NASA/CR–2014-218288

# A Trajectory Algorithm to Support En Route and Terminal Area Self-Spacing Concepts: Third Revision

*Terence S. Abbott*
*Stinger Ghaffarian Technologies, Inc., Hampton, Virginia*

July 2014

## NASA STI Program . . . in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA scientific and technical information (STI) program plays a key part in helping NASA maintain this important role.

The NASA STI program operates under the auspices of the Agency Chief Information Officer. It collects, organizes, provides for archiving, and disseminates NASA's STI. The NASA STI program provides access to the NASA Aeronautics and Space Database and its public interface, the NASA Technical Report Server, thus providing one of the largest collections of aeronautical and space science STI in the world. Results are published in both non-NASA channels and by NASA in the NASA STI Report Series, which includes the following report types:

- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA Programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counterpart of peer-reviewed formal professional papers, but having less stringent limitations on manuscript length and extent of graphic presentations.

- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.

- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.

- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.

- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.

- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services also include organizing and publishing research results, distributing specialized research announcements and feeds, providing information desk and personal search support, and enabling data exchange services.

For more information about the NASA STI program, see the following:

- Access the NASA STI program home page at *http://www.sti.nasa.gov*

- E-mail your question to help@sti.nasa.gov

- Fax your question to the NASA STI Information Desk at 443-757-5803

- Phone the NASA STI Information Desk at 443-757-5802

- Write to:
  STI Information Desk
  NASA Center for AeroSpace Information
  7115 Standard Drive
  Hanover, MD 21076-1320

NASA/CR–2014-218288

# A Trajectory Algorithm to Support En Route and Terminal Area Self-Spacing Concepts: Third Revision

*Terence S. Abbott*
*Stinger Ghaffarian Technologies, Inc., Hampton, Virginia*

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

Prepared for Langley Research Center
under Contract  NNL10AA14B

July 2014

Available from:

NASA Center for AeroSpace Information
7115 Standard Drive
Hanover, MD 21076-1320
443-757-5802

# Table of Contents

## Nomenclature

2D: 2 dimensional

4D: 4 dimensional

ADS-B: Automatic Dependence Surveillance Broadcast

BOD: Bottom-Of-Descent

CAS: Calibrated Airspeed

DTG: Distance-To-Go

MSL: Mean Sea Level

RF: Radius-to-Fix

STAR: Standard Terminal Arrivals

TAS: True Airspeed

TCP: Trajectory Change Point

TOD: Top-Of-Descent

TTG: Time-To-Go

VTCP: Vertical Trajectory Change Point

### Subscripts

Subscripts associated with waypoints and TCPs, e.g., $TCP_2$, denote the location of the waypoint or TCP in the TCP list. Larger numbers denote locations closer to the end of the list, with the end of the list being the runway threshold. Subscripts in variables indicate that the variable is associated with the TCP with that subscript, e.g., $Altitude_2$ is the altitude value associated with $TCP_2$.

## Units and Dimensions

Unless specifically defined otherwise, units (dimensions) are as follows:

| | |
|---|---|
| time: | seconds |
| position: | degrees, + north and + east |
| altitude: | feet, above MSL |
| distance: | nautical miles |
| speed: | knots |
| track: | degrees, true, beginning at north, positive clockwise |

# Abstract

*This document describes an algorithm for the generation of a four dimensional trajectory. Input data for this algorithm are similar to an augmented Standard Terminal Arrival (STAR) with the augmentation in the form of altitude or speed crossing restrictions at waypoints on the route. This version of the algorithm accommodates constant radius turns and cruise altitude waypoints with calibrated airspeed, versus Mach, constraints. The algorithm calculates the altitude, speed, along path distance, and along path time for each waypoint. Wind data at each of these waypoints are also used for the calculation of ground speed and turn radius.*


# Introduction

Concepts for self-spacing of aircraft operating into airport terminal areas have been under development since the 1970's (refs. 1-20). Interest in these concepts has recently been renewed due to a combination of emerging, enabling technology (Automatic Dependent Surveillance Broadcast data link, ADS-B) and the continued growth in air traffic with the ever increasing demand on airport (and runway) throughput. Terminal area self-spacing has the potential to provide an increase in runway capacity through an increase in the accuracy of runway threshold crossing times, which can lead to a decrease of the variability of the runway threshold crossing times. Current concepts use a trajectory based technique that allows for the extension of self-spacing capabilities beyond the terminal area to a point prior to the top of the en route descent.

The overall NASA Langley concept for a trajectory-based solution for en route and terminal area self-spacing is fairly simple and was originally documented in reference 21. By assuming a 4D trajectory for an aircraft and knowing that aircraft's position, it is possible to determine where that aircraft is on its trajectory. Knowing the position on the trajectory, the aircraft's estimated time-to-go (TTG) to a point, in this case the runway threshold, is known. To apply this to a self-spacing concept, a TTG is calculated for a leading aircraft and for the ownship. Note that the trajectories do not need to be the same. The nominal spacing time and spacing error can then be computed as:

nominal spacing time = planned spacing time interval + traffic TTG.

spacing error = ownship TTG – nominal spacing time.

The foundation of this spacing concept is the ability to generate a 4D trajectory. The algorithm presented in this paper uses as input a simple, augmented 2D path definition (i.e., a traditional STAR, with relevant speed and altitude crossing constraints) along with a forecast wind speed profile for each waypoint. The algorithm then computes a full 4D trajectory defined by a series of trajectory change points (TCPs). The input speed (Mach or CAS) or altitude crossing constraint includes the deceleration rate or vertical angle value required to meet the constraint. The TCPs are computed such that speed values, Mach or CAS, and altitudes change linearly between them. TCPs also define the beginning and ending segments of turns, with the midpoint defined as a fly-by waypoint. The algorithm also uses the waypoint forecast wind speed profile in a linear interpolation to calculate the wind speed at the altitude the computed trajectory crosses the waypoint. Wind speed values are then used to calculate the ground speeds along the path.

The major complexity in computing a 4D trajectory involves the interrelationship of ground speed with the path distance around turns. In a turn, the length of the estimated ground path and the associated turn radius will interact with the waypoint winds and with any change in the specified speed during the turn, i.e., a speed crossing-restriction at the waypoint. Either of these conditions will cause a change in the estimated turn radius. The change in the turn radius will affect the length of the ground path which can then interact with the distance to the deceleration point, which thereby affects the turn radius calculation. To accommodate these interactions, the algorithm uses a multi-pass technique in generating the 4D path, with the ground path estimation from the previous calculation used as the starting condition for the current calculation.

## Algorithm Overview

The basic functions for this trajectory algorithm are shown in figure 1. Figure 1 also contains logic and some simple calculations that are not included in the body of this document. Also note that waypoints are considered to be TCPs but not all TCPs are waypoints.

For the 2D input, the first and last waypoints must be fully constrained, i.e., have both a speed and altitude constraint defined. With the exception of the first waypoint, which is the waypoint farthest from the runway threshold, constraints must also include a variable that defines the means for meeting that constraint. For altitude constraints, this is the inertial descent angle; for speed constraints, it is the air mass CAS deceleration rate. A separate, single Mach-to-CAS transition speed (CAS) value may also be input for profiles that involve a constant Mach / CAS descent segment. Additionally, an altitude / CAS restriction (e.g., in the U.S., the 10,000 ft / 250 kt restriction) may also be entered.

The algorithm computes the altitude and speed for each waypoint. It also calculates every point along the path where an altitude or speed transition occurs. These points are considered vertical TCPs (VTCPs). TCPs also define the beginning and ending segments of turns, with the midpoint defined as a fly-by waypoint. Turn data are generated by dividing the turn into two parts (from the beginning of the turn to the midpoint and from the midpoint to the end of the turn) to provide better ground speed (and resulting turn radius) data relative to a single segment estimation. A fixed, average bank angle value is used in the turn radius calculation. The algorithm also uses the forecast wind speed profile for a waypoint in a linear interpolation to calculate the wind speed at the altitude the computed trajectory crosses the waypoint (if the crossing altitude is not at a forecast altitude). For non-waypoint TCPs, the generator uses the forecast wind speed profile from the two waypoints on either side of the TCP in a double linear interpolation based on altitude and distance (to each waypoint). Of significant importance for the use of the data generated by this algorithm is that altitude and speeds (Mach or CAS) change linearly between the TCPs, thus allowing later calculations of DTG or TTG for any point on the path to be easily performed.

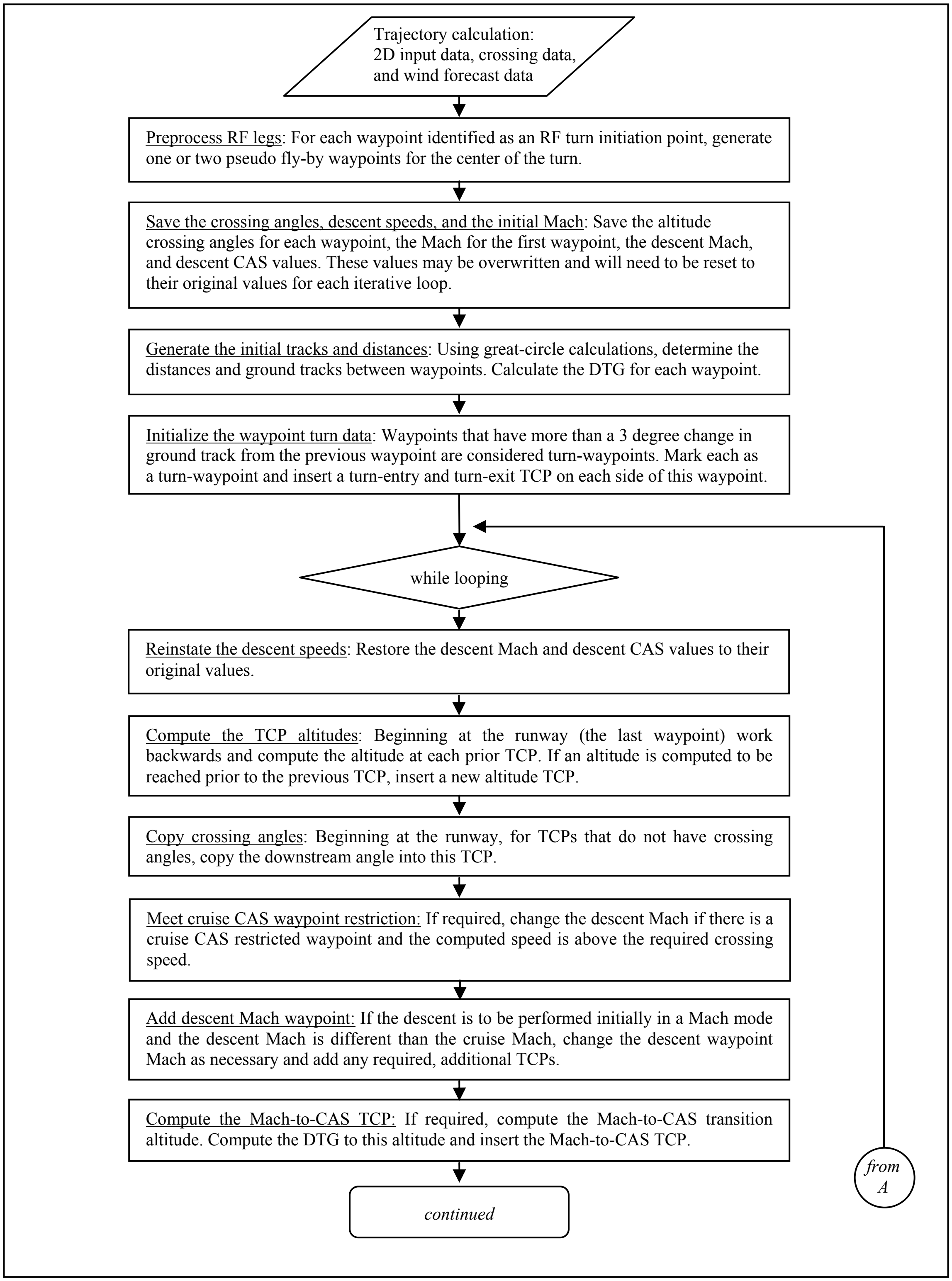


Figure 1. Basic functions.

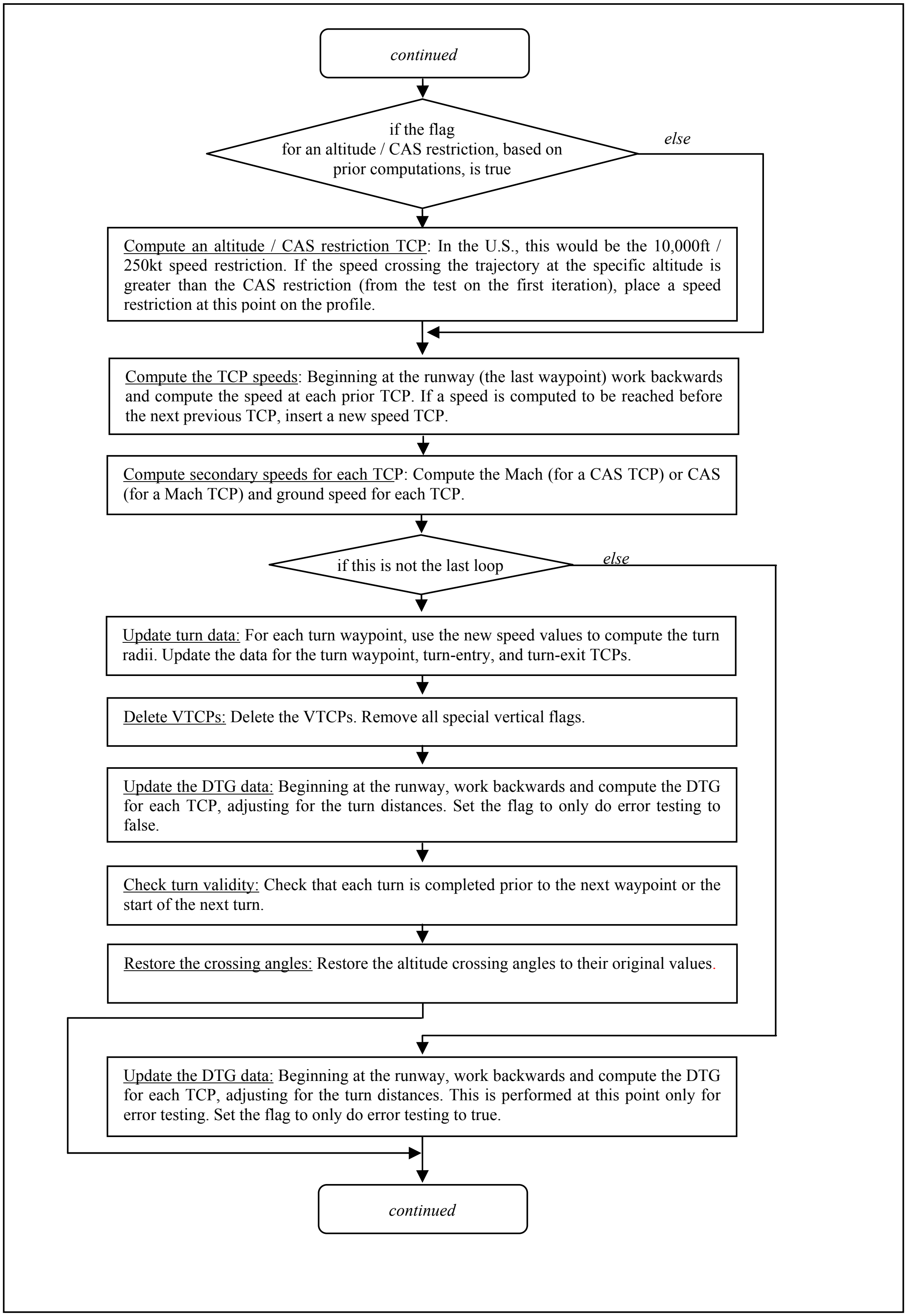


Figure 1 (continued). Basic functions.

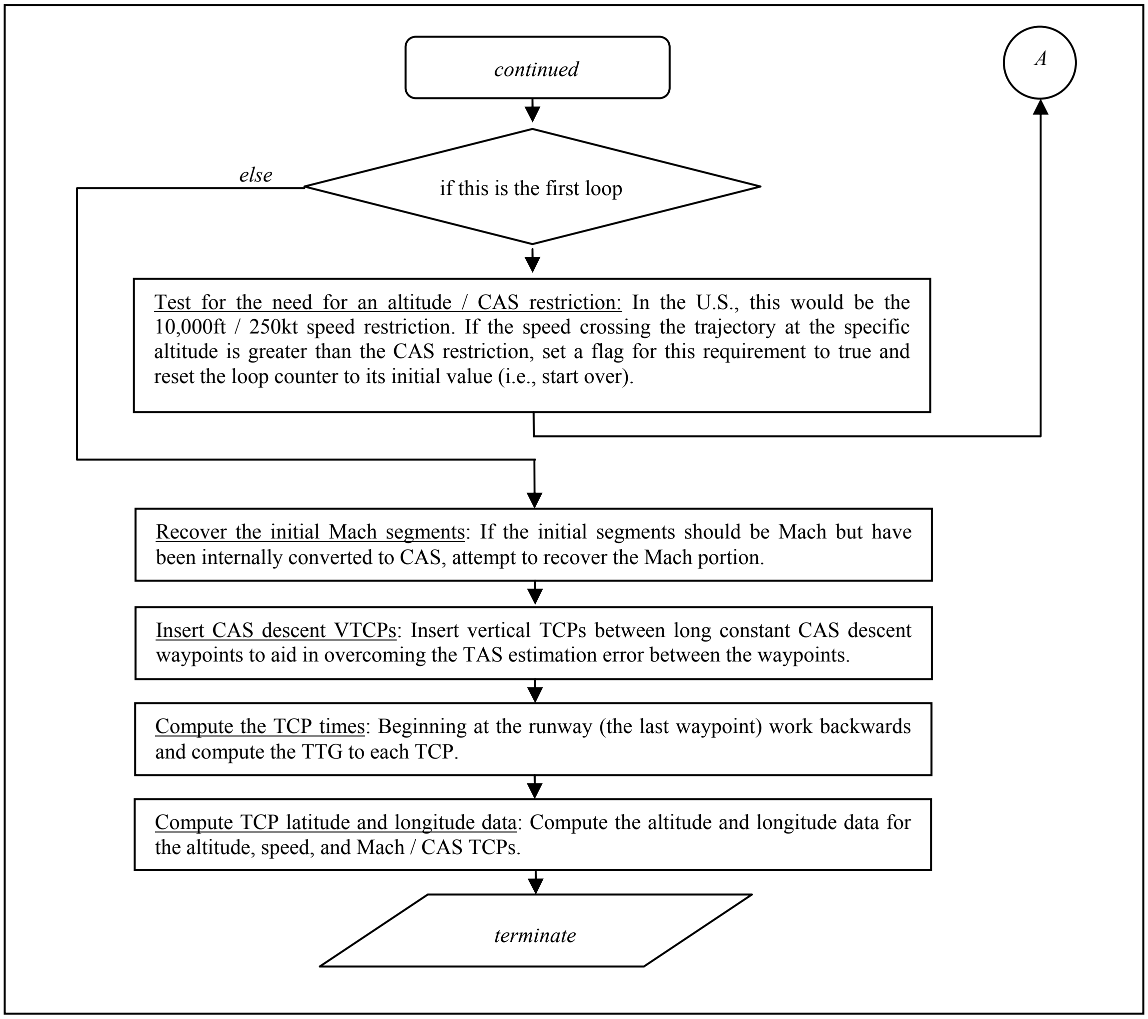


Figure 1 (continued). Basic functions.

