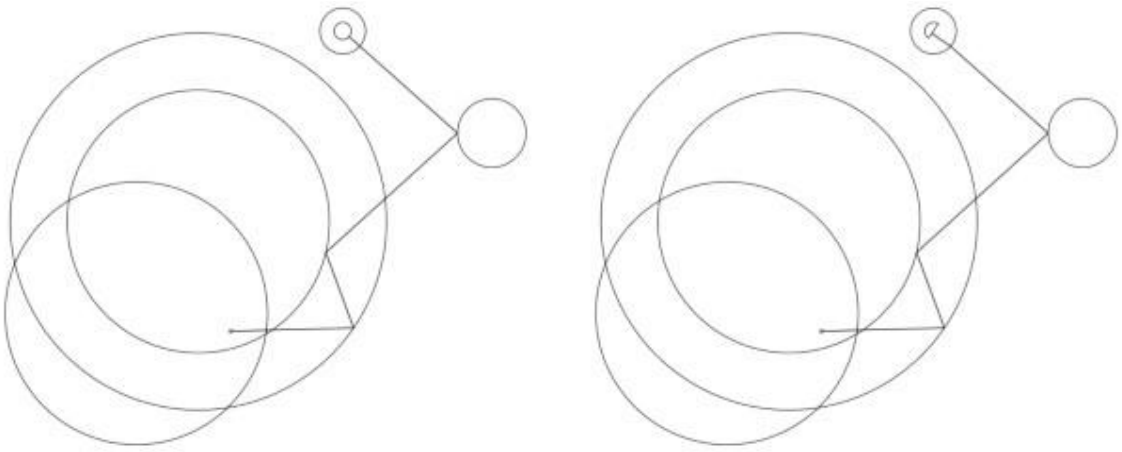
1. Team size (e.g. 4 pilots)
2. Pilot rank within team from which down the scoring team members are selected. This is usually 1 - the best pilot scores for the team - but some competitions choose to disregard the best-placed pilot in each team, and only rank teams according to the second team member.
3. Number of scoring team members (e.g. 2 pilots)

For each task, the task times of the scoring team members of each team are added up. The team's competition score consists of the sum of those team task times. Teams are ranked by the team competition time in ascending order. If two or more teams achieve the same time, they are ranked in the same place.

Appendix A: Algorithm for finding shortest path

1 Introduction

Distance calculation in paragliding and hang-gliding races is based on finding the shortest path between a given start point and a goal (cylinder or goal line), touching any number of intermittent cylinders. This calculation can then be used to determine the distance of a set task, but also for the distance a pilot still must fly to complete a task.



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.

While finding the shortest path seems to be simple for human beings using pen and paper, there exists no closed mathematical solution for this problem. We therefore must resort to an iterative approach that converges on a minimum in most cases.

The calculations required to determine this path are best suited to plane geometry, so all WGS84 coordinates, of starting point, of the turnpoint cylinder centers and goal points, 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.

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

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 distance a pilot still must fly to goal, the set of points is created from the pilot's track point and the remaining cylinders not yet 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. The point found on the P2 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 iteration, 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 iterations has been reached.

2.2 Required settings

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

2.2.1 Tolerance

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

A tolerance of 1 metre is enough, since all other distance measurements are also in meters. A smaller value may shorten the planar distance by a fraction of one metre, but this will not always be reflected in the final ellipsoidal distance and will require many more additional iterations.

2.2.2 Maximum number of iterations

The algorithm does not in all cases converge on the absolute minimum distance. To ensure completion, and prevent infinite loops, we need to define a maximum number of iterations, to cover the case where a local minimum is encountered, and a subsequent iteration results in a distance larger than the previous one is found. This could happen in complex cylinder configurations or if a cylinder erroneously intersects another. The maximum number of iterations is set to 200.

2.2.3 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 evaluated is also important. While the number of iterations 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.

2.2.4 Summary of required settings:

- The tolerance value is 1 metre.
- The maximum number of iterations is 200.
- Points are evaluated sequentially upwards.