## Algorithm Input Data

The algorithm takes as input a list of waypoints, their trajectory-specific data, and associated wind profile data. The list order must begin with the first waypoint on the trajectory and end with the runway threshold waypoint. The trajectory-specific data includes: the waypoint's name and latitude / longitude data, e.g., $Latitude_2$ and $Longitude_2$; an altitude crossing restriction, if one exists, and its associated crossing angle, e.g., *Crossing* $Altitude_2$ and *Crossing* $Angle_2$; and a speed crossing restriction (Mach or CAS), if one exists, and its associated CAS rate, e.g., *Crossing* $CAS_2$ *and Crossing* $Rate_2$. A value of 0 as an input for an altitude or speed crossing constraint denotes that there is no constraint at this point. A *Crossing Mach* may not occur after any non-zero *Crossing CAS* input. The units for *Crossing Rate* are knots per second.

In this algorithm, a radius-to-fix (RF) segment is indicated by the addition of a center-of-turn position, e.g., *Center of Turn* $Latitude_2$ and *Center of Turn* $Longitude_2$, for the input waypoint at the initiation of the turn. Additional requirements for the RF segment are provided in a subsequent section.

To accommodate a descent from the cruise altitude, a Mach value, *Mach Descent Mach*, may be specified that is different from the cruise Mach value. A CAS value may also be specified for the Mach-to-CAS transition speed, *Mach Transition CAS*, during the descent. Additionally, a CAS speed limit at a defined altitude may also be included. In the U.S., this would typically be set to 250 kt at 10,000 ft.

For the wind forecast, a minimum of two altitude reports (altitude, wind speed, and wind direction) should be provided at each waypoint. The altitudes should span the estimated altitude crossing at the associated waypoint. The algorithm assumes that the input data are valid.

## Internal Algorithm Variables

The significant variables computed by this algorithm are:

| | |
|---|---|
| *Altitude* | *the computed altitude at the TCP* |
| *CAS* | *the computed CAS at the TCP* |
| *DTG* | *the computed, cumulative distance from the runway* |
| *Ground Speed* | *the computed ground speed at the TCP* |
| *Ground Track* | *the computed ground track at the TCP* |
| *Mach* | *the computed Mach at the TCP* |
| *TTG* | *the computed, cumulative time from the runway* |

Additionally, the algorithm denotes TCPs in accordance with how they are generated. TCPs are identified as:

- Input, from the input waypoint data;
- An internally generated, radius-to fix (RF) center of turn waypoint;
- Turn-entry, identifying a TCP that marks the start of a turn;

- Turn-exit, identifying a TCP that marks the end of a turn; and
- Vertical TCPs (VTCPs), denoting a change in the altitude or speed profile.

TCPs may also be marked with a vertical identifier denoting one of the following:

- Altitude, denoting a change in the descent angle;
- Speed, denoting a change in the CAS or Mach;
- Top of descent point, TOD;
- Altitude CAS restriction, denoting a speed change due to a speed restriction at a specific altitude, e.g., 250 kt at 10,000'; and
- Mach-to-CAS, denoting the Mach-to-CAS transition point.

TCPs are also denoted relative to the associated primary speed value, i.e., the crossing speed is Mach or CAS derived.

There are also several input variables that may become overwritten within the algorithm that are required to be restored for subsequent calculation cycles within the algorithm. These variables include the following:

- *Saved Mach Descent Mach*, which is the saved input value of *Mach Descent Mach*.
- *Saved Mach Transition CAS*, which is the saved input value of *Mach Transition CAS*.
- *Saved Mach at First Waypoint*, which is the saved input Mach value for the first waypoint, i.e., $Crossing\ Mach_{First\ Waypoint}$, assuming that one exists.

## Description of Major Functions

The functions shown in figure 1 are described in detail in this section. The functions are presented in the order as shown in figure 1. Secondary functions are described in a subsequent section. In these descriptions, the waypoints, which are from the input data and are fixed geographic points, are considered to be TCPs but not all TCPs are waypoints. Nesting levels in the pseudo-code description are denoted by the level of indentation of the document formatting. Additionally, long sections of logic may end with *end of* statements to enhance the legibility of the text.

### Preprocess RF Legs

A radius-to-fix (RF) turn segment is a constant radius turn between two waypoints, with lines tangent to the arc around a center of turn point (fig. 2). This function determines if a valid RF turn exists, and if so, calculates a pseudo-waypoint relative to the center-of-turn point and inserts it into the waypoint list. The calculated pseudo-waypoint then allows the remainder of the turn calculations performed by this algorithm to be processed as a standard turn. This function is performed in the following manner:

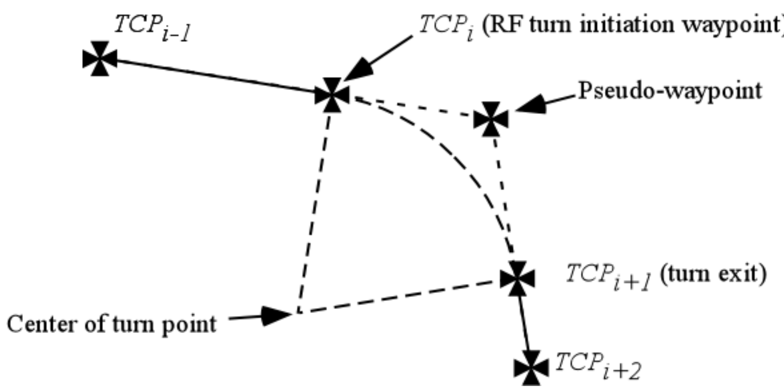


Figure 2. Example of an RF turn.

*error = false*

*Big Turn Error = false*

A set of RF turn waypoints is identified by the inclusion of a non-zero value for the latitude and longitude for the center of turn point in the data for the RF turn initiation waypoint. Because three waypoints are needed in an RF turn calculation, two each for the determination of the inbound and outbound track angles, testing is only performed to the number of the last waypoint - 2.

*for (i = index number of the first waypoint + 1; i ≤ index number of the last waypoint - 2; i = i + 1)*

Determine if this is an RF turn waypoint via the inclusion of the turn center's latitude and longitude data.

*if* $((\text{Center Of Turn Latitude}_i \neq 0)$ *and* $(\text{Center Of Turn Longitude}_i \neq 0))$

Determine the turn direction.

$a_1 = \text{arctangent2}(\ \text{sine}(\text{Longitude}_i - \text{Longitude}_{i-1}) * \text{cosine}(\text{Latitude}_i), \text{cosine}(\text{Latitude}_{i-1}) * \text{sine}(\text{Latitude}_i) - \text{sine}(\text{Latitude}_{i-1}) * \text{cosine}(\text{Latitude}_i) * \text{cosine}(\text{Longitude}_i - \text{Longitude}_{i-1}))$

$a_3 = arctangent2( sine(Longitude_{i+1} - Longitude_i) * cosine(Latitude_{i+1}), cosine(Latitude_i) * sine(Latitude_{i+1}) - sine(Latitude_i) * cosine(Latitude_{i+1}) * cosine(Longitude_{i+1} - Longitude_i))$

$deltax = DeltaAngle(a_1, a_3)$

where the secondary function *DeltaAngle* is described in a subsequent section.

If *deltax* is positive, this is a right-hand turn.

*if (deltax ≥ 0) TurnSign = 1*

*else TurnSign = -1*

Calculate the instantaneous angle at the ending waypoint.

$a_2 = arctangent2( sine(Longitude_{i+1} - Center\ Of\ Turn\ Longitude_i) * cosine(Latitude_{i+1}), cosine(Center\ Of\ Turn\ Latitude_i) * sine(Latitude_{i+1}) - sine(Center\ Of\ Turn\ Latitude_i) * cosine(Latitude_{i+1}) * cosine(Longitude_{i+1} - Center\ Of\ Turn\ Longitude_i)) + TurnSign * 90$

*Adjust* $a_2$ *such that* $0 \geq a_2 \geq 360$

$deltaa = DeltaAngle(a_1, a_2)$

Correct the *deltaa* value if it is in the wrong direction.

*if ((TurnSign > 0) and (deltaa < 0))*

*deltaa = deltaa + 360*

*else if ((TurnSign < 0) and (deltaa > 0))*

*deltaa = deltaa - 360*

If the turn is greater than 170°, break it into two parts so that the standard turn calculations can be performed.

*if (|deltaa| > 170) BigTurn = true*

If the turn is less than 3° or more than 260°, it is in error.

*if ((|deltaa| < 3) or (|deltaa| > 260)) error = true*

Perform a center-of-turn test.

*if (error = false)*

The radius for point 1 must equal the radius for point 2.

$r_1 = arccosine( sine(Center\ Of\ Turn\ Latitude_i) * sine(Latitude_i) + cosine(Center\ Of\ Latitude_i) * cosine(Latitude_i) * cosine(Center\ Of\ Turn\ Longitude_i -$

$Longitude_i)\ \ )$

$r_2 = arccosine(\ sine(Center\ Of\ Turn\ Latitude_i) * sine(Latitude_{i+1}) +$
$cosine(Center\ Of\ Turn\ Latitude_i) * cosine(Latitude_{i+1}) *$
$cosine(Center\ Of\ Turn\ Longitude_i - Longitude_{i+1})\ \ )$

The radii are considered not equal if the difference is greater than 200 ft. The overall RF leg is considered in error if the turn radius is greater than 10 nmi.

$if\ ((|r_1 - r_2| > (200 / 6076))\ or\ (r_1 > 10))\ error = True$

*if (error = false)*

If the turn is greater than 170°, generate two waypoints, otherwise, just generate one waypoint.

*if (BigTurn) n = 2*

*else n = 1*

*a = TurnSign * 90*

$for\ (k = 1; k \leq n; k = k + 1)$

Calculate the pseudo-RF waypoint.

The following is the angle from the turn center toward the pseudo waypoint.

$a_3 = a_1 - a$

$Adjust\ a_3\ such\ that\ 0 \geq a_3 \geq 360$

*if (BigTurn)*

$if\ (\ k = 1)\ a_{1b} = a_3 + 0.25 * deltaa$

$else\ a_{1b} = a_3 + 0.75 * deltaa$

*else*

Just one new waypoint, split the turn in half.

$a_{1b} = a_3 + 0.5 * deltaa$

$Adjust\ a_{1b}\ such\ that\ 0 \geq a_{1b} \geq 360$

*if (k = 1)*

$RadialRadialIntercept(Latitude_i, Longitude_i, a_1,$
$Center\ Of\ Turn\ Latitude_i, Center\ Of\ Turn\ Longitude_i, a_{1b},$
$Latitude_{rf}, Longitude_{rf}),$

noting that $Latitude_{rf}$ *and* $Longitude_{rf}$ are returned values.

*else*

*RadialRadialIntercept(*$Latitude_{i+1}$, $Longitude_{i+1}$, $a_2 + 180$,
*Center Of Turn* $Latitude_{i-1}$, *Center Of Turn* $Longitude_{i-1}$, $a_{1b}$,
$Latitude_{rf}$, $Longitude_{rf}$*),*

The new waypoint is inserted at location $i+1$ in the waypoint list. This inserted waypoint will appear as an input waypoint to the remainder of the algorithm. The waypoint is inserted between $waypoint_i$ and $waypoint_{i+1}$ from the original list. The function *InsertWaypoint* should be appropriate for the actual data structure implementation of this function.

*InsertWaypoint(i + 1)*

Note that $Wpt_{i+1}$ is the newly created waypoint.

*Mark* $Wpt_{i+1}$ *as though it was an input waypoint and give it a unique name.*

Also marking this waypoint as a special, RF turn center waypoint. This special marking is used in subsequent sections to denoted that the center-of-turn point has already been calculated.

$Wpt_{i+1}$ *= rf-turn-center*

$Latitude_{i+1} = Latitude_{rf}$

$Longitude_{i+1} = Longitude_{rf}$

*Copy the wind data from* $Wpt_i$*, the RF initiation waypoint, to* $Wpt_{i+1}$*, the pseudo-waypoint.*

Save the center of turn data. The Turn Data values are associated with each waypoint or TCP record and contain, if appropriate, data relating to turn conditions for that TCP.

*Turn Data Center* $Latitude_{i+1}$ = *Center Of Turn* $Latitude_i$

*Turn Data Center* $Longitude_{i+1}$ = *Center Of Turn* $Longitude_i$

Increment $i$ because a waypoint was added and the new waypoint at $i + 1$ should not be processed again.

$i = i + 1$

*end of for* $(k = 1; k \leq n; k = k + 1)$

*end of if (error = false)*

*end of if ((Center Of Turn* $Latitude_i \neq 0$*) and (Center Of Turn* $Longitude_i \neq 0$*))*

*end of for (i = index number of the first waypoint + 1; ...)*

## Generate Initial Tracks and Distances

This is an initialization function that initializes the *Mach Segment* flag, denoting that the speed in this segment is based on Mach, and calculates the point-to-point distances and ground tracks between input waypoints. Great circle equations are used for these calculations, noting that the various dimensional conversions, e.g., degrees to radians, are not shown in the following text.

Generate the initial distances, the center-to-center distances, and ground tracks between input waypoints

*for (i = index number of the first waypoint; i ≤ index number of the last waypoint; i = i + 1)*

Start with setting the Mach segments flags to false.

*Mach Segment*$_i$ *= false*

Compute the waypoint-center to waypoint-center distances.

*if (i = index number of the first waypoint) Center to Center Distance*$_i$ *= 0*

*else*

*Center to Center Distance*$_i$ *=*
*arccosine( sine(Latitude*$_{i-1}$*) * sine(Latitude*$_i$*) + cosine(Latitude*$_{i-1}$*) * cosine(Latitude*$_i$*) * cosine(Longitude*$_{i-1}$ *- Longitude*$_i$*) )*

*Ground Track*$_{i-1}$ *=*
*arctangent2( sine(Longitude*$_i$ *- Longitude*$_{i-1}$*) * cosine(Latitude*$_i$*), cosine(Latitude*$_{i-1}$*) * sine(Latitude*$_i$*) - sine(Latitude*$_{i-1}$*) * cosine(Latitude*$_i$*) * cosine(Longitude*$_i$ *- Longitude*$_{i-1}$*))*

*end of for (i = index number of the first waypoint; i ≤ index number of the last waypoint; i = i + 1)*

Now set the runway's ground track.

*Ground Track*$_{last\ waypoint}$ *= Ground Track*$_{last\ waypoint\ -\ 1}$

The cumulative distance, DTG, is computed as follows:

*DTG*$_{last\ waypoint}$ *= 0*

*for (i = index number of the last waypoint; i > index number of the first waypoint; i = i - 1)*

*DTG*$_{i-1}$ *= DTG*$_i$ *+ Center to Center Distance*$_i$

### Initialize Waypoint Turn Data

The *Initialize Waypoint Turn Data* function is used to determine if a turn exists at a waypoint and if so, inserts turn-entry and turn-exit TCPs. Waypoints that have more than a 3 degree change in ground track between the previous waypoint and the next waypoint are considered turn-waypoints. The function is performed in the following manner:

*i = index number of the first waypoint + 1*

*Last Track = Ground Track*$_{first\ waypoint}$

Note that the first and last waypoints cannot be turns.

*while (i < index number of the last waypoint)*

*Track Angle After = Ground Track*$_i$

*a = DeltaAngle(Last Track, Track Angle After)*

Check for a turn that is greater than 170 degrees.

*if (|a| > 170)*

Set an error and ignore the turn.

*Mark this as an error condition.*

*a = 0*

If the turn is more than 3-degrees, compute the turn data.

*if (|a| > 3)*

*half turn = a / 2*

*Track Angle Center = Last Track + half turn*

This is the center of the turn, e.g., the original input waypoint.

*Ground Track*$_i$ *= Track Angle Center*

*Turn Data Track1*$_i$ *= Last Track*

*Turn Data Track2*$_i$ *= Track Angle After*

If this is not an RF turn, then the turn radius needs to be calculated.

*if (Wpt*$_i$ *≠ rf-turn-center) Turn Data Turn Radius*$_i$ *= 0*

*Turn Data Path Distance*$_i$ *= 0*

Insert a new TCP at the end of the turn.

The new TCP is inserted at location $i{+}1$ in the TCP list. The TCP is inserted between $TCP_i$ and $TCP_{i+1}$ from the original list. The function *InsertWaypoint* should be appropriate for the actual data structure implementation of this function.

*InsertWaypoint(i + 1)*

Note that $TCP_{i+1}$ is the new TCP.

$TCP_{i+1} = turn\text{-}exit$

$DTG_{i+1} = DTG_i$

$Ground\ Track_{i+1} = Track\ Angle\ After$

The start of the turn TCP is as follows,

*InsertWaypoint(i)*

$TCP_i = turn\text{-}entry$

Note that the original TCP is now at index i + 1.

$DTG_i = DTG_{i+1}$

$Ground\ Track_i = Last\ Track$

$Last\ Track = Track\ Angle\ After$

$i = i + 2$

*end of if* $(|a| > 3)$

*else* $Last\ Track = Ground\ Track_i$

$i = i + 1$

*end of while (i < index number of the last waypoint)*

Effectively, this function:

- Marks each turn-waypoint and sets its ground track angle to the computed angle at the midpoint of the turn.

- Inserts a co-distance turn-entry TCP before this turn-waypoint with the ground track angle for this turn-entry TCP set to the value of the inbound ground track angle.