3 Definitions

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

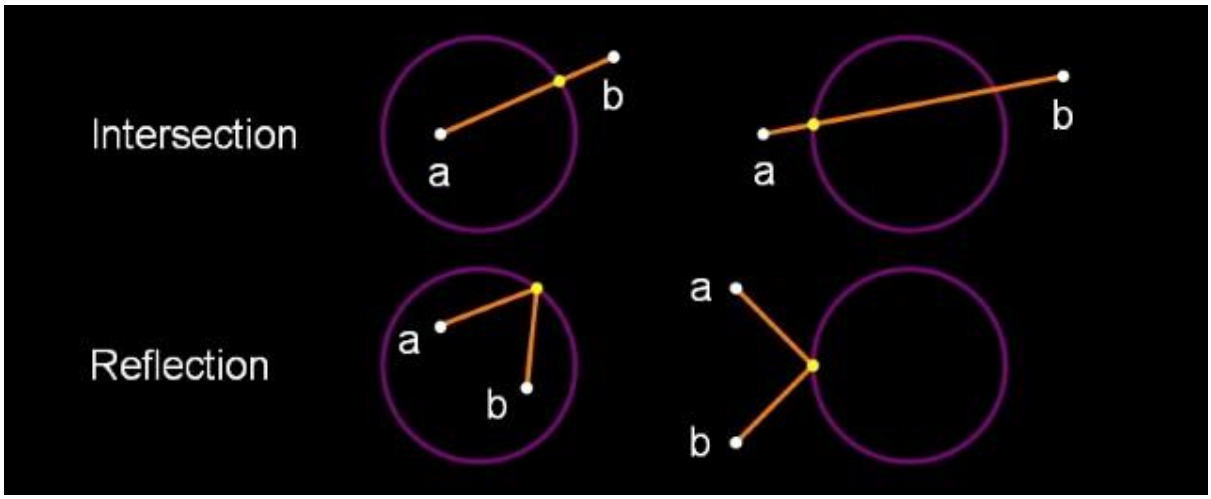
The entry point is the getShortestPath function, which initiates and keeps track of the distances found and the number of iterations.