- Inserts a co-distance turn-exit TCP after this turn-waypoint with the ground track angle for this turn-exit TCP set to the value of the outbound ground track angle.

An example illustrating the inserted turn-start and turn-end TCPs is shown in figure 3.

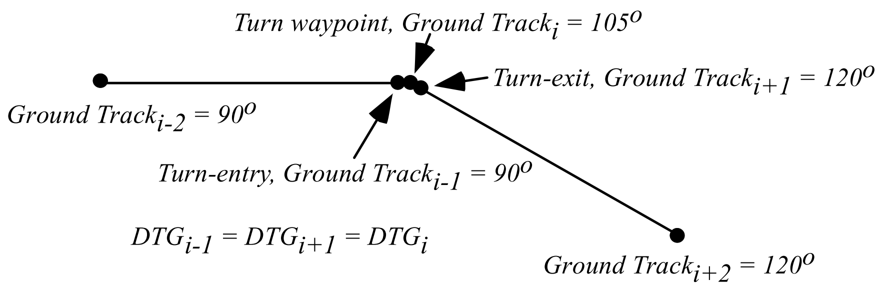


Figure 3. Initialized turn waypoint.

## Compute TCP Altitudes

Beginning with the last waypoint, the *Compute TCP Altitudes* function computes the altitudes at each previous TCP and inserts any additional altitude TCPs that may be required to denote a change in the altitude profile. The function uses the current altitude constraint ($TCP_i$ in fig. 4), searches backward for the previous constraint ($TCP_{i-3}$ in fig. 4), and then computes the distance required to meet this previous constraint. The altitudes for all of the TCPs within this distance are computed and added to the data for the TCPs. If the along-path distance to meet the previous constraint is not at a TCP, a new altitude VTCP is inserted at this distance. An example of this is shown in figure 5. In addition, if the Crossing Angle for a waypoint is set to -99, this denotes that the algorithm is to internally compute the Crossing Angle between this and the next higher, altitude constrained waypoint, noting that this option should only be used in situations where the relevant waypoint pairs are known to procedurally have a fixed angle between them. This function is performed in the following steps:

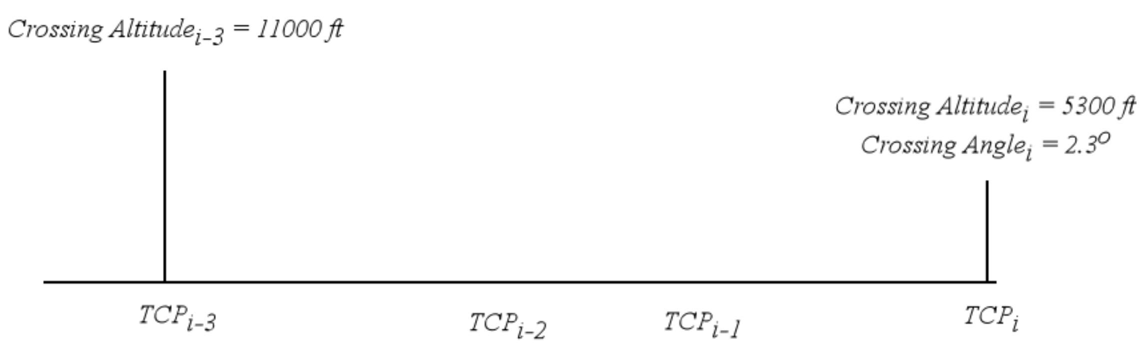


Figure 4. Input altitude crossing constraints.

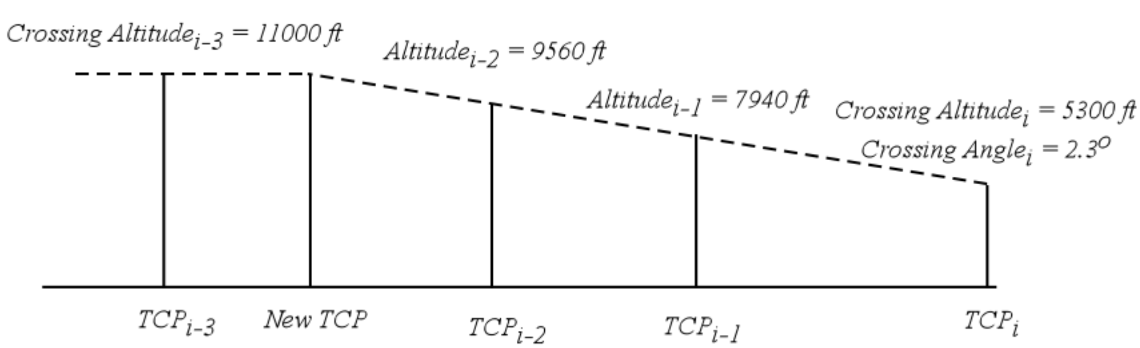


Figure 5. Computed altitude profile with TCP added.

Set the current constraint index number, *cc*, equal to the index number of the last waypoint,

*cc = index number of the last waypoint*

Set the altitude of this waypoint to its crossing altitude,

$Altitude_{cc} = Crossing\ Altitude_{cc}$

Set a flag denoting that the TOD point has not been identified

*Have TOD = false*

*While (cc > index number of the first waypoint)*

If this is the TOD, mark this point.

*if Have TOD is false and* $Altitude_{cc}$ *is equal or greater than* $Altitude_1$

*Have TOD = true*

*mark this as the TOD point.*

Determine if the previous constraint cannot be met.

*If* $(Altitude_{cc} > Crossing\ Altitude_{cc})$

The constraint has not been made.

*If this is the last pass through the algorithm, mark this as an error condition.*

$Altitude_{cc} = Crossing\ Altitude_{cc}$

Find the prior waypoint index number *pc* that has an altitude constraint, e.g., a crossing altitude ($Crossing\ Altitude_{pc} \neq 0$). This may not always be the previous (i.e., *cc - 1*) waypoint.

Initial condition is the previous TCP.

*pc = cc - 1*

*while ( (pc > index number of the first waypoint) and (* $(TCP_{pc} \neq$ *input waypoint) or* $(Crossing\ Altitude_{pc} = 0))$ *) pc = pc - 1*

Save the previous crossing altitude,

$Prior\ Altitude = Crossing\ Altitude_{pc}$

Save the current crossing altitude (*Test Altitude*) at $TCP_{cc}$ and the descent angle (*Test Angle*) noting that the first and last waypoints always have altitude constraints and except for the first waypoint, all constrained altitude points must have descent angles.

$Test\ Altitude = Crossing\ Altitude_{cc}$

$Test\ Angle = Crossing\ Angle_{cc}$

If the Test Angle value, i.e., AUTO DESCENT ANGLE, denotes that this is angle is to be computed internally as a linear descent between the two altitude constrained waypoints then the following calculations are performed:

*if (Test Angle = AUTO DESCENT ANGLE)*

$dx = DTG_{pc} - DTG_{cc}$

$dy = Prior\ Altitude - Test\ Altitude$

$Test\ Angle = arctangent2\ (dy, 6076 * dx)$

$Crossing\ Angle_{cc} = Test\ Angle$

Test for an extreme angle, e.g., 7.5°.

*if (Test Angle > maximum allowable descent angle) mark this as an error condition.*

Compute all of the TCP altitudes between the current TCP and the previous crossing waypoint.

$k = cc$

*while* $k > pc$

If the previous altitude has already been reached, set the remaining TCP altitudes to the previous altitude.

*if (Prior Altitude ≤ Test Altitude)*

*for* $(k = k - 1;\ k > pc;\ k = k - 1)\ Altitude_k = Test\ Altitude$

Set the altitude at the last test point.

$Altitude_{pc} = Test\ Altitude$

*else*

Compute the distance from $TCP_k$ to the *Prior Altitude* using the altitude difference between the *Test Altitude* and the *Prior Altitude* with the *Test Angle*. If there is no point at this distance, add a TCP at that distance.

Compute the distance *dx* to make the altitude.

$dx = (Prior\ Altitude - Test\ Altitude) / (6076 * tangent(Test\ Angle))$

Compute the altitude *z* at the previous TCP.

$z = ((DTG_{k-1} - DTG_k) * 6076) * tangent(Test\ Angle) + Test\ Altitude$

If there is a TCP prior to this distance or if $z$ is very close to the *Prior Altitude*, compute and insert its altitude.

*if ( (*$DTG_{k-1} < (DTG_k + dx)$*) or ( |z - Prior Altitude| < some small value) )*

*if (|z - Prior Altitude| < some small value)* $Altitude_{k-1}$ *= Prior Altitude*

*else* $Altitude_{k-1} = z$

Check to see if the constraint has been reached with a 100 ft tolerance; if not, set an error condition.

*if ( (k-1) = pc)*

*if ( |*$Altitude_{pc}$ *- Crossing* $Altitude_{pc}$*| > 100ft ) mark this as an error condition*

Always set the crossing exactly to the crossing value.

$Altitude_{pc}$ *= Crossing* $Altitude_{pc}$

Update the Test Altitude.

*Test Altitude =* $Altitude_{k-1}$

Decrement the counter to set it to the prior TCP.

$k = k - 1$

*end of if ( (*$DTG_{k-1} < (DTG_k + dx)$*) ) or ( |z - Prior Altitude| < some small value) )*

*else*

The altitude constraint is reached prior to the TCP, a new VTCP will need to be inserted at that point. The distance to the new TCP is,

$d = DTG_k + dx$

Compute the ground track at distance $d$ along the trajectory and save it as *Saved Ground Track*.

*Saved Ground Track = GetTrajGndTrk(d)*

Insert a new VTCP at location $k$ in the TCP list. The VTCP is inserted between $TCP_{k-1}$ and $TCP_k$ from the original list. The function *InsertWaypoint* should be appropriate for the actual data structure implementation of this function.

*InsertWaypoint(k)*

Update the data for the new VTCP which is now $TCP_k$.

*if (*$VSegType_k$ *= no type)* $VSegType_k$ *= ALTITUDE*

$DTG_k = d$

$Altitude_k = Prior\ Altitude$

Add the ground track data which must be computed if the new VTCP occurs within a turn. The functions *WptInTurn* and *ComputeGndTrk* are described in subsequent sections.

*if (WptInTurn(k))* $Ground\ Track_k = ComputeGndTrk(k, d)$

*else* $Ground\ Track_k = Saved\ Ground\ Track$

Compute and add the wind data at distance *d* along the path to the data of $TCP_k$.

$GenerateWptWindProfile(d, TCP_k)$

$Test\ Altitude = Prior\ Altitude$

Since $TCP_k$, has now been added prior to *pc*, the current constraint counter *cc* needs to be incremented by 1 to maintain its correct position in the list.

$cc = cc + 1$

The function loops back to *while k > pc.*

Now go to the next altitude change segment on the profile.

$cc = k$

The function loops back to *while cc > index number of the first waypoint.*

## Copy Crossing Angles

The *Copy Crossing Angles* is a simple function that starts with the next to last TCP and copies the subsequent crossing angle if the current TCP does not have a crossing angle. E.g.,

*for (i = index number of the last waypoint - 1; i ≥ index number of the first waypoint; i = i - 1)*

*if* $(Crossing\ Angle_i = 0)$ $Crossing\ Angle_i = Crossing\ Angle_{i+1}$

## Meet Cruise CAS Waypoint Restriction

The *Meet Cruise CAS Waypoint Restriction* function changes, if required, the descent Mach if there is a high altitude, CAS restricted waypoint and the computed speed is above the required crossing speed for that CAS waypoint.

The calling function provides as input and retains the subsequent outputs for the following variables: *TodId, TodMach, TodMachRate,* and *MachCasAtTod*. The variable *TodId* is the name of the top-of-descent waypoint (TOD) and is initialized as an empty string by the calling program. This *Meet Cruise CAS Waypoint Restriction* function may modify the Mach and speed change rate that occurs at the TOD, *TodMach* and *TodMachRate*, respectively, and these values are then passed to subsequent functions that require these data. The variable *MachCasAtTod* is a flag that if true, indicates that the Mach-to-CAS transitions occurs at the TOD point.

If the Mach value for the first waypoint is not set, i.e., the path does not start with a Mach segment, and the function terminates with *MachCasAtTod* set to false. Otherwise, the following is performed.

*if (Crossing Mach* $_{first\ waypoint}$ *= 0) terminate this function. Otherwise,*

Set the initial values.

*MachCasAtTod = false*

*MachCasModified = false*

*CasIndex = index number of the first waypoint*

*AltAtMach = 0.*

*LastMach = 0*

*z = 0*

*done = false*

If the TOD Mach data have been modified in a previous invocation of *Add Descent Mach Waypoint*, indicated by a non-empty value for *TodId*, reset their values.

*if (TodId ≠ empty)*

*fini = false*

*i = index number of the first waypoint*

Find the waypoint with the name defined in *TodId.*

*while ((i ≤ (index number of the last waypoint)) and ( fini = false ))*

*if (*$Id_i$ *= TodId)*

*fini = true*

*Crossing* $Mach_i$ *= TodMach*

*Crossing* $CAS_i$ *= 0*

*Crossing* $Rate_i$ *= TodMachRate*

*TodId = empty string*

*i = i + 1*

*end of if (TodId ≠ empty)*

Find the first CAS waypoint.

*fini = false*

*i = index number of the first waypoint*

*while ((i ≤ index number of the last waypoint) and (fini = false))*

*if (Crossing* $CAS_i$ *> 0)*

*CasIndex = i*

*fini = true*

*i = i + 1*

Determine if the trajectory is already at the CAS altitude, i.e., the initial altitude is the CAS altitude, and if so, start in a CAS mode, not Mach.

*if (Crossing* $Altitude_{first\ waypoint} = Altitude_{CasIndex}$*)*

*done = true*

*for (k = index number of the first waypoint; k < CasIndex; k = k + 1)*

*if (Crossing* $Mach_k$ *> 0)*

Change the route data so that the trajectory is starting in a CAS mode.

Invoke the secondary function *MachToCas*. This function is described in a subsequent section.

*Crossing* $CAS_k$ *= MachToCas( Crossing* $Mach_k$*,* $Altitude_{CasIndex}$ *)*

*Crossing* $Mach_k = 0$

$MachSegment_k$ *= false*

*end of if (Crossing* $Mach_k$ *> 0)*

*if (done = false)*

Find the last Mach value.

*fini = false*

*i = index number of the first waypoint*

*while ((i < index number of the last waypoint) and (fini = false))*

*if ( Crossing* $CAS_i$ *> 0) fini = true*

*else if ( Crossing* $Mach_i$ *> 0) LastMach = Crossing* $Mach_i$

$i = i + 1$

Determine the descent Mach value.

*if (Mach Descent Mach ≠ 0) DescentMach = Mach Descent Mach*

*else DescentMach = LastMach*

Determine the Mach-to-CAS transition CAS value.

*if ( Mach Transition CAS > 0)*

*MachCas = Mach Transition CAS*

*if (Mach Transition CAS < Crossing* $CAS_{CasIndex}$*) MachCas = Crossing* $CAS_{CasIndex}$

*else MachCas = Crossing* $CAS_{CasIndex}$

Find the last Mach altitude.

*fini = false*

*i = index number of the first waypoint*

*while ((i ≤ index number of the last waypoint) and (fini = false))*

*if (Crossing* $CAS_i$ *> 0) fini = true*

*else if (Crossing* $Altitude_i$ *> 0) AltAtMach = Crossing* $Altitude_i$

$i = i + 1$

Determine if the Mach is slower than the descent CAS.

Invoke the secondary function *MachCasTransitionAltitude* which calculates the altitude where the Mach and CAS are equal. This function is described in a subsequent section.

*z = MachCasTransitionAltitude( MachCas, DescentMach )*

*if (z > Crossing* $Altitude_{first\ waypoint}$*)*

The path is already below the transition altitude, change the route data so it starts in a CAS mode.

*for (k = index number of the first waypoint; k < index number of the last waypoint; k = k + 1)*

*done = true*

*if (Crossing* $Mach_k$ *> 0)*

*Crossing* $CAS_k$ *= MachCas*

*Crossing* $Mach_k = 0$

*MachSegment*$_k$ *= false*

*end of if (done = false)*

*if (done = false)*

Find the last Mach value.

*fini = false*

*i = index number of the first waypoint*

*while ((i ≤ index number of the last waypoint) and (fini = false))*

*if (Crossing* $CAS_i > 0$*) fini = true*

*else if (Crossing* $Mach_i > 0$*) LastMach = Crossing* $Mach_i$

$i = i + 1$

Determine the descent Mach.

*if (Mach Descent Mach ≠ 0) DescentMach = Mach Descent Mach*

*else DescentMach = LastMach*

Find the Mach-to-CAS transition CAS.

*if (Mach Transition CAS > 0) MachCas = Mach Transition CAS*

Make sure that the crossing restriction can be obtained.

*if (Mach Transition CAS < Crossing* $CAS_{CasIndex}$*) MachCas = Crossing* $CAS_{CasIndex}$

*else MachCas = Crossing* $CAS_{CasIndex}$

Find the last Mach altitude.

*fini = false*

*i = index number of the first waypoint*

*while ((i ≤ index number of the last waypoint) and (fini = false))*

*if (Crossing* $CAS_i > 0$*) fini = true*

*else if (Crossing* $Altitude_i > 0$*) AltAtMach = Crossing* $Altitude_i$

$i = i + 1$

Determine if the Mach is slower than the descent CAS.

*z = MachCasTransitionAltitude( MachCas, DescentMach )*

*if (z > Crossing* $Altitude_{first\ waypoint}$*)*

The path is already below the transition altitude, change the route data so it is starting in a CAS mode.

*for (k = index number of the first waypoint; k < index number of the last waypoint; k = k + 1)*

*done = true*

*if (Crossing* $Mach_k$ *> 0)*

*Crossing* $CAS_k$ *= MachCas*

*Crossing* $Mach_k$ *= 0*

$MachSegment_k$ *= false*

*end of if (done = false)*

If the path still starts with a Mach segment, which may have already been modified in this function, test for other special cases.

*if (done = false)*

If required, handle the special case of an accelerated descent.

*if (DescentMach > LastMach)*

Invoke the secondary function *HandleDescentAccelDecel*. This function handles the special case of a Mach acceleration in the descent where the first CAS crossing restriction cannot be met. This function is described in a subsequent section. This function may modify the waypoint data.

*HandleDescentAccelDecel( CasIndex, LastMach, MachCasModified, DescentMach, MachCas )*

If the descent data are changed, recalculate *z*.

*if (MachCasModified)*

*z = MachCasTransitionAltitude ( MachCas, DescentMach )*

Next, update the waypoint data.