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 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 iterations. 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 optimized distances obtained by the previous and the current iteration is less than the tolerance, or the maximum number of iterations has been reached.

```
// Inputs:
// points - array of point objects
// line - goal line endpoints, or empty array

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

  // opsCount is the number of iterations allowed
  opsCount = 200;

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

    // See if the difference between the last distance is
    // 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
// line - goal line endpoints, or empty array

function optimizePath(points, count, 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);

    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)

function getTargetPoints(points, count, index)
{
  // 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 last point).

  if (index == count - 1) {
    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.

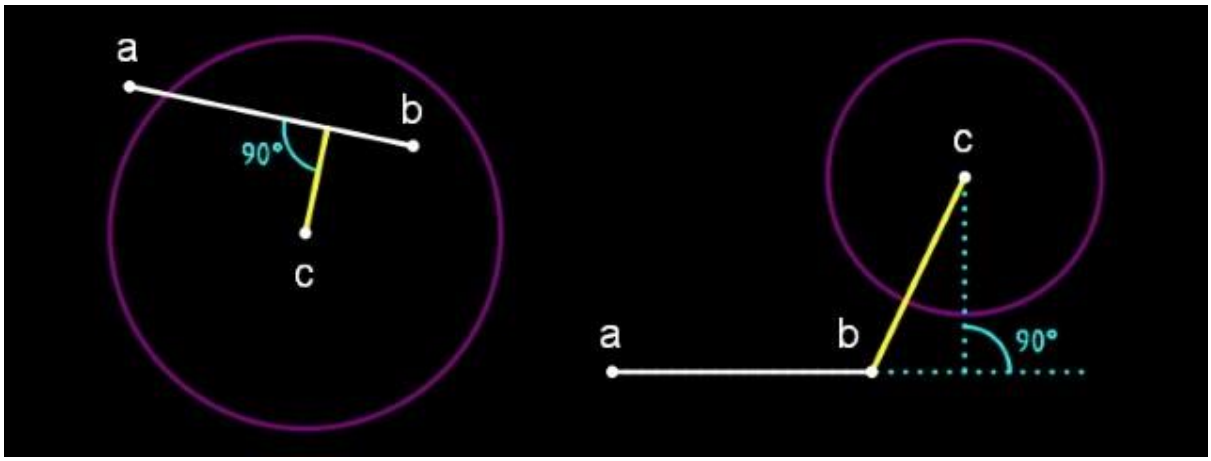
The distance from point C center to the AB line segment determines if it intersects the cylinder (distCtoAB < C radius). If it does and both points are inside the cylinder it requires the reflection solution, otherwise one of the intersection solutions. If the line does not intersect the cylinder or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

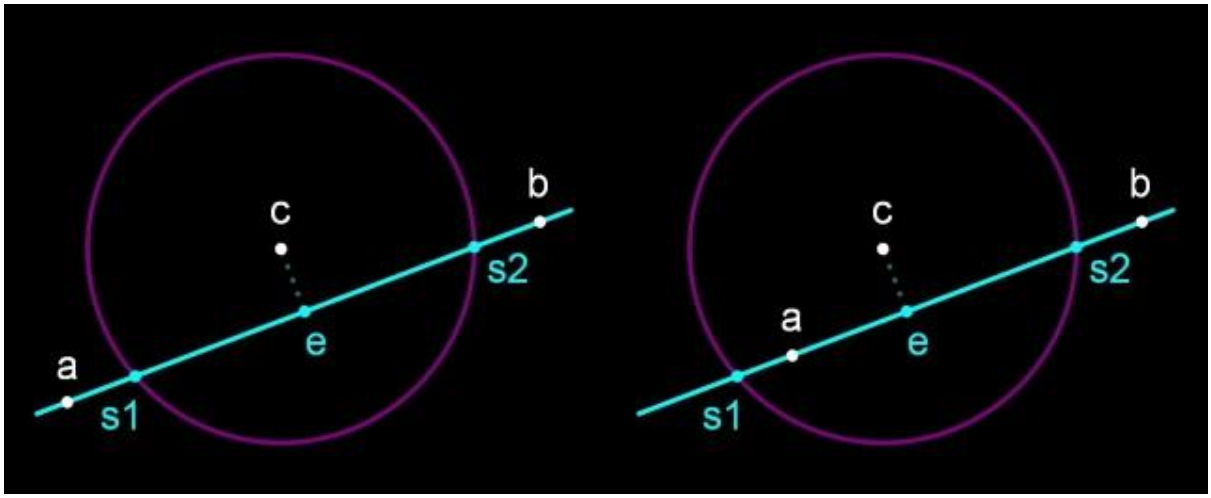


The distance from C to the AB line segment is shown by the yellow line.

```
// 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).