*Mach Descent Mach = DescentMach*

*Mach Transition CAS = MachCas*

*end of if (DescentMach > LastMach)*

*if (z < Crossing* $Altitude_{CasIndex}$*)*

At this point, the descent CAS or Mach needs to be changed.

*m = CasToMach( MachCas, Crossing* $Altitude_{CasIndex}$ *)*

*if (m > DescentMach)*

Change the descent CAS.

*MachCas = MachToCas( DescentMach, Crossing* $Altitude_{CasIndex}$ *)*

*else*

*DescentMach = CasToMach( MachCas, Crossing* $Altitude_{CasIndex}$ *)*

*Mach Descent Mach = DescentMach*

*z = Crossing Altitude* $_{CasIndex}$

Perform an extreme limits test, assuming that a valid Mach value will be between 0.6 and 0.9 Mach.

*if ((DescentMach > 0.9) or (DescentMach < 0.6)) mark this as an error condition*

*end of if (z < Crossing* $Altitude_{CasIndex}$*)*

Make sure that there is sufficient distance to slow from the Mach-to-CAS transition speed to make the crossing CAS.

*if ( (z ≥ Crossing* $Altitude_{CasIndex}$*) and (MachCas > Crossing* $CAS_{CasIndex}$*) and (Crossing* $Rate_{CasIndex}$ *> 0) and (MachCasModified = false))*

Find the distance at *z*. This is an iterative solution.

*i = CasIndex - 1*

*fini = false*

Calculate the headwind at the end point. This calculation the secondary function *InterpolateWindWptAltitude*, described in a subsequent section.

*InterpolateWindWptAltitude( Wind* $Profile_{CasIndex}$*,* $Altitude_{CasIndex}$*, Ws, Wd )*

*HeadWind = Ws * cosine( Wd - GndTrack* $_{CasIndex}$ *)*

*CurrentGs = ComputeGndSpeedUsingTrack( Crossing* $CAS_{CasIndex}$*,* $GndTrack_{CasIndex}$*,* $Altitude_{CasIndex}$*, Ws, Wd)*

*Iterate = false*

$OnePass = true$

$MCasHold = MachCas$

$LastCut = 0$

$while\ (fini = false)$

$i = CasIndex - 1$

$while\ ((i > \text{index number of the first waypoint})\ and\ (Altitude_{i} < z))\ i = i - 1$

$if\ ((Altitude_{i} - Altitude_{i+1}) \leq 0)\ a = 0$

$else\ a = (z - Altitude_{i+1}) / (Altitude_{i} - Altitude_{i+1})$

Calculate the distance, *dx*, required to reach the altitude.

$dx = a * (DTG_{i} - DTG_{i+1}) + DTG_{i+1} - DTG_{CasIndex}$

$InterpolateWindWptAltitude(\ Wind\ Profile_{CasIndex},\ z,\ Ws2,\ Wd2\ )$

$Hw2 = Ws2 * cosine(\ Wd2 - GndTrack_{i}\ )$

$AvgHw = (HeadWind + Hw2) / 2$

Invoke the secondary function *EstimateNextCas*. *EstimateNextCas* is an iterative function to estimate the CAS value at the next waypoint.

$CasTest = EstimateNextCas(\ Crossing\ CAS_{CasIndex},\ CurrentGs,\ true,\ MCasHold,\ AvgHw,\ z,\ dx,\ Crossing\ Rate_{CasIndex})$

If required, set up the iteration values, where the iteration value is in CAS.

$if\ (OnePass = true)$

$if\ (CasTest < MachCas)\ Iterate = true$

$else\ fini = true$

$OnePass = false$

Calculate the iteration step size.

$LastCut = |MachCas - CasTest|$

Limit the step size to no smaller than 2 kt.

$if\ (LastCut < 2)\ LastCut = 2$

$if\ (Iterate)$

*if (MachCas ≥ CasTest) s = MachCas - LastCut*

*else s = MachCas + LastCut*

*LastCut = 0.5 * LastCut*

*if (s > MCasHold) s = MCasHold*

Determine if the Mach-to-CAS estimate is valid.

*if (((s + 0.25) ≥ MachCas) and (|s - MachCas| < 1))*

  *fini = true*

  Calculate the Mach-to-CAS altitude for the current estimate.

  *z = MachCasTransitionAltitude ( MachCas, DescentMach )*

  Determine if a deceleration is needed prior to the TOD. Add a 50 ft buffer value.

  *if (z > (AltAtMach + 50))*

    Find the TOD waypoint.

    *fini2 = false*

    *j = index number of the first waypoint*

    *while ((j < index number of the last waypoint) and (fini2 = false))*

      *if (*$Waypoint_j$ *is marked as the TOD point) fini2 = true*

      *else j = j + 1*

    The altitude index for the test is the TOD altitude point.

    *if (fini2 and (i = j))*

      *Mach Descent Mach = CasToMach( Mach Transition CAS, AltAtMach )*

      *MachCasAtTod = true*

  *end of if (z > (AltAtMach + 50))*

*end of if (((s + 0.25) ≥ MachCas) and (|s - MachCas| < 1))*

*else*

  *Mach Transition CAS = s*

  *MachCas = s*

*z = MachCasTransitionAltitude( MachCas, DescentMach )*

*if (z > Altitude$_i$) z = Altitude$_i$*

*j = j + 1*

Add a test to limit the number of iterations to 10.

*if (j ≥ 10) fini = true*

*end of if (Iterate)*

*end of while (fini = false)*

*end of if (done = false)*

### Add Descent Mach Waypoint

The *Add Descent Mach Waypoint* function changes the descent waypoint Mach if the descent Mach, *Mach Descent Mach,* is different than the cruise Mach. The function also will add any required, additional TCPs.

The calling program provides as input and retains the subsequent outputs for the following variables: *TodId, TodMach, and TodMachRate*. The variable *TodId* is the name of the top-of-descent waypoint and is initialized as a null string by the calling program. Since this function may overwrite the Mach and speed change rate for an input waypoint, these variables allow the function to retain the original values for Mach and speed change rate and to then reset these variables to their original values prior to recalculating new values.

If the Mach value for the first waypoint is not set, i.e., the path does not start with a Mach segment, or there is no defined descent Mach, i.e., *Mach Descent Mach = 0*, the function terminates. Otherwise,

If the previous TOD data for an input waypoint have been changed, these data are restored to their original values.

*fini = false*

*i = index number of the first waypoint*

The last designated Mach waypoint,

*LastMachIndex = index number of the first waypoint*

The first designated CAS waypoint,

*FirstCasIndex = index number of the first waypoint*

*TodIndex = 0*

Find the Mach and CAS waypoints.

*fini = false*

*i = index number of the first waypoint*

*while ( (i ≤ index number of the last waypoint) and (fini = false) )*

*if (Crossing* $Mach_i$ *> 0) LastMachIndex = i*

*else if (Crossing* $CAS_i$ *> 0)*

*FirstCasIndex = i*

*fini = true*

*i = i + 1*

Find the TOD waypoint and Mach.

*fini = false*

*i = index number of the first waypoint*

*while ( (i <index number of the last waypoint) and (fini = false) )*

*if ((*$Altitude_i < Altitude_{first\ waypoint}$*) or (Cas* $Cross_i$ *> 0))*

*if (*$Altitude_i \neq Altitude_{first\ waypoint}$*) TodIndex = i - 1*

*else TodIndex = i*

*fini = true*

*else if (Crossing* $Mach_i$ *> 0) MachAtTod = Crossing* $Mach_i$

*i = i + 1*

If the vertical segment type has not been defined, mark this as the TOD.

*if ((TodIndex > 0) and (*$VSegType_{TodIdx}$ *= no type))* $VSegType_{TodIdx}$ *= TOD ALTITUDE*

Check for errors. There cannot be a programmed descent Mach if there is a downstream Mach restriction.

*if ( (LastMachIndex > TodIndex) or (FirstCasIndex ≤ TodIndex) ) mark this as an error condition*

*else*

Save the Mach values for all input waypoints so that they may be reset on subsequent passes back to their original input values.

*if (*$Waypoint_{TodIndex}$ *= input waypoint)*

*copy the name of* $Waypoint_{TodIndex}$ *into TodId*

*TodMach* = *Crossing* $Mach_{TodIndex}$

*TodMachRate* = *Crossing* $Rate_{TodIndex}$

*if ( (*$Waypoint_{TodIndex}$ *= input waypoint) and (Crossing* $Rate_{TodIndex}$ *> 0) )*

*CAS Rate = Crossing* $Rate_{TodIndex}$

*else CAS Rate = 0.75 kt / sec (a default value)*

The following is added to force a subsequent speed calculation.

*Crossing* $Rate_{TodIndex}$ *= CAS Rate*

If the aircraft will slow during the descent, do the following:

*if (MachAtTod ≥ Mach Descent Mach)*

Overwrite the TOD Mach value.

*Crossing* $Mach_{TodIndex}$ *= Mach Descent Mach*

*else*

This is a special case where the aircraft is accelerating to the descent Mach.

Invoke the secondary function *DoTodAcceleration*. This function is described in a subsequent section.

*DoTodAcceleration( TodIdx, MachAtTod )*

*Crossing* $Mach_{TodIndex}$ *= MachAtTod*

## Compute Mach-to-CAS TCP

If a Mach-to-CAS transition is required, this functions computes the Mach-to-CAS altitude and inserts a Mach-to-CAS TCP. This function is only performed if the input data starts with a Mach *Crossing Speed* for the first waypoint. The function determines the appropriate Mach and CAS values, calculates the altitude that these values are equal, and then determines the along-path distance where this altitude occurs on the profile. A Mach-to-CAS TCP is then inserted into the TCP list.

Find the last *Crossing Mach* and the first *Crossing CAS* in the list.

*First CAS = 0*

*i = index number of the first waypoint*

*while ( (i <index number of the last waypoint) and (First CAS = 0) )*

*if (Crossing* $Mach_i$ *> 0)*

*Last Mach = Crossing* $Mach_i$

$Last\ Mach\ Altitude = Altitude_i$

$else\ if\ (Crossing\ CAS_i > 0)$

$First\ CAS = Crossing\ CAS_i$

$CAS\ Rate = CAS\ Rate_i$

$i = i + 1$

If there is a Mach-to-CAS CAS transition speed input, use this value for the *First CAS* value.

*if (Mach Transition CAS > 0) First CAS = Mach Transition CAS*

Compute the Mach-to-CAS transition altitude.

*z = ComputeMachCasAltitude( FirstCas, LastMach )*

For an actual implementation, it would be beneficial to check for an error at this point. If $z$ is greater than the altitude associated with the *Last Mach* TCP or if $z$ is less than the altitude associated with the *First CAS* TCP, then an error should be noted.

Find where $z$ first occurs.

*i = index number of the first waypoint + 1*

*finished = false*

*while ( (i < index number of the last waypoint) and (finished = false))*

$if\ (Altitude_i > z)\ i = i + 1$

*else finished = true*

Find the distance to this altitude.

$x = Altitude_{i-1} - Altitude_i$

$if\ (x \leq 0)\ ratio = 0$

$else\ ratio = (z - Altitude_i) / x$

$d = ratio * (DTG_{i-1} - DTG_i) + DTG_i$

Compute the ground track at distance $d$ along the trajectory and save it as *Saved Ground Track*.

*Saved Ground Track = GetTrajGndTrk( d )*

Insert a new TCP at location $i$ in the TCP list. The TCP is inserted between $TCP_{i-1}$ and $TCP_i$ from the original list. The function *InsertWaypoint* should be appropriate for the actual data structure implementation of this function.

*InsertWaypoint( i )*

*Mark this TCP as the Mach-to-CAS transition TCP.*

Add the data for this new TCP.

*Crossing* $Mach_i$ *= Last Mach*

*Crossing* $CAS_i$ *= First CAS*

*CAS* $Rate_i$ *= CAS Rate*

$DTG_i = d$

$Altitude_i = z$

*Crossing* $Angle_i$ *= Crossing* $Angle_{i+1}$

*Ground* $Track_i$ *= Saved Ground Track*

$Mach_i$ *= Last Mach*

$CAS_i$ *= First CAS*

Compute and add the wind data at distance *d* along the path to the data of $TCP_i$.

*GenerateWptWindProfile(* $DTG_i$, $TCP_i$ *)*

*Mark all TCPs from the first TCP (*$TCP_{first\ waypoint}$*) to* $TCP_{i-1}$ *as Mach TCPs.*

## Compute Altitude / CAS Restriction TCP

If an altitude / CAS restriction is required, the *Compute Altitude / CAS Restriction TCP* function computes the altitude / CAS restriction point and insert an altitude / CAS TCP. This is the (U.S.) point where the trajectory transitions through 10,000 ft and a 250 kt restriction is required. This function is only performed if the previously computed flag *Need10KRestriction* is true. The function determines the along-path distance where this altitude / CAS occurs on the profile. A TCP is then inserted into the TCP list at this point. The restriction values are *Descent Crossing Altitude* and *Descent Crossing CAS*.

Find the first TCP that is below the *Descent Crossing Altitude* in the list.

*i = index number of the first waypoint*

*k = i*

*fini = false*

*while ( (i <index number of the last waypoint) and (fini = false) )*

*if (*$Altitude_i$ *< Descent Crossing Altitude)*

*k = i*

*fini = true*

$i = i + 1$

Find the last CAS restriction prior to the first waypoint below *Descent Crossing Altitude*.

$i = k - 1$

*fini = false*

*Last CAS = 0*

*while ( (i > 0) and (fini = false) )*

*if (Crossing* $CAS_i > 0$*)*

*Last CAS = Crossing* $CAS_i$

*fini = true*

$i = i - 1$

Determine if an altitude or CAS TCP is required. If it is, add it.

*if ( (*$TCP_k$ *is a Mach segment) and (Last CAS > Descent Crossing CAS) )*

$i = k;$

Find the distance to this altitude.

$x = Altitude_{i-1} - Altitude_i$

*if* $(x \leq 0)$ *ratio = 0*

*else ratio = (Descent Crossing Altitude -* $Altitude_i$*) / x*

$d = ratio * (DTG_{i-1} - DTG_i) + DTG_i$

Compute the ground track at distance *d* along the trajectory and save it as *Saved Ground Track*.

*Saved Ground Track = GetTrajGndTrk( d )*

Insert a new TCP at location *i* in the TCP list. The TCP is inserted between $TCP_{i-1}$ and $TCP_i$ from the original list. The function *InsertWaypoint* should be appropriate for the actual data structure implementation of this function.

*InsertWaypoint( i )*

Mark this TCP as the altitude / CAS restriction TCP.

$VSegType_i$ *= altitude CAS restriction*

$TurnType_i$ *= no turn*

Add the data for this new TCP.

$Crossing\ Mach_i = 0$

$Crossing\ CAS_i = Descent\ Crossing\ CAS$

Use a high value, arbitrary CAS rate.

$CAS\ Rate_i = 0.75\ kt\ /\ sec$

$DTG_i = d$

$Altitude_i = Descent\ Crossing\ Altitude$

$Crossing\ Angle_i\ = Crossing\ Angle_{i+1}$

*Set the Mach flag for* $TCP_i$ *to false*

$Ground\ Track_i = Saved\ Ground\ Track$

$Mach_i = 0$

$CAS_i = Descent\ Crossing\ CAS$

Compute and add the wind data at distance *d* along the path to the data of $TCP_i$.

$GenerateWptWindProfile(\ DTG_i,\ TCP_i\ )$

## Test for Altitude / CAS Restriction Requirement

The *Test for Altitude / CAS Restriction Requirement* function determines if the addition of an altitude / CAS restriction point is required. This is the (U.S.) point where the trajectory transitions through 10,000 ft and a 250 kt restriction is required. This function determines the value of the *Need10KRestriction* flag. The function can only be called after an initial, preliminary trajectory has been generated. The restriction values are *Descent Crossing Altitude* and *Descent Crossing CAS*.

*Need10KRestriction = false*

*if ( (Descent Crossing Altitude > 0) and (Descent Crossing CAS > 0) ) ok = true*

*else ok = false*

If we don't start above 10,000ft, skip this whole routine.

*if (ok and (*$Altitude_{first\ waypoint}$ *> Descent Crossing Altitude))*

Find the first point below *Descent Crossing Altitude*

*fini = false*

*i = 0*

*while ( (i <index number of the last waypoint) and (fini = false) )*

*if* ($Altitude_i$ *< Descent Crossing Altitude)*

Find the distance to this altitude.

$x = Altitude_{i-1} - Altitude_i$

*if* $(x \leq 0)$ *ratio = 0*

*else ratio =* $(Descent\ Crossing\ Altitude - Altitude_i) / x$

$s = ratio * (CAS_{i-1} - CAS_i) + CAS_i$

*if (s > (Descent Crossing Cas + 2) ) Need10KRestriction = true*

*fini = true*

*i = i + 1*

## Compute TCP Speeds

The *Compute TCP Speeds* function is similar to *Compute TCP Altitudes* in its design. Beginning with the last waypoint, this function computes the Mach or CAS at each previous TCP and inserts any additional speed TCPs that may be required to denote a change in the speed profile. The function uses the current speed constraint, searches backward for the previous constraint, and then computes the distance required to meet this previous constraint. The speeds for all of the TCPs within this distance are computed and added to the data for the TCPs. If the along-path distance to meet the previous constraint is not at a TCP, a new speed VTCP is inserted at this distance. This function invokes two secondary functions, described in the subsequent text, with the invocation dependent on the constraint speed, whether it is a Mach or a CAS value. This function is performed in the following steps:

Set the current constraint index number, *cc*, equal to the index number of the last waypoint,

*cc = index number of the last waypoint*

The speed of the first waypoint is set to its crossing speed.

*if* $(Crossing\ Mach_{first\ waypoint} > 0)$

$Mach_{first\ waypoint} = Crossing\ Mach_{first\ waypoint}$

$CAS_{first\ waypoint} = MachToCas(\ Mach_{first\ waypoint},\ Altitude_{first\ waypoint})$

*else*

$CAS_{first\ waypoint} = Crossing\ CAS_{first\ waypoint}$

$Mach_{first\ waypoint} = CasToMach(\ CAS_{first\ waypoint},\ Altitude_{first\ waypoint})$

The speed of the last waypoint is set to its crossing speed,

$CAS_{cc} = Crossing\ CAS_{cc}$.

A flag signifying that Mach segment computation has begun is set to false,

*Doing Mach = false*

*While (cc > index number of the first waypoint)*

Set the Mach flag if the current TCP is the Mach-to-CAS transition point.

*if ($TCP_{cc}$ = Mach Transition CAS) Doing Mach = true*

*if (Doing Mach) ComputeTcpMach(cc)*

*else ComputeTcpCas(cc)*

*end of while cc > index number of the first waypoint*

## Compute Secondary Speeds

The *Compute Secondary Speeds* function adds the Mach values to CAS TCPs, the CAS values to Mach TCPs, and the ground speed values to all TCPs. This function is performed in the following steps:

*Doing Mach = false*

Working backwards from the runway, compute the relevant speeds.

*for (i = index number of the last waypoint; i ≥ index number of the first waypoint; i = i - 1)*

Set the flag if the current TCP is the Mach-to-CAS transition point.

*if ($TCP_i$ = Mach Transition CAS) Doing Mach = true*

*if (Doing Mach) $Cas_i$ = MachToCas( $Mach_i$, $Altitude_i$ )*

*else $Mach_i$ = CasToMach( $Cas_i$, $Altitude_i$ )*

Compute the ground track.

*if (i = index number of the first waypoint) track = Ground $Track_i$*

*else if (WptInTurn(i) or ($TCP_i$ = turn-exit)) track = Ground $Track_i$*

*else track = Ground $Track_{i-1}$*

Compute the ground speed. This also requires the computation of the wind at this point.

*InterpolateWindWptAltitude( Wind $Profile_i$, $Altitude_i$, Wind Speed, Wind Direction )*

*Ground $Speed_i$ = ComputeGndSpeedUsingTrack ( $Cas_i$, track, $Altitude_i$, Wind Speed, Wind Direction )*

*end of for (i = index number of the last waypoint; i ≥ index number of the first waypoint; i = i - 1)*

### Update Turn Data

The *Update Turn Data* function computes the turn data for each turn waypoint and modifies the associated waypoint's turn data sub-record. This function performs as follows:

*KtsToFps = 1.69*

*Nominal Bank Angle = 22*

*index = index number of the first waypoint + 1*

*while (index < index number of the last waypoint)*

Find the next input waypoint with a turn.

*while ( (index < index number of the last waypoint) and ( (*$TCP_{index} \neq$ *input waypoint) or (not WptInTurn( index )))) index = index + 1*

If there are no errors and there is a turn of more than 3-degrees, compute the turn data.

*if (index < index number of the last waypoint)*

Find the start of the turn.

*i = index - 1*

*while (*$TCP_i \neq$ *turn-entry) i = i - 1*

*start = i*

The following are all approximations and are based on a general, constant radius turn.

The start of turn to the midpoint data is as follows, noting that the ground speeds for all points must be valid at this point.

The overall distance *d* for this part of the turn is,

$d = DTG_{start} - DTG_{index}$

The special case with 0 distance between the points is,

*if* $(d \leq 0)$ $AvgGsFirstHalf = (Ground\ Speed_{start} + Ground\ Speed_{index}) / 2$

*else*

The overall average ground speed is computed as follows, noting that it is the sum of segment distance / overall distance * average segment ground speed.

*AvgGsFirstHalf = 0*

*for (j = start; j* $\leq$ *(index - 1); j = j + 1)*

$dx = DTG_j - DTG_{j+1}$

$AvgGsFirstHalf = AvgGsFirstHalf + (dx / d)$
$* (Ground\ Speed_j + Ground\ Speed_{j+1}) / 2$

Now, find the end of the turn.

$i = index + 1$

$while\ (TCP_i \neq turn\text{-}exit)\ i = i + 1$

$end = i$

Now, find the midpoint to the end of the turn.

The overall distance for this part of the turn is,

$d = DTG_{index} - DTG_{end}$

Test for the special case, 0 distance between the points.

$if\ (d \leq 0)$

$AvgGsLastHalf = (Ground\ Speed_{index} + Ground\ Speed_{end}) / 2$

$else$

Compute the overall average ground speed noting that it is the sum of segment distance / overall distance * average segment ground speed.

$AvgGsLastHalf = 0$

$for\ (j = index; j \leq (end - 1); j = j + 1)$

$dx = DTG_j - DTG_{j+1}$

$AvgGsLastHalf = AvgGsLastHalf + (dx / d) *$
$(Ground\ Speed_j + Ground\ Speed_{j+1}) / 2$

$end\ of\ for\ (j = index; j \leq (end - 1); j = j + 1)$

$end\ of\ else\ if\ (d \leq 0)$

$full\ turn = DeltaAngle(Ground\ Track_{start}, Ground\ Track_{end})$

$half\ turn = full\ turn / 2$

Compute the outputs from the average ground speed.

$Average\ Ground\ Speed = (AvgGsFirstHalf + AvgGsLastHalf) / 2$

Save the ground speed data in the turn data for this waypoint.

$Turn\ Data\ Average\ Ground\ Speed_{index} = Average\ Ground\ Speed$

Compute the turn radius and associated data. This set of calculations is not performed if the waypoint is a special, RF center-of-turn turn waypoint.

*if* $(Wpt_i \neq rf\text{-}turn\text{-}center)$

The general equation is turn rate = c tan(bank angle) / v. If the bank angle is a constant, turn rate = c0 / v. The *Nominal Bank Angle* = 22 degrees.

$c0 = 57.3 * 32.2 / KtsToFps * tangent(Nominal\ Bank\ Angle)$

$w = c0 / Average\ Ground\ Speed$

The time to make the turn is,

$Turn\ Data\ Turn\ Time_{index} = |full\ turn| / w$

The turn radius is,

$$Turn\ Data\ Turn\ Radius_{index} = (57.3 * KtsToFps * Average\ Ground\ Speed) / (6076 * w)$$

The along-path distance for the turn is,

$Turn\ Data\ Path\ Distance_{index} = |full\ turn| * Turn\ Data\ Turn\ Radius_{index} / 57.3$

*else*

These are the data for an RF turn. The along-path distance for the turn is,

$Turn\ Data\ Path\ Distance_{index} = |full\ turn| * Turn\ Data\ Turn\ Radius_{index} / 57.3$

The time to make the turn is,

$$Turn\ Data\ Turn\ Time_{index} = Turn\ Data\ Path\ Distance_{index} / Average\ Ground\ Speed * 3600$$

Save the turn data for the first half of the turn, denoted by the "1" in the variable name.

$Turn\ Data\ Cas1_{index} = CAS_{start}$

$Turn\ Data\ Average\ Ground\ Speed1_{index} = AvgGsFirstHalf$

$Turn\ Data\ Track1_{index} = Ground\ Track_{start}$

The *Straight Distance* values are the distances from the turn-entry TCP to the waypoint and from the waypoint to the turn-exit TCP. See the example in figure 6.

$Turn\ Data\ Straight\ Distance1_{index} = Turn\ Data\ Turn\ Radius_{index} * tangent(|half\ turn|)$

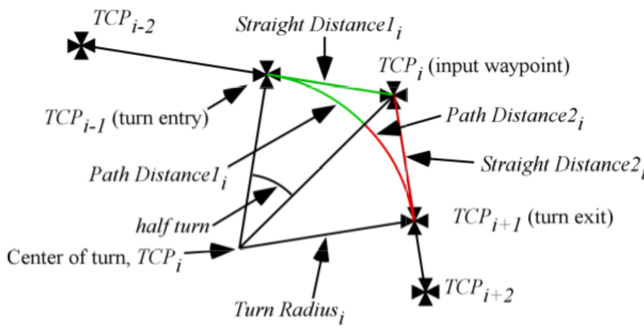

Figure 6. Turn distances for waypoint$_i$.

The Path Distance values are the along-the-path distances from the turn-entry TCP to a point one-half way along the turn and from this point to the turn-exit TCP. See the example in figure 6.

$Turn\ Data\ Path\ Distance1_{index} = |half\ turn| * Turn\ Data\ Turn\ Radius_{index} / 57.3$

Compute the midpoint waypoint data. This set of calculations is not performed if the waypoint is a special, RF center-of-turn waypoint.

*if* $(Wpt_i \neq rf\text{-}turn\text{-}center)$

$w = c0 / AvgGsFirstHalf$

$Turn\ Data\ Turn\ Time1_{index} = |half\ turn| / w$

*else*

These are the data for an RF turn.

$Turn\ Data\ Turn\ Time1_{index} = Turn\ Data\ Path\ Distance1_{index} / AvgGsFirstHalf * 3600$

The data for the midpoint to the end of the turn, denoted by the "2" in the variable name, are as follows:

$Turn\ Data\ Cas2_{index} = CAS_{end}$

$Turn\ Data\ Average\ Ground\ Speed2_{index} = AvgGsLastHalf$

$Turn\ Data\ Track2_{index} = Ground\ Track_{end}$

The distances for the second half of the turn are the same as for the first, but their calculations are recomputed here for clarity.

$Turn\ Data\ Straight\ Distance2_{index} = Turn\ Data\ Turn\ Radius_{index} * tangent(|half\ turn|)$

$Turn\ Data\ Path\ Distance2_{index} = |half\ turn| * Turn\ Data\ Turn\ Radius_{index} / 57.3$

Compute the data for the last half of the turn. Again, this set of calculations is not performed if the waypoint is a special, RF center-of-turn waypoint.

*if* ($Wpt_i \neq$ *rf-turn-center)*

$w = c0 / AvgGsLastHalf$

$Turn\ Data\ Turn\ Time2_{index} = |half\ turn| / w$

*else*

These are the data for an RF turn.

$Turn\ Data\ Turn\ Time2_{index} = Turn\ Data\ Path\ Distance2_{index} / AvgGsLastHalf * 3600$

The *DTG* values are as follows:

$DTG_{start} = DTG_{index} + Turn\ Data\ Path\ Distance1_{index}$

$DTG_{end} = DTG_{index} - Turn\ Data\ Path\ Distance2_{index}$

Since the turn waypoints have been moved, the wind data need to be updated for the new locations.

*if* ($TCP_{start} \neq$ *input waypoint) GenerateWptWindProfile(* $DTG_{start}$, $TCP_{start}$ *)*

*if* ($TCP_{end} \neq$ *input waypoint) GenerateWptWindProfile(* $DTG_{end}$, $TCP_{end}$ *)*

*end of if (index < index number of the last waypoint)*

*index = index + 1*

*end of while (index < index number of the last waypoint)*

## Delete TCPs

The *Delete TCPs* function deletes the altitude, speed, and Mach-to-CAS TCPs. The remaining TCPs will only consist of input waypoints, turn-entry, and turn-exit TCPS. This function also removes any flags that associate any remaining TCPs with a speed or altitude change, e.g., a waypoint marked as the 10,000 ft, 250 kt restriction.

## Update DTG Data

The *Update DTG Data* function is performed after the turn data have been updated and the VTCPs have been deleted. Only input, turn-entry, and turn-exit TCPs should be in the list at this time. If the input test flag, *TestOnly*, is true, then only the testing portions of this function are used.

*if (TestOnly = false)* $DTG_{first\ waypoint} = 0$

*i = index number of the last waypoint*

*while (i > index number of the first waypoint)*

Determine if there is a turn at either end and adjust accordingly.

*if (WptInTurn( i ))*

*if (TestOnly = false)* $DTG_{i-1} = DTG_i$ *+ Turn Data Path Distance1*$_i$

The following is the difference between going directly from the waypoint to going along the curved path.

*PriorDistanceOffset = Turn Data Straight Distance1*$_i$ *- Turn Data Path Distance1*$_i$

*else PriorDistanceOffset = 0*

Find the next input waypoint.

$n = i - 1$

*while* ($TCP_n \neq$ *input waypoint*) $n = n - 1$

*if (WptInTurn( n ))*

The following is the difference between going directly from the waypoint to going along the curved path.

*DistanceOffset = Turn Data Straight Distance2*$_n$ *- TurnData.PathDistance2*$_n$

The DTG to the input waypoint is then:

*if (TestOnly = false)* $DTG_n$ *= (Center to Center Distance*$_i$ *- PriorDistanceOffset - DistanceOffset) +* $DTG_i$

If the *DistanceOffset* is greater than *Center to Center Distance*$_i$, then the turn is too big.

*if (DistanceOffset > Center to Center Distance*$_i$*) mark this as an error condition*

The turn-exit DTG is then,

*if (TestOnly = false)* $DTG_{n+1} = DTG_n$ *- Turn Data Path Distance2*$_n$

*else if (TestOnly = false)*

The next waypoint is not in a turn.

$DTG_n$ *= Center to Center Distance*$_i$ *- PriorDistanceOffset +* $DTG_i$

$i = n$

*end of while (i > 0)*

## Check Turn Validity

The *Check Turn Validity* function is performed after the turn data have been updated and the VTCPs have been deleted. Only input, turn-entry, and turn-exit TCPs should be in the list at this time. The function simple checks that there are no turns within turns by examining the DTG values.

*for (i = index number of the first waypoint; i < index number of the last waypoint; i = i + 1)*

*if ($DTG_i < DTG_{i+1}$) mark this as an error condition*

## Recover the Initial Mach Segments

This function, *Recover the Initial Mach Segments*, attempts to recover the Mach portion of the trajectory if the initial segments should be Mach but have been internally converted to CAS in the function *Meet Cruise CAS Waypoint Restriction*. This function uses the Mach value that was saved at the start of this program from the first waypoint of the original route. This saved Mach value, *First Waypoint Mach*, is compared to the Mach equivalent value of the CAS at the initial waypoints and if these Mach values are the same, these waypoints are marked as Mach segments instead of CAS segments.

Only perform this function if the calculated trajectory does not start with a Mach segment but the original route does start with a Mach value.

*if ((Mach Segment$_{\text{index number of the first waypoint}}$ = false) and (First Waypoint Mach ≠ 0 ))*

*Mach = CasToMach( Crossing CAS$_{\text{index number of the first waypoint}}$, Altitude$_{\text{index number of the first waypoint}}$)*

*if (Mach ≈ First Waypoint Mach)*

*fini = false*

*i = index number of the last waypoint*

*FirstCas = Crossing CAS$_{\text{index number of the first waypoint}}$*

If there is no Mach transition altitude set, set the transition values.

*if (Mach Transition Altitude = 0)*

*Mach Descent Mach = First Waypoint Mach*

*Mach Transition Mach = First Waypoint Mach*

*Mach Transition Cas = FirstCas*

*Mach Transition Altitude = Altitude$_{\text{index of first waypoint}}$*

*while (( i < (index number of the last waypoint - 1)) and ( fini = false))*

Test that the CAS computed for the waypoint is the same as the *FirstCas*, that except for the first waypoint that there is not speed crossing condition at the waypoint, and that the altitude computed for the waypoint is the same as the altitude for the first waypoint.

*if (( $Cas_i$ = FirstCas) and ( (i = index number of the last waypoint) or*
*((Crossing $Mach_i$ = 0) and ( Crossing $CAS_i$ = 0))) and*

*($Altitude_i$ = Crossing $Altitude_{index\ number\ of\ the\ first\ waypoint}$))*

If the previous conditions are turn, set this waypoint as a Mach segment.

*Mach $Segment_i$ = true*

Change the speed crossing values for the first waypoint.

*if ( Crossing $CAS_i > 0$)*

*Crossing $CAS_i = 0$*

*Crossing $Mach_i$ = First Waypoint Mach*

*end of if (( $Cas_i$ = FirstCas)...)*

*else fini = true*

*$i = i + 1$*

## Insert CAS Descent VTCPs

This function inserts vertical TCPs between constant CAS descent waypoints to improve the TAS estimation when using the data provided by this algorithm. This updating occurs at 3,000 ft intervals.

*Update Altitude = 3000*

Find the first CAS point.

*$j = 0$*

*while ( (Mach $Segment_i$ = true) and (j < index number of the last waypoint)) $j = j + 1$*

*for ( i = j; i < (index number of the last waypoint - 1); $i = i + 1$)*

*DeltaZ = $Altitude_i$ - $Altitude_{i+1}$*

Update at 3000 ft intervals but skip the update if the waypoint is within 500 ft of the test altitude.

*if ( (DeltaZ ≥ (Update Altitude + 500)) and ($Cas_i \approx Cas_{i+1}$))*

*z = $Altitude_i$ - Update Altitude*

*dx = $DTG_i$ - $DTG_{i+1}$*

*a = arctangent2 (DeltaZ, 6076 * dx)*

*d = $DTG_i$ - Update Altitude / tan(a) / 6076*

Compute the ground track at distance *d* along the trajectory and save it as *Saved Ground Track*.

$Saved\ Ground\ Track = GetTrajGndTrk(\ d\ )$

$k = i + 1$

Insert a new VTCP at location k in the TCP list. The VTCP is inserted between $TCP_{k-1}$ and $TCP_k$ from the original list. The function *InsertWaypoint* should be appropriate for the actual data structure implementation of this function.

$InsertWaypoint(\ k\ )$

Update the waypoint-type data in the new waypoint.

$WptType_k = VTCP$

$VSegType_k = TAS\ adjustment$

$TurnType_k = no\ turn$

Update the crossing data in the new waypoint.

$Crossing\ Mach_k = 0$

$Crossing\ CAS_k = 0$

$Crossing\ Rate_k = 0$

$CAS_k = CAS_{k+1}$

$DTG_k = d$

$Altitude_k = z$

$Mach_k = CasToMach(\ CAS_k,\ Altitude_k)$

$Mach\ Segment_k = false$

$Crossing\ Angle_k = Crossing\ Angle_{k+1}$

$Ground\ Track_k = Saved\ Ground\ Track$

Compute and add the wind data at this waypoint.

$GenerateWptWindProfile(\ DTG_k,\ TCP_k)$

Compute the wind at the waypoint altitude and then waypoint's ground speed.

$InterpolateWindWptAltitude(\ Wind\ Profile_k,\ Altitude_k,\ Ws,\ Wd\ )$

$Ground\ Speed_k = ComputeGndSpeedUsingTrack(\ CAS_k,\ Ground\ Track_{k-1},\ Altitude_k,\ Ws,\ Wd\ )$

## Compute TCP Times

The function *Compute TCP Times* calculates the time to each TCP. The calculations begin at the runway (the last waypoint), working backwards, and compute the TTG to each TCP.

$TTG_{\text{index number of the last waypoint}} = 0$

*for (i = index number of the last waypoint; i > index number of the first waypoint; i = i - 1)*

*Average Ground Speed* $= (\text{Ground Speed}_{i-1} + \text{Ground Speed}_i) / 2$

$x = DTG_{i-1} - DTG_i$

Test for an error condition where the distance is less than 0.

*if (x < 0)*

If the distance is close to 0, e.g., within 200 ft., set the distance to the previous and ignore the error.

*if* $(x \geq (-200 / 6076))$

$DTG_i = DTG_{i-1}$

$x = 0$

Allow a larger margin of error for an RF turn.