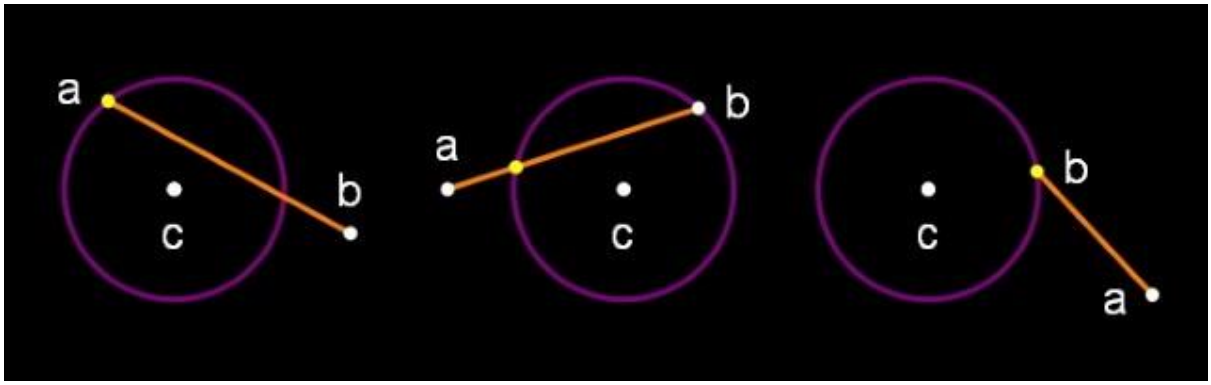

```
// Inputs:
// c, a, b - target cylinder, previous point, next point
// distAB - AB line segment length

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.

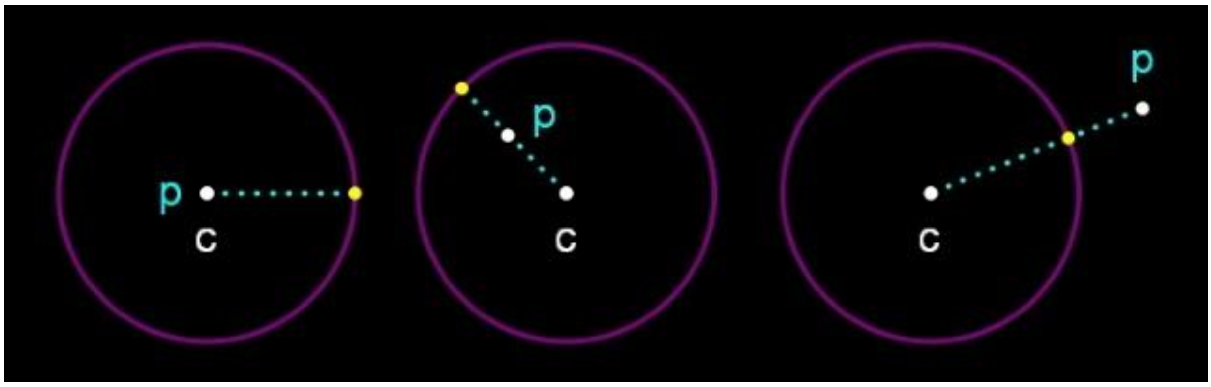
Returns false if neither point is on the circle, otherwise true.




```
// 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

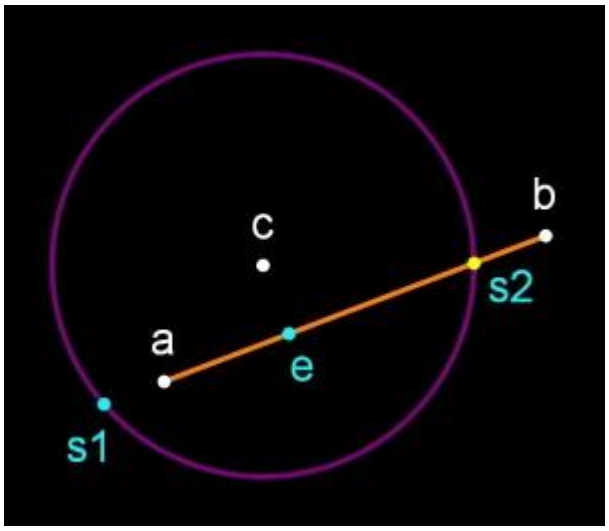
Projects a point (P) on the circle C, at the intersection of the circle by the CP line. The result is stored in the C fix position.




```
// 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.

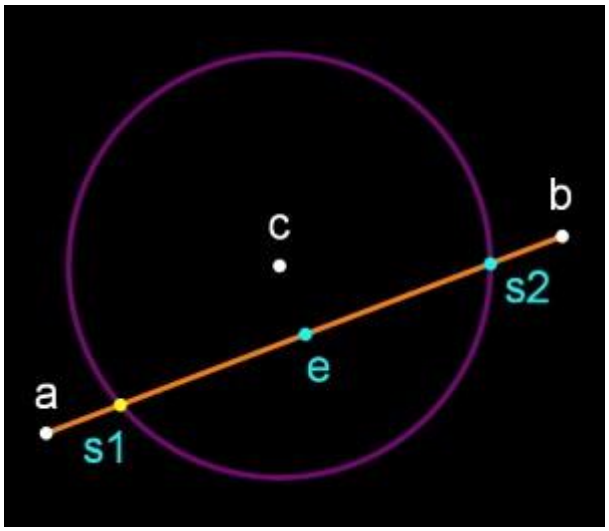



```
// Inputs:
// c, a, b - target cylinder, previous point, next point
// distAB - AB line segment length

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).