*else if* $( (x \geq -0.05)$ *and* $(\text{TurnType}_i = \text{turn-entry}) )$ *and* $(\text{Center Of Turn Latitude}_i \neq 0 ))$

$DTG_i = DTG_{i-1}$

$x = 0$

*else mark this as an error condition*

*Delta Time = 3600 * x / Average Ground Speed*

$TTG_{i-1} = TTG_i + \text{Delta Time}$

## Compute TCP Latitude and Longitude Data

With the exception of the input waypoints, the *Compute TCP Latitude and Longitude Data* function computes the latitude and longitude data for all of the TCPs.

*In Turn = false*

*Last Base = index number of the first waypoint*

*Next Input = index number of the first waypoint*

*Turn Index = index number of the first waypoint*

*Turn is Clockwise = true*

*Turn Adjustment = 0*

*Base Latitude =* $Latitude_{Last\ Base}$

*Base Longitude =* $Longitude_{Last\ Base}$

*for (i = index number of the first waypoint; i ≤ index number of the last waypoint; i = i + 1)*

*if (*$TCP_i$ *= turn-entry)*

*Turn Adjustment = 0*

*InTurn = True;*

Find the major waypoint for this turn.

*Next Input = i + 1*

*while ( (*$TCP_{Next\ Input}$ *≠ input waypoint) and (Next Input ≤ index number of the last waypoint) )*
*Next Input = Next Input + 1*

*Turn Index = Next Input*

*a = DeltaAngle(Ground* $Track_i$*, Ground* $Track_{Next\ Input}$*)*

*x = Turn Data Turn* $Radius_{Turn\ Index}$ */ cosine(a)*

*if (a > 0) Turn Clockwise =true*

*else Turn Clockwise = false*

*if (Turn Clockwise) a1 = Ground* $Track_{Turn\ Index}$ *+ 90*

*else a1 = Ground* $Track_{Turn\ Index}$ *- 90*

Now compute the relative latitude and longitude values. The function *RelativeLatLon* is described in a subsequent section.

*RelativeLatLong(*$Latitude_{Turn\ Index}$*,* $Longitude_{Turn\ Index}$*, a1, x), returning Center Latitude and Center Longitude*

*end of if (*$TCP_i$ *= turn-entry)*

*if (In Turn)*

*Turn Adjustment = 0*

*if (Turn Clockwise) a1 = Ground* $Track_i$ *- 90*

*else a1 = Ground* $Track_i$ *+ 90*

*if ($TCP_i$ = input waypoint)*

*Turn Data Center Latitudei = Center Latitude*

*Turn Data Center Longitudei = Center Longitude*

*RelativeLatLong(Center Latitude, Center Longitude, a1, Turn Data Turn Radius$_{Turn\ Index}$), returning Turn Data Latitude$_i$ and Turn Data Longitude$_i$*

*end of if ($TCP_i$ = input waypoint)*

*else RelativeLatLon(Center Latitude, Center Longitude, a1, Turn Data Turn Radius$_{Next\ Input}$), returning Latitude$_i$ and Longitude$_i$*

*if ($TCP_i$ = turn-exit)*

*Turn Adjustment = Turn Data Straight Distance2$_{Turn\ Index}$ - Turn Data Path Distance2$_{Turn\ Index}$*

*In Turn = false*

*Last Base = Next Input*

*Base Latitude = Latitude$_{Last\ Base}$*

*Base Longitude = Longitude$_{Last\ Base}$*

*end of if (In Turn)*

*else*

*if ($TCP_i$ = input waypoint)*

*Turn Adjustment = 0*

*Last Base = i*

*Base Latitude = Latitude$_{Last\ Base}$*

*Base Longitude = Longitude$_{Last\ Base}$*

*else*

*RelativeLatLong(Base Latitude, Base Longitude, Ground Track$_{i-1}$, DTG$_{Last\ Base}$ - DTG$_i$ + Turn Adjustment), returning Latitude$_i$ and Longitude$_i$*

*end of for (i = index number of the first waypoint; i ≤ index number of the last waypoint; i = i + 1)*

## Description of Secondary Functions

The secondary functions are listed in alphabetical order. Note that standard aeronautical functions, such as CAS to Mach conversions, *CasToMach*, are not expanded in this document but may be found numerous references, e.g., reference 22. It may also be of interest to include atmospheric temperature or temperature deviation in the wind data input and calculate the temperature at the TCP crossing altitudes to improve the calculation of the various speed terms.

### BodDecelerationDistance

The function BodDecelerationDistance estimates the distance required for the special case of a deceleration to a CAS restricted waypoint from the Mach-to-CAS transition. This function is invoked from *HandleDescentAccelDecel*, which passes in the index number for the bottom-of-descent (TOD) waypoint, *BodIndex*, the Mach transition to CAS altitude, *MachTransitionAlt*, and the CAS at the Mach transition to CAS, *TransitionCas*. The function returns the distance from the index point of the deceleration, *Distance*.

Estimate the distance to the new Mach value. Begin by finding the time to do the deceleration.

$t = (TransitionCas - Crossing\ CAS_{BodIdx}) / Crossing\ Rate_{BodIdx}$

Compute the wind speed and direction at the current altitude.

$InterpolateWindWptAltitude(\ Wind\ Profile_{BodIdx},\ Altitude_{BodIdx},\ Ws,\ Wd\ )$

Calculate the ground track at the current point.

$if\ (WptInTurn(\ BodIdx\ ))\ track = Ground\ Track_{BodIdx-1}$

$else\ track = Ground\ Track_{BodIdx}$

Calculate the ground speed over this segment.

$BodGs = ComputeGndSpeedUsingTrack(\ Crossing\ CAS_{BodIdx},\ track,\ Altitude_{BodIdx},\ Ws,\ Wd\ )$

$DescentGs = ComputeGndSpeedUsingTrack(\ TransitionCas,\ track,\ MachTransitionAlt,\ Ws,\ Wd\ )$

Calculate the average groundspeed, *AvgGS*.

$AvgGs = (BodGs + DescentGs) / 2$

The distance estimate is *AvgGs * t* .

$Distance = AvgGs * t / 3600$

### ComputeGndSpeedUsingMachAndTrack

The *ComputeGndSpeedUsingMachAndTrack* function computes a ground speed from track angle (versus heading), *track*, Mach, *Mach*, altitude, *Altitude*, and wind data, *Wind Speed* and *Wind Direction*.

$CAS = MachToCas(\ Mach,\ Altitude\ )$

$Ground\ Speed = ComputeGndSpeedUsingTrack(\ CAS,\ track,\ Altitude,\ Wind\ Speed,\ Wind\ Direction\ )$

## ComputeGndSpeedUsingTrack

The *ComputeGndSpeedUsingTrack* function computes a ground speed from track angle (versus heading), *track*, CAS, *CAS*, altitude, *Altitude*, and wind data, *Wind Speed* and *Wind Direction*.

*b = DeltaAngle( track, Wind Direction )*

*if (CAS ≤ 0) r = 0*

*else r = (Wind Speed / CasToTas Conversion( CAS, Altitude )) * sine(b)*

Limit the correction to something reasonable.

*if (|r| > 0.8) r = 0.8 * r / |r|*

*heading = track + arcsine(r)*

*a = DeltaAngle( heading, Wind Direction )*

*TAS = CasToTas Conversion( CAS, Altitude )*

$Ground\ Speed = (Wind\ Speed^2 + TAS^2 - 2 * Wind\ Speed * TAS * cosine(a)\ )^{0.5}$

## ComputeGndTrk

The *ComputeGndTrk* function computes the ground track at the along-path distance equal to *distance*., where distance must lie between $TCP_{i-1}$ and $TCP_{i+1}$. It is assumed that the value for $Ground\ Track_i$ is invalid. The function uses a linear interpolation based on $DTG_{i-1}$ and $DTG_{i+1}$, with the index value *i* input into the function and where the distance, *distance,* must lie between these points.

$d = DTG_{i-1} - DTG_{i+1}$

$if\ (d \leq 0)\ Ground\ Track = Ground\ Track_{i-1}$

*else*

$a = (1 - (distance - DT_{i+1}) / d) * \ DeltaAngle(Ground\ Track_{i-1}, Ground\ Track_{i+1})$

$Ground\ Track = Ground\ Track_{i-1} + a$

## ComputeTcpCas

The index variable *cc* is passed into and out of the *ComputeTcpCas* function. Beginning with the last waypoint, this function computes the CAS at each previous TCP and inserts any additional speed TCPs that may be required to denote a change in the speed profile. The function uses the current speed constraint, searches backward for the previous constraint, and then computes the distance required to meet this previous constraint. The speeds for all of the TCPs within this distance are computed and added to the data for the TCPs. If the along-path distance to meet the previous constraint is not at a TCP, a new speed VTCP is inserted at this distance. Because there is no general closed form solution to compute distances to meet the deceleration constraints, an iterative technique is used in this function. This function is performed in the following steps:

*While ((cc > index number of the first waypoint) and* ($TCP_{cc} \neq$ *Mach Transition CAS))*

Determine if the previous constraint cannot be met.

*If* $(CAS_{cc} > Crossing\ CAS_{cc})$

*If this is the last pass through the algorithm, mark this as an error condition*

$CAS_{cc} = Crossing\ CAS_{cc}$

Find the prior waypoint index number *pc* that has a CAS constraint, e.g., a crossing CAS ($Crossing\ CAS_{pc} \neq 0$). This may not always be the previous (i.e., *cc - 1*) waypoint.

The initial condition is the previous TCP.

$pc = cc - 1$

*while* ( ($pc >$ *index number of the first waypoint*) *and* ($TCP_{pc} \neq$ *Mach Transition CAS*) *and* ($Crossing\ CAS_{pc} = 0$) ) $pc = pc - 1$

Save the previous crossing speed,

$Prior\ Speed = Crossing\ CAS_{pc}$

Save the current crossing speed (*Test Speed*) at $TCP_{cc}$ and the deceleration rate (*Test Rate*) noting that the first and last waypoints always have speed constraints and except for the first waypoint, all constrained speed points must have deceleration rates.

$Test\ Speed = Crossing\ CAS_{cc}$

$Test\ Rate = Crossing\ Rate_{cc}$

Compute all of the TCP speeds between the current TCP and the previous crossing waypoint.

$k = cc$

*while* $k > pc$

If the previous speed has already been reached, set the remaining TCP speeds to the previous speed.

*if* $(Prior\ Speed \leq Test\ Speed)$

*for* $(k = k - 1;\ k > pc;\ k = k - 1)$

$CAS_k = Test\ Speed$

$Mach_k = CasToMach(\ CAS_k,\ Altitude_k)$

Set the speeds at the last test point.

$CAS_{pc} = Test\ Speed$

*if* $(Mach_{pc} = 0)$ $Mach_{pc} = CasToMach(\ CAS_{pc},\ Altitude_{pc})$

*else*

Estimate the distance required to meet the crossing restriction using the winds at the current altitude. This is a first-estimation.

Compute the time to do the deceleration.

$t = (Prior\ Speed - Test\ Speed) / Test\ Rate$

Compute the wind speed and direction at the current altitude.

*InterpolateWindWptAltitude( Wind Profile*$_k$*, Altitude*$_k$*,Wind Speed1, Wind Direction1 )*

The ground track at the current point is,

*if (WptInTurn( k )) Track = Ground Track*$_k$

*else Track = Ground Track*$_{k-1}$

*Current Ground Speed = ComputeGndSpeedUsingTrack( Test Speed, Track, Altitude*$_k$*,Wind Speed1, Wind Direction1 )*

Compute the wind speed and direction at the prior altitude.

*InterpolateWindWptAltitude( Wind Profile*$_{k-1}$*, Altitude*$_k$*,Wind Speed1, Wind Direction1 )*

The ground speed at the prior point.

*Prior Ground Speed = ComputeGndSpeedUsingTrack( Prior Speed, GndTrack*$_{k-1}$*, Altitude*$_{k-1}$*, Wind Speed1, Wind Direction1 )*

*Average Ground Speed = (Prior Ground Speed + Current Ground Speed) / 2*

The distance estimate, *dx*, is *Average Ground Speed * t*.

*dx = Average Ground Speed * t / 3600*

Recalculate the distance required to meet the speed using the previous estimate distance *dx*.

Begin by computing the altitude*, AltD,* at distance *dx.*

*if (Altitude*$_k \geq$ *Altitude*$_{k-1}$*) AltD = Altitude*$_k$

*else*

*AltD = (6076 * dx) * tangent(Crossing Angle*$_k$*) + Altitude*$_k$

*if (AltD* $\geq$ *Altitude*$_{k-1}$*) AltD = Altitude*$_k$

The new distance *x* is $DTG_k + dx$.

Compute the winds at *AltD* and distance *x*.

*InterpolateWindAtDistance( AltD, x, Wind Speed2, Wind Direction2 )*

The track angle at this point, with *GetTrajGndTrk* defined in this section:

*Track2 = GetTrajGndTrk( x )*

The ground speed at altitude *AltD* is then,

*Prior Ground Speed = ComputeGndSpeedUsingTrack( Prior Speed, Track2, AltD, Wind Speed2, Wind Direction2 )*

*Average Ground Speed = (Prior Ground Speed + Current Ground Speed) / 2*

*dx = Average Ground Speed * t / 3600*

If there is a TCP prior to *dx*, compute and insert its speed.

If the distance is very close to the waypoint, just set the speed.

*if ( (*$DTG_{k-1}$ *< (*$DTG_k$ *+ dx + some small value) )*

*if (|*$DTG_{k-1}$ *-* $DTG_k$ *- dx| < some small value)* $CAS_{k-1}$ *= Prior Speed*

*else*

Compute the speed at the waypoint using $v^2 = v_0^2 + 2ax$ to get v.

The headwind at the end point is,

*HeadWind2 = Wind Speed2 * cosine(Wind Direction2 - Ground* $Track_{k-1}$*)*

$dx = DTG_{k-1} - DTG_k$

The value of $CAS_{k-1}$ is computed using function *EstimateNextCas*, described in this section.

$CAS_{k-1}$ *= EstimateNextCas( Test Speed, Current Ground Speed, false, Prior Speed, Head Wind2,* $Altitude_k$*, dx, Crossing* $Rate_{cc}$*)*

Determine if the constraint is met.

*if ( (k-1) = pc)*

Determine the allowable crossing window, accounting for special conditions.

*if (( (pc + 1) < index number of the last waypoint) and (*$VSegType_{pc}$ *= MACH_CAS) ) CrossingWindow = 5*

*else CrossingWindow = 1*

Was the crossing window speed met? If not, set this as an error.

*if (|$CAS_{pc}$ - Crossing $CAS_{pc}$| > CrossingWindow)*
*Mark this as an error condition*

Always set the crossing exactly to the crossing speed.

$CAS_{pc}$ = *Crossing* $CAS_{pc}$

Set the test speed to the computed speed.

*Test Speed* = $CAS_{k-1}$

Back up the index counter to the next intermediate TCP.

$k = k - 1$

*end of if ( ($DTG_{k-1}$ < ($DTG_k$ + dx + some small value) )*

*else*

The constraint occurs between this TCP and the previous TCP. A new VTCP needs to be added at this point.

The along path distance *d* where the VTCP is to be inserted is:

$d = DTG_k + dx$

Save the ground track value at this distance.

*Saved Ground Track = GetTrajGndTrk( d )*

Insert a new VTCP at location *k* in the TCP list. The VTCP is inserted between $TCP_{k-1}$ and $TCP_k$ from the original list. The function *InsertWaypoint* should be appropriate for the actual data structure implementation of this function.

*InsertWaypoint( k )*

Update the data for the new VTCP which is now $TCP_k$.

$WptType_k$ = *VTCP*

*if ($VSegType_k$ = no type) $VSegType_k$ = SPEED*

$TurnType_k$ = *no turn*

$DTG_k = d$

The altitude at this point is computed as follows, recalling that the new waypoint is $TCP_k$:

*if ($Altitude_{k+1} \geq Altitude_{k-1}$) $Altitude_k = Altitude_{k-1}$*

*else* $Altitude_k$ = *(6076 * dx) * tangent(Crossing* $Angle_{k+1}$*)* + $Altitude_{k+1}$

$CAS_k$ = *Prior Speed*

Add the ground track data which must be computed if the new VTCP occurs within a turn. The functions *WptInTurn* and *ComputeGndTrk* are described in this sections.

*if (WptInTurn( k )) Ground* $Track_k$ = *ComputeGndTrk( k, d )*

*else Ground* $Track_k$ = *Saved Ground Track*

Compute and add the wind data at distance *d* along the path to the data of $TCP_k$.

*GenerateWptWindProfile( d,* $TCP_k$*)*

*Test Speed = Prior Speed*

Since $TCP_k$, has now been added prior to *pc*, the current constraint counter *cc* needs to be incremented by 1 to maintain its correct position in the list.

*cc = cc + 1*

*end of while k > pc.*

Now go to the next altitude change segment on the profile.

*cc = k*

*end of while cc > index number of the first waypoint*

## ComputeTcpMach

The index variable cc is passed into and out of the *ComputeTcpMach* function. This function is similar to *ComputeTcpCas* with the exception that the computed Mach rate will need to be recomputed with any change of altitude. Beginning with the last Mach waypoint (the Mach waypoint that is closest to the runway in terms of DTG), this function computes the Mach at each previous TCP and inserts any additional speed TCPs that may be required to denote a change in the speed profile. The function uses the current speed constraint, searches backward for the previous constraint, and then computes the distance required to meet this previous constraint. The speeds for all of the TCPs within this distance are computed and added to the data for the TCPs. If the along-path distance to meet the previous constraint is not at a TCP, a new speed VTCP is inserted at this distance. Because there is no general closed form solution to compute distances to meet the deceleration constraints, an iterative technique is used in this function. This function is performed in the following steps:

*While (cc > index number of the first waypoint)*

Determine if the previous constraint cannot be met.

*If* ($Mach_{cc}$ > *Crossing* $Mach_{cc}$)

*If this is the last pass through the algorithm, mark this as an error condition*

$Mach_{cc}$ = *Crossing* $Mach_{cc}$

Find the prior waypoint index number *pc* that has a Mach constraint, e.g., a crossing Mach (*Crossing* $Mach_{pc} \neq 0$). This may not always be the previous (i.e., *cc - 1*) waypoint.

Initial condition is the previous TCP.

*pc = cc - 1*

*finished = false*

*while ( (pc > index number of the first waypoint) and (*$TCP_{pc} \neq$ *Mach Transition CAS)*
*and (Crossing* $CAS_{pc} = 0$*) ) pc = pc - 1*

Save the previous crossing speed,

*Prior Speed = Crossing* $Mach_{pc}$

Save the current crossing speed (*Test Speed*) at $TCP_{cc}$ and the deceleration rate (*Test Rate*) noting that the first and last waypoints always have speed constraints and except for the first waypoint, all constrained speed points must have deceleration rates.

*Test Speed = Crossing* $Mach_{cc}$

*Test Rate = CasToMach( Altitude*$_{cc}$*, Crossing Rate*$_{cc}$*)*

Compute all of the TCP speeds between the current TCP and the previous crossing waypoint.

*k = cc*

*while k > pc*

If the previous speed has already been reached, set the remaining TCP speeds to the previous speed.

*if (Prior Speed ≤ Test Speed)*

*for (k = k - 1; k > pc; k = k - 1)*

$Mach_k$ *= Test Speed*

$CAS_k$ *= MachToCas(* $Mach_k$*,* $Altitude_k$*)*

*Mark* $TCP_k$ *as a Mach segment.*

Set the speeds at the last test point.

$Mach_{pc}$ *= Test Speed*

$CAS_{pc}$ *= MachToCas(* $Mach_{pc}$*,* $Altitude_{pc}$*)*

*else*

Estimate the distance required to meet the crossing restriction using the winds at the current altitude. This is a first-estimation.

Compute the time to do the deceleration.

*t = (Prior Speed - Test Speed) / Test Rate*

Compute the wind speed and direction at the current altitude.

*InterpolateWindWptAltitude( Wind Profile*$_k$*, Altitude*$_k$*,Wind Speed1, Wind Direction1 )*

The ground track at the current point is,

*if (WptInTurn(k)) Track = Ground Track*$_k$

*else Track = Ground Track*$_{k-1}$

*Current Ground Speed = ComputeGndSpeedUsingMachAndTrack( Test Speed, Track, Altitude*$_k$*, Wind Speed1, Wind Direction1 )*

Compute the wind speed and direction at the prior altitude.

*InterpolateWindWptAltitude( Wind Profile*$_{k-1}$*, Altitude*$_k$*,Wind Speed1, Wind Direction1 )*

The ground speed at the prior altitude and speed is,

*Prior Ground Speed = ComputeGndSpeedUsingMachAndTrack( Prior Speed, GndTrack*$_{k-1}$*, Altitude*$_{k-1}$*, Wind Speed1, Wind Direction1 )*

*Average Ground  Speed = (Prior Ground Speed + Current Ground Speed) / 2*

The distance estimate, *dx*, is *Average Ground  Speed * t*.

*dx = Average Ground Speed * t / 3600*

Compute the distance required to meet the speed using the previous estimate distance *dx*.

Begin by computing the altitude*, AltD,* at distance *dx*.

*if (Altitude*$_k \geq$ *Altitude*$_{k-1}$*) AltD = Altitude*$_k$

*else*

*AltD = (6076 * dx) * tangent(Crossing Angle*$_k$*) + Altitude*$_k$

*if (AltD* $\geq$ *Altitude*$_{k-1}$*) AltD = Altitude*$_k$

Compute the average Mach rate.

*MRate1 = CasToMach( Crossing Rate*$_{cc}$*, Altitude*$_k$*)*

*MRate2 = CasToMach( Crossing Rate*$_{cc}$*, AltD )*

*Test Rate = (MRate1 + MRate2) / 2*

*t = (Prior Speed - Test Speed) / Test Rate*

The new distance *x* is $DTG_k + dx$.

Compute the winds at *AltD* and distance *x*.

*InterpolateWindAtDistance( AltD, x, Wind Speed2, Wind Direction2 )*

The track angle at this point, with *GetTrajGndTrk* defined in this section, is:

*Track2 = GetTrajGndTrk( x )*

The ground speed at altitude *AltD* is then,

*Prior Ground Speed = ComputeGndSpeedUsingMachAndTrack( Prior Speed, Track2, AltD, Wind Speed2, Wind Direction2 )*

*Average Ground Speed = (Prior Ground Speed + Current Ground Speed) / 2*

*dx = Average Ground Speed * t / 3600*

If there is a TCP prior to *dx*, compute and insert its speed.

If the distance is very close to the waypoint, just set the speed.

*if* $( (DTG_{k-1} < (DTG_k + dx + \text{some small value}) )$

*if* $(|DTG_{k-1} - DTG_k - dx| < \text{some small value})$

$Mach_{k-1} = \text{Prior Speed}$

*Mark* $TCP_k$ *as a Mach segment.*

*else*

Compute the speed at the waypoint using $v^2 = v_0^2 + 2ax$ to get v.

The headwind at the end point is,

$HeadWind2 = \text{Wind Speed2} * \text{cosine}(\text{Wind Direction2} - \text{Ground Track}_{k-1})$

$dx = DTG_{k-1} - DTG_k$

Compute the average Mach rate.

$MRate1 = CasToMach(\text{Crossing Rate}_{cc}, Altitude_k)$

$MRate2 = CasToMach(\text{Crossing Rate}_{cc}, Altitude_{k-1})$

*Test Rate = (MRate1 + MRate2) / 2*

The value of $Mach_{k-1}$ is computed using function *EstimateNextmach*, described in this section.

*$Mach_{k-1}$ = EstimateNextMach( Test Speed, Current Ground Speed, Prior Speed, Head Wind2, $Altitude_k$, dx, Test Rate )*

Determine if the constraint is met.

*if ( (k-1) = pc)*

Was the crossing speed met within 0.002 Mach? If not, set this as an error.

*if ($|Mach_{pc}$ - Crossing $Mach_{pc}| > 0.002$) Mark this as an error condition*

Always set the crossing exactly to the crossing speed.

*$Mach_{pc}$ = Crossing $Mach_{pc}$*

Set the test speed to the computed speed.

*Test Speed = $Mach_{k-1}$*

Back up the index counter to the next intermediate TCP.

*k = k - 1*

*end of if ( ($DTG_{k-1}$ < ($DTG_k$ + dx + some small value) )*

*else*

The constraint occurs between this TCP and the previous TCP. A new VTCP needs to be added at this point.

The along path distance *d* where the VTCP is to be inserted is:

*d = $DTG_k$ + dx*

Save the ground track value at this distance.

*Saved Ground Track = GetTrajGndTrk( d )*

Insert a new VTCP at location *k* in the TCP list. The VTCP is inserted between $TCP_{k-1}$ and $TCP_k$ from the original list. The function *InsertWaypoint* should be appropriate for the actual data structure implementation of this function.

*InsertWaypoint( k )*

Update the data for the new VTCP which is now $TCP_k$.

*$WptType_k$ = VTCP*

*if ($VSegType_k$ = no type) $VSegType_k$ = SPEED*

*$TurnType_k$ = no turn*

*$DTG_k = d$*

The altitude at this point is computed as follows, recalling that the new waypoint is $TCP_k$:

*if ($Altitude_{k+1} \geq Altitude_{k-1}$) $Altitude_k = Altitude_{k-1}$*

*else $Altitude_k = (6076 * dx) * tangent(Crossing\ Angle_{k+1}) + Altitude_{k+1}$*

*$Mach_k$ = Prior Speed*

*Mark $TCP_k$ as a Mach segment.*

Add the ground track data which must be computed if the new VTCP occurs within a turn. The functions *WptInTurn* and *ComputeGndTrk* are described in this sections.

*if (WptInTurn( k )) Ground $Track_k$ = ComputeGndTrk( k, d )*

*else Ground $Track_k$ = Saved Ground Track*

Compute and add the wind data at distance *d* along the path to the data of $TCP_k$.

*GenerateWptWindProfile( d, $TCP_k$)*

*Test Speed = Prior Speed*

Since $TCP_k$, has now been added prior to *pc*, the current constraint counter *cc* needs to be incremented by 1 to maintain its correct position in the list.

*cc = cc + 1*

*end of while k > pc.*

Now go to the next altitude change segment on the profile.

*cc = k*

*end of while cc > index number of the first waypoint.*

## DeltaAngle

The *DeltaAngle* function returns angle *a*, the difference between *Angle1* and *Angle2*. The returned value may be negative, i.e., -180 degrees ≥ *DeltaAngle* ≥ 180 degrees.

*a = Angle2 - Angle1*

*Adjust "a" such that $0 \geq a \geq 360$*

*if (a > 180) a = a - 360*

## DoTodAcceleration

The *DoTodAcceleration* function handles the special case when there is an acceleration to the descent Mach at the top-of-descent. This function is invoked from *Add Descent Mach Waypoint*, which passes in the index number for the TOD waypoint, *TodIndex*, and the Mach value at the TOD, *MachAtTod.* The function will insert the Mach acceleration point into the waypoint list if a valid acceleration point can be found.

Make an initial estimate of the distance to the new Mach value. The function *TodAccelerationDistance* returns the values *Valid* , *k*, and *dx*.

*TodAccelerationDistance( TodIdx, MachAtTod, Mach Descent Mach, Valid , k, dx )*

*if (Valid)*

Add the VTCP for the end of the TOD acceleration.

$d = DTG_{TodIdx} - dx$

The original ground track will be needed for the new TCP, so save it.

*OldGroundTrack = GetTrajGndTrk( d )*

Save the wind data at this distance as a temporary TCP.

*GenerateWptWindProfile( d, TemporaryTcp )*

The new waypoint is downstream of the current value of *k*.

$k = k + 1$

*InsertWaypoint( k )*

Note that $Wpt_k$ is the newly created waypoint.

$WptType_k = VTCP$

$TurnType_k = \text{no turn}$

If the new waypoint is not already marked as a special vertical type, mark it as a top-of-descent acceleration point.

*if* $(VSegType_k = NONE)\ VSegType_k = \text{TOD acceleration}$

$DTG_k = d$

Calculate the altitude for the new TCP.

$Altitude_k = Altitude_{TodIdx} - (6076 * dx) * \text{tangent}(Crossing\ Angle_{k+1})$

$Mach_k = Mach\ Descent\ Mach$

$Mach\ Cross_k = Mach\ Descent\ Mach$

$MachSegment_k = true$

Set the *Crossing Rate* to the default value of 0.75.

$Crossing\ Rate_k = 0.75$

Add the appropriate ground track value.

*if (WptInTurn( k )) Ground* $Track_k$ *= ComputeGndTrk( k, d )*

*else Ground* $Track_k$ *= OldGroundTrack*

Copy the wind data from *TemporaryTcp* into $Wpt_k$.

*end of if (Valid)*

else set an error for being unable to accelerate to the descent Mach value.

### EstimateNextCas

*EstimateNextCas* is an iterative function to estimate the CAS value, *CAS*, at the next TCP. Note that there is no closed-form solution for this calculation of CAS. The input variable names described in this function are from the calling routine and are, in order, the target CAS value, *Test CAS*, the ground speed at the estimation starting point, *Current Ground Speed*, an estimation limiting flag, *No Limit Flag*, the CAS at the estimation starting point, *Prior CAS*, the head wind at the estimation starting point, *Head Wind*, the altitude at the estimation starting point, *Altitude*, the distance from the estimation starting point to the point where the CAS is to be estimated, *Distance*, and the deceleration rate to be used in this estimation, *CAS Rate*. Also, the input deceleration value must be greater than 0, *CAS Rate* > 0. The function returns the estimated CAS value.

*Guess CAS = Test CAS*

Set up a condition to get at least one pass.

*d = -10 * Distance*

*size = 1.01 * (Prior CAS - Guess CAS)*

*count = 0*

*if ( (Distance > 0) and (CAS Rate > 0) )*

Iterate a solution. The counter count is used to terminate the iteration if the distance estimation does reach a solution within 0.001 nmi.

*while ( (|Distance - d| > 0.001) and (count < 10) )*

*if (Distance > d) Guess CAS = Guess CAS - size*

*else Guess CAS = Guess CAS + size*

*size = size / 2*

The estimated time t to reach this speed,

*t = (Guess CAS - Test CAS) / CAS Rate*

The new ground speed,

*Gs2 = CasToTas Conversion( guess, Altitude ) - Head Wind*

*d = ((Current Ground Speed + Gs2) / 2) * (t / 3600)*

*count = count + 1*

*end of the while loop*

Limit the computed *CAS*, if necessary.

*if ((NoLimit = false) and (Guess CAS > Prior CAS)) Guess CAS = Prior CAS*

*return Guess CAS*

### EstimateNextMach

*EstimateNextMach* is an iterative function to estimate the Mach value, *Mach*, at the next TCP. Note that there is no closed-form solution for this calculation. The input variable names described in this function are from the calling routine. Also, the input deceleration value must be greater than 0, *Mach Rate* > 0.

*Mach = Test Speed*

Set up a condition to get at least one pass.

*d = -10 * dx*

*size = 1.01 * (Prior Speed - Test Speed)*

*count = 0*

*if ( (dx > 0) and (Test Rate > 0))*

Iterate a solution. The counter count is used to terminate the iteration if the distance estimation does reach a solution within 0.001 n.mi.

*while ( (|d - dx| > 0.001) and (count < 10))*

*if (d > dx) Mach = Mach - size*

*else Mach = Mach + size*

*size = size / 2*

The estimated time t to reach this speed, *t = (Mach - Test Speed) / Test Rate*

The new ground speed,

*CAS = MachToCas( Mach, Altitude )*

*Gs2 = CasToTas Conversion( CAS, Altitude ) - Head Wind2*

*d = ((Current Ground Speed + Gs2) / 2) * (t / 3600)*

*count = count + 1*

*end of the while loop*

Limit the computed *Mach*, if necessary.

*if (Mach > Prior Speed) Mach = Prior Speed*

## GenerateWptWindProfile

The function *GenerateWptWindProfile* is used to compute new wind profile data. This function is a double-linear interpolation using the wind data from the two bounding input waypoints to compute the wind profile for a new VTCP, $TCP_k$. The interpolations are between the wind altitudes from the input data and the ratio of the distance *d* at a point between $TCP_{i-1}$ and $TCP_i$ and the distance between $TCP_{i-1}$ and $TCP_i$. E.g.,

- Find the two bounding input waypoints, $TCP_{i-1}$ and $TCP_i$, between which *d* lies, e.g., $TCP_{i-1} \geq d \geq TCP_i$.

- Using the altitudes from the wind profile of $TCP_i$, compute and temporarily save the wind data at these altitudes using the wind data from $TCP_{i-1}$ (e.g., $Wind\ Speed_{Temporary,\ Altitude1}$).

- Compute the wind speed and wind direction for each altitude using the ratio *r* of the distances. Assuming that the difference between $DTG_{i-1}$ and $DTG_i \neq 0$, and that $DTG_{i-1} > DTG_i$.

$$r = (DTG_{i-1} - d) / (DTG_{i-1} - DTG_i)$$

Iterate the following for each altitude in the profile.

$$Wind\ Speed_{k,\ Altitude1} = (\ (1 - r) * Wind\ Speed_{Temporary,\ Altitude1}) + (r * Wind\ Speed_{i,\ Altitude1})$$

$$a = DeltaAngle(Wind\ Direction_{Temporary,\ Altitude1},\ Wind\ Direction_{i,\ Altitude1})$$

$$Wind\ Direction_{k,\ Altitude1} = Wind\ Direction_{k,\ Altitude1} + (r * a)$$

Figure 7 is an example of the computation data for the wind computation at a 9,000 ft altitude. In this example, $TCP_{i-1}$ has wind data at 10,000 and 8,000 ft and $TCP_i$ has wind data at 9,000 ft.

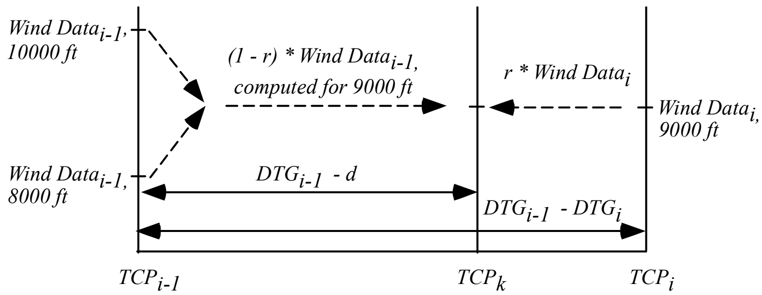


Figure 7. Example for computing a single wind data altitude.

## GetTrajectoryData

The *GetTrajectoryData* function computes the trajectory data at the along-path distance equal to *d* and saves these data in a temporary TCP record. The function uses a linear interpolation based on the DTG values of the two TCPs bounding this distance and the distance *d* to compute the trajectory data at this point.

## GetTrajGndTrk

The *GetTrajGndTrk* function computes the ground track at the along-path distance, *distance*.

*if ( (distance < 0) or (distance > $DTG_{first\ waypoint}$) ) Ground Track = Ground $Track_{first\ waypoint}$*

*else*

Find where distance is on the path.

*i = index number of the last waypoint*

*while (distance > $DTG_i$) i = i -1*

*if (distance = $DTG_i$) Ground Track = Ground $Track_i$*

*else*

*x = DTGi - $DTG_{i+1}$*

*if ($x \leq 0$) r = 0*

*else r = (distance - $DTG_{i+1}$) / x*

*if (r > 1) r = 1*

*dx = (1 - r) * DeltaAngle( Ground $Track_i$, Ground $Track_{i+1}$)*

*Ground Track = Ground $Track_i$ + dx*

**HandleDescentAccelDecel**

The function *HandleDescentAccelDecel* is designed to handle the special case of a Mach acceleration in the descent where the first CAS crossing restriction cannot be met. The calling program provides as input and retains the subsequent outputs for the following variables: *CasIndex, CruiseMach, MachCasModified, DescentMach,* and *MachCas*. The variable *CasIndex* is the index value in the TCP list for the first CAS constrained waypoint. The variable *CruiseMach* is the last Mach crossing restriction value prior to the first CAS segment. The variable *MachCasModified* is a flag returned by this function if the *DescentMach* or *MachCas* values are changed. The variables *DescentMach* and *MachCas* are the planned descent Mach and planned Mach-to-CAS transition CAS, respectively, and these values may be modified by this function.

Initialize variables.

*i = 0*

*z = 0*

*fini = false*

*MachCasModified = false*

Perform up to two iterations to calculate any required Mach or CAS change in the descent.

*while ( (fini = false) and (i < 2))*

Calculate *z* at the descent Mach and the Mach-to-CAS CAS.

*z = MachCasTransitionAltitude( MachCas, DescentMach )*

Determine if *z* is below the CAS crossing restriction.

$if\ (z < Altitude_{CasIndex})$

Set the CAS to the value at this altitude, knowing the crossing restriction can't be met.

$MachCas = MachToCas(\ DescentMach,\ Altitude_{CasIndex}\ )$

$else\ if\ (z > Altitude\ Cross_{first\ waypoint})$

Set the Mach to the descent CAS at the cruise altitude.

$m = CasToMach(\ MachCas,\ Altitude_{first\ waypoint}\ )$

*if (m > CruiseMach) DescentMach = m*

$if\ (\ MachCas < Crossing\ CAS_{CasIndex})$

$MachCas = Crossing\ CAS_{CasIndex}$

*i = i + 1*

*else fini = true*

*end of while ( (fini = false) and (i < 2))*

Find the TOD TCP.

*fini = false*

*TodIndex = 0*

*i = index number of the first waypoint*

*while ((i < index number of the last waypoint) and (fini = false))*

*if ( ( $Altitude_i$ < $Altitude_{first\ waypoint}$) or (Crossing $CAS_i$ > 0) )*

*if (( $Altitude_i \neq Altitude_{first\ waypoint}$)) TodIndex = i - 1*

*else TodIndex = i*

*fini = true*

*i = i + 1*

*end of while ((i < index number of the last waypoint) and (fini = false))*

Calculate the entire decent distance.

$d = DTG_{TodIndex} - DTG_{CasIndex}$

Estimate the distance, *Daccel,* to the new Mach value.

*TodAccelerationDistance( TodIndex, CruiseMach, MachDescentMach, Valid, AccelIndex, Daccel )*

Estimate the distance, *Ddecel,* to the CAS crossing speed.

*BodDecelerationDistance( CasIndex, z, Mach Transition CAS, Ddecel )*

*fini = false*

*m = DescentMach*

The nominal speed values won't work, there is insufficient distance to obtain the acceleration and then slow to the crossing speed. Iterate until a solution is found.

*while ((fini = false) and (d < (Daccel + Ddecel)))*

Iterate the solution.

Slightly change the Mach and then find the CAS.

*m = m - 0.002*

*if (m < Cruise Mach)*

*m = Cruise Mach*

*fini = true*

Estimate the distance to the new Mach value.

*TodAccelerationDistance( TodIndex, Cruise Mach, m, Valid, AccelIndex, Daccel )*

Find the altitude where the acceleration ends.

$z = Crossing\ Altitude_{first\ waypoint} - (Daccel / d) * (Crossing\ Altitude_{first\ waypoint} - Crossing\ Altitude_{CasIndex})$

*CAS = MachToCas( m, z )*

Estimate the distance to the CAS crossing speed.

*BodDecelerationDistance( CasIndex, z, CAS, Ddecel )*

*if (d ≥ (Daccel + Ddecel))*

*fini = true*

*Modify the descent Mach and CAS values.*

*modified = true*

*DescentMach = m*

Add a buffer to the CAS so that subsequent Mach-to-CAS calculation won't cause an error.

*MachCas = CAS + 0.1*

*end of if (d ≥ (Daccel + Ddecel))*

## InterpolateWindAtDistance

The function *InterpolateWindAtDistance* is used to compute the wind speed and direction at an altitude, *Altitude*, for a specific distance, *Distance*, along the path. This function is a linear interpolation using the wind data from the input waypoints that bound the along-path distance.

Find the bounding input waypoints.

*i0 = index number of the first waypoint*

*j = index number of the first waypoint*

*fini = false*

*if (Distance < 0) Distance = 0*

*while ( (fini = false) and (j < (index number of the last waypoint - 1) ) )*

*if ( ($TCP_j$ = input waypoint) and ($DTG_j \geq$ Distance) ) i0 = j*

*if ($DTG_j$ < Distance ) fini = true*

*end of the while loop*

*i1 = i0 + 1*

*j = i1*

*fini = false*

*while ( (fini = false) and (j < index number of the last waypoint) )*

*if ( ($TCP_j$ = input waypoint) and ($DTG_j \leq$ Distance) )*

*i1 = j*

*fini = true*

*end of if*

*j = j + 1*

*end of the while loop*

*if (i1 > index number of the last waypoint) i1 = index number of the last waypoint*

*if (i0 = i1) InterpolateWindWptAltitude( $TCP_{i0}$, Altitude )*

*else*

Interpolate the winds at each waypoint.

*InterpolateWindWptAltitude( $TCP_{i0}$, Altitude ), returning Spd0 and Dir0*

*InterpolateWindWptAltitude( $TCP_{i1}$, Altitude ), returning Spd1 and Dir1*

Interpolate the winds between the two waypoints.

*r = ($DTG_{i0}$ - Distance) / ($DTG_{i0}$ - $DTG_{i1}$)*

*Wind Speed = ( (1 - r) * Spd0) + (r * Spd1)*

*a = DeltaAngle( Dir0, Dir1 )*

*Wind Direction = Dir0 + (r * a)*

## InterpolateWindWptAltitude

The function *InterpolateWindWptAltitude* is used to compute the wind speed and direction at an altitude, *Altitude*, for a specific TCP. This function is a linear interpolation using the wind data from the current TPC.

Find the index numbers, *p0* and *p1*, for the bounding altitudes.

$p0 = 0$

$p1 = 0$

$for\ (k = 1;\ k \leq Number\ of\ Wind\ Altitudes_i;\ k = k + 1)$

$\quad if\ (Wind\ Altitude_{i,\ k} \leq Altitude)\ p0 = k$

$\quad if\ (\ (Wind\ Altitude_{i,\ k} \geq Altitude) and\ (p1 = 0)\ )\ p1 = k$

$if\ (p1 = 0)\ p1 = Number\ of\ Wind\ Altitudes_i$

If $Altitude = Wind\ Altitude_{p0}$ or if $Altitude = Wind\ Altitude_{p1}$ then the wind data from that point is used. Otherwise, *Altitude* is not at an altitude on the wind profile of $TCP_i$, *i.e.,* $z = Wind\ Altitude_{i,\ k}$, *then:*

$if\ \ (Wind\ Altitude_{p1} \leq Wind\ Altitude_{p0})\ r = 0$

$else\ r = (Altitude - Wind\ Altitude_{p0}) / (Wind\ Altitude_{p1} - Wind\ Altitude_{p0})$

$Wind\ Speed = (\ (1 - r) * Wind\ Speed_{p0}) + (r * Wind\ Speed_{p1})$

$a = DeltaAngle(Wind\ Direction_{p0},\ Wind\ Direction_{p1})$

$Wind\ Direction = Wind\ Direction_{p0} + (r * a)$

## MachCasTransitionAltitude

The function *MachCasTransitionAltitude* is used to compute the altitude where the input Mach, *Mach*, and CAS, *Cas*, values would be equivalent

$$z = (1 - (((((0.2 * ((Cas/661.48)^2) + 1)^{3.5}) - 1) / (((0.2 * (Mach^2) + 1)^{3.5}) - 1))^{0.19026})) / 0.00000687535$$

return the value of *z.*

## RadialRadialIntercept

The function *RadialRadialIntercept* determines if two place-and-radial sets, each defined by a latitude, a longitude, and a track angle, will intersect and if so, calculates the latitude and longitude of the intercept point. Inputs are values of latitude, *Latitude*, longitude, *Longitude*, and angle*, Angle*; one set of each for the two place-and-radial sets. If a valid intercept can be calculated, then the intercept point's latitude and longitude are output, *NewLatitude* and *NewLongitude*, and the function returns a valid indication. Otherwise, the function returns an invalid indication.

Calculate the distance and the track angle between the two input positions.

$distance_{1,2} = arccosine( sine(Latitude_1) * sine(Latitude_2) + cosine(Latitude_1) * cosine(Latitude_2) * cosine(Longitude_1 - Longitude_2) )$

$track_{1,2} = arctangent2( sine(Longitude_2 - Longitude_1) * cosine(Latitude_2), cosine(Latitude_1) * sine(Latitude_2) - sine(Latitude_1) * cosine(Latitude_2) * cosine(Longitude_2 - Longitude_1))$

Check for error in the intercept calculation.

*error = false*

$track_1 = Angle_1 - track_{1,2} + 90$

*Adjust* $track_1$ *such that* $0 \geq track_1 \geq 360$

$track_2 = Angle_2 - track_{1,2} + 90$

*Adjust* $track_2$ *such that* $0 \geq track_2 \geq 360$

Determine the quadrant.

$ang_1 = track_2 + 180$

*Adjust* $ang_1$ *such that* $0 \geq ang_1 \geq 360$

*if ( ( |DeltaAngle(track1 , track2)| < 2) or ( |DeltaAngle(track1 , ang1)| < 2))*

Determine if the angles are really 180 degrees apart.

$ang_2 = Angle_2 + 180$

*Adjust* $ang_2$ *such that* $0 \geq ang_2 \geq 360$

$ang_3 = DeltaAngle(Angle_1, ang_2)$

$ang_4 = DeltaAngle(Angle_1, track_{1,2})$

*if ((|ang3| > 2) or (|ang4| > 2)) error = true*

*if (error = false)*

$RelativeLatLong(Latitude_1, Longitude_1, track_{1,2}, distance_{1,2} / 2, NewLatitude, NewLongitude)$

*else*

Determine the quadrant.

*if* ($track_1 \leq 90$) *quadrant1 = 1*

*else if* ($track_1 \leq 180$) *quadrant1 = 2*

*else if* ($track_1 \leq 270$) *quadrant1 = 3* *else quadrant1 = 4*

*if* $(track_2 \leq 90)$ *quadrant2 = 1*

*else if* $(track_2 \leq 180)$ *quadrant2 = 2*

*else if* $(track_2 \leq 270)$ *quadrant2 = 3*

*else quadrant2 = 4*

*if (quadrant1 = 1)*

    *if ((quadrant2 = 2) or (quadrant2 = 3)) error = true*

    *if ((quadrant2 = 1) and (chktk1 < chktk2)) error = true*

*else if (quadrant1 = 2)*

    *if ((quadrant2 = 1) or (quadrant2 = 4)) error = true*

    *if ((quadrant2 = 2) and (chktk1 > chktk2)) error = true*

*else if (quadrant1 = 3)*

    *if ((quadrant2 = 1) or (quadrant2 = 2) or (quadrant2 = 4)) error = true*

    *if* $(track_1 > track_2)$ *error = true*

*else*

    *if ((quadrant2 = 1) or (quadrant2 = 2) or (quadrant2 = 3)) error = true*

    *if* $(track_1 < track_2)$ *error = true*

*if (error = false)*

    $trx_1 = |Angle_1 - track_{1,2}|$

    *Adjust* $trx_1$ *such that* $0 \geq trx_1 \geq 360$

    $trx_2 = |Angle_2 - (track_{1,2} + 180)|$

    *Adjust* $trx_2$ *such that* $0 \geq trx_2 \geq 360$

    *if* $(trx_1 > 180)$ $trx_1 = 360 - trx_1$

    *if* $(trx_2 > 180)$ $trx_2 = 360 - trx_2$

    $ang_5 = 180 - trx_1 - trx_2$

    *if* $((ang_5 = 0)$ *or* $((ang_5\text{-}180) = 0)$ *or* $(distance_{1,2} = 0))$ *error = true* *if (error = false)*

*$distance_2$ = $distance_{1,2}$ * sine($trx_2$) / sine($ang_5$)*

*if ($distance_2 \leq 0$) $distance_2$ = - $distance_2$*

*if ($distance_2$ > max_intercept_range) error = true*

*else RelativeLatLong($Latitude_1$, $Longitude_1$, $Angle_1$, $distance_2$, NewLatitude, NewLongitude)*

*if (error) return false*

*else return true*

## RelativeLatLon

The function *RelativeLatLon* computes the latitude and longitude from input values of latitude, *BaseLat*, longitude, *BaseLon*, angle, *Angle*, and range, *Range*.

*if (Angle = 180) Latitude = -Range / 60 + BaseLat*

*else Latitude = ( (Range * cos(Angle) ) / 60) + BaseLat*

*if ( (BaseLat = 0) or (BaseLat = 180) ) Longitude = BaseLon*

*else if (Angle = 90) Longitude = BaseLon + Range / (60 * cos(BaseLat) )*

*else if (Angle = 270) Longitude = BaseLon - Range / (60 * cos(BaseLat) )*

*else*

*r1 = tangent(45 + 0.5 * Latitude)*

*r2 = tangent(45 + 0.5* BaseLat)*

*if ( (r1 = 0) or (r2 = 0) ) Longitude = 20, just some number, mark this as an error condition.*

*else Longitude = BaseLon + (180 / pi *(tangent(Angle)* (log(r1) - log(r2))))*

## TodAccelerationDistance

The *TodAccelerationDistance* function estimates the distance required for the special case of an acceleration from the top-of-descent Mach to the descent Mach at the top-of-descent. This function is invoked from *HandleDescentAccelDecel* and *DoTodAcceleration*, which passes in the index number for the TOD waypoint, *TodIndex*, and the Mach value at the TOD, *MachAtTod.* The function returns a validity flag to indicate if a TOD acceleration is valid, *Valid*, and if valid, the indices in the TCP list where the acceleration occurs, *AccelIndex*, and the distance from the index point of the acceleration, *Distance*.

Perform an initialization of flags and counters.

*fini = false*

*skip = true*

*k = TodIndex*

Make an initial guess of the distance to the new Mach value.

*Descent Speed = Mach Descent Mach*

$Mach\ Rate_1 = CasToMach(\ 0.75\ kt\ /\ sec,\ Altitude_{TodIndex})$

Compute the time required to do the deceleration.

$t = (Mach\ Descent\ Mach - MachAtTod\ )\ /\ Mach\ Rate_1$

Compute the wind speed and direction at the current altitude.

$InterpolateWindWptAltitude(\ Wind\ Profile_{TodIndex},\ Altitude_{TodIndex},\ Wind\ Speed,\ Wind\ Direction\ )$

Get the ground track at the current point.

$if\ (\ WptInTurn(\ Waypoint_{TodIndex})\ )\ track = Ground\ Track_{TodIndex\ +\ 1}$

$else\ track = Ground\ Track_{TodIndex}$

$TOD\ Ground\ Speed = ComputeGndSpeedUsingMachAndTrack(\ MachAtTod,\ track,\ Altitude_{TodIndex},$
$Wind\ Speed,\ Wind\ Direction\ )$

$Descent\ Ground\ Speed = ComputeGndSpeedUsingMachAndTrack(\ Mach\ Descent\ Mach,\ track,$
$Altitude_{TodIndex},\ Wind\ Speed,\ Wind\ Direction\ )$

The average ground speed is as follows:

*Average Ground Speed = ( TOD Ground Speed + Descent Ground Speed ) / 2*

The distance estimate, *dx*, is *Average Ground Speed * t* with a conversion to nm.

*dx = Average Ground Speed * t / 3600*

Now compute better estimates, doing this twice to refine the estimation.

$for\ (\ i = 1;\ i \leq 2;\ i = i + 1\ )$

*skip = false*

Determine if this distance is beyond the next downstream waypoint.

*k = TodIndex*

$d = DTG_{TodIndex} - dx$

$while\ (\ (\ k < (\ index\ number\ of\ the\ last\ waypoint - 1\ )\ )\ and\ (\ DTG_{k+1} > d\ )\ )$

*if ( ( $k \neq$ TodIndex ) and ( Crossing Rate$_k > 0$ ) ) skip = True;*

*$k = k + 1$*

Compute the wind speed and direction at the new altitude.

*InterpolateWindWptAltitude( Waypoint$_k$, Altitude$_k$, Wind Speed, Wind Direction )*

The ground speed at the this point is:

*Descent Ground Speed = ComputeGndSpeedUsingMachAndTrack( Mach Descent Mach, Ground Track$_k$, Altitude$_k$, Wind Speed, Wind Direction )*

The average ground speed is:

*Average Ground Speed = ( TOD Ground Speed + Descent Ground Speed ) / 2*

The distance, *dx*, is:

*dx = Average Ground Speed * t / 3600*

If there is a valid deceleration point, add it.

*Valid = not skip*

*AccelIndex = k*

*Distance = dx*

## WptInTurn

The *WptInTurn* function simply determines if the waypoint is between a turn-entry TCP and a turn-exit TCP. If this is true, then the function returns a value of true, otherwise it returns a value of false.

*fini = false*

*within = false*

*$j = i + 1$*

*while ((fini = false) and (j < (index number of the last waypoint)))*

*if (TurnType$_j$ = turn-entry) fini = true*

*else if (TurnType$_j$ = turn-exit)*

*fini = true*

*within = true*

*$j = j + 1$*

*return within*

## Summary

The algorithm described in this document takes as input a list of waypoints, their trajectory-specific data, and associated wind profile data. A full 4D trajectory can then be generated by the techniques described. A software prototype has been developed from this documentation. An example of the data input and the prototype-generated output is provided in the Appendix.

## References


1. Abbott, T. S.; and Moen, G. C,: *Effect of Display Size on Utilization of Traffic Situation Display for Self-Spacing Task*, NASA TP-1885, 1981.

2. Abbott, Terence S.: *A Compensatory Algorithm for the Slow-Down Effect on Constant-Time-Separation Approaches*, NASA TM-4285, 1991.

3. Sorensen, J. A.; Hollister, W.; Burgess, M.; and Davis, D.: *Traffic Alert and Collision Avoidance System (TCAS) - Cockpit Display of Traffic Information (CDTI) Investigation*, DOT/FAA/RD-91/8, 1991.

4. Williams, D. H.: *Time-Based Self-Spacing Techniques Using Cockpit Display of Traffic Information During Approach to Landing in a Terminal Area Vectoring Environment*, NASA TM-84601, 1983.

5. Koenke, E.; and Abramson, P.: *DAG-TM Concept Element 11, Terminal Arrival: Self Spacing for Merging and In-trail Separation, Advanced Air Transportation Technologies Project*, 2004.

6. Abbott, T. S.: *Speed Control Law for Precision Terminal Area In-Trail Self Spacing*, NASA TM 2002-211742, 2002.

7. Osaguera-Lohr, R. M.; Lohr, G. W.; Abbott, T. S.; and Eischeid, T. M.: *Evaluation Of Operational Procedures For Using A Time-Based Airborne Interarrival Spacing Tool*, AIAA-2002-5824, 2002.

8. Lohr, G. W.; Osaguera-Lohr, R. M.; and Abbott, T. S.: *Flight Evaluation of a Time-based Airborne Inter-arrival Spacing Tool*, Paper 56, Proceedings of the 5th USA/Europe ATM Seminar at Budapest, Hungary, 2003.

9. Krishnamurthy, K