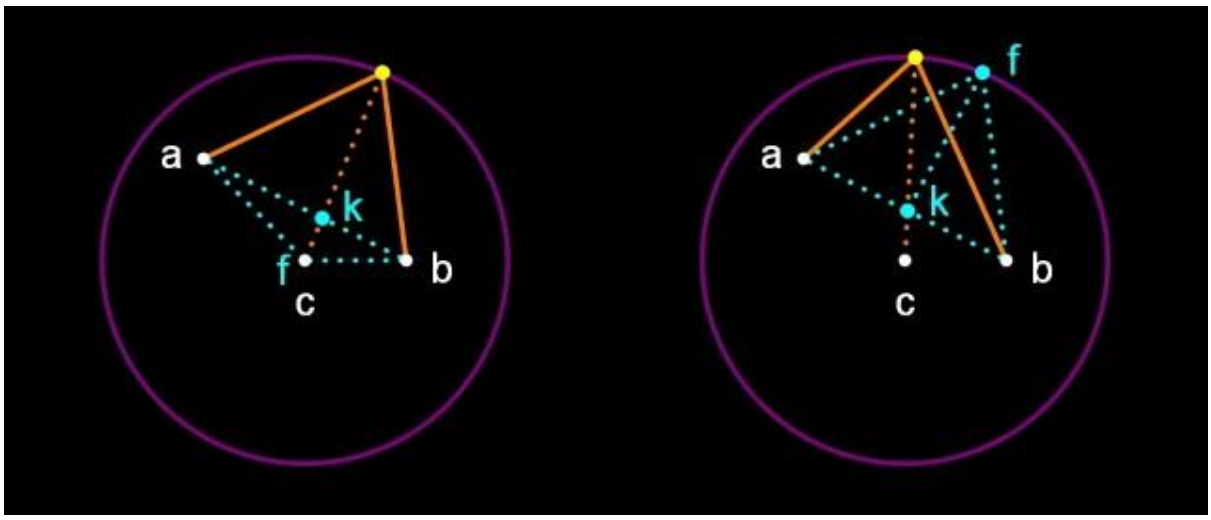
The result is stored in the C fix position and is the intersection that is closest to point A. This will lie between point A and point E (the midpoint of the line between the intersection points).




```
// 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

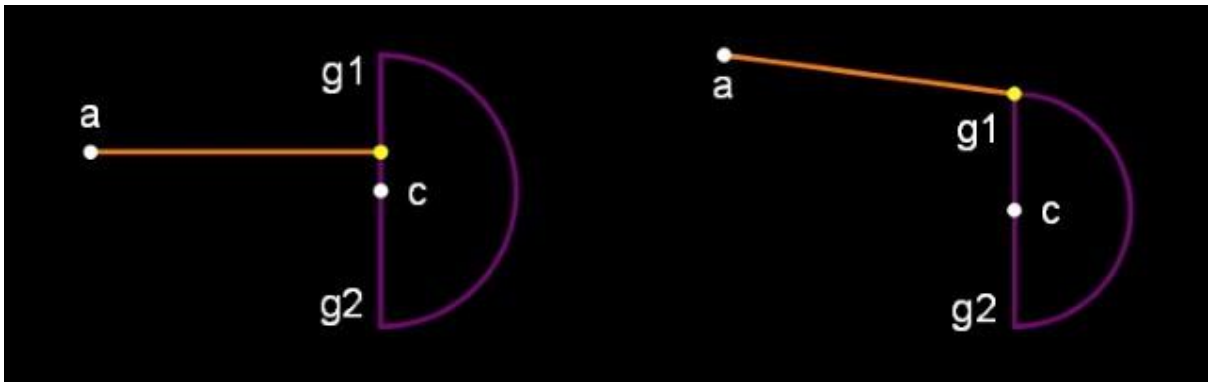
Sets the reflection point of two external or internal points (A and B) on the circle C. This uses the triangle AFB (where F is the current C fix position) to find the point (K) on the AB line segment where it is cut by the AFB angle bisector. The reflected point is found by projecting K on the circle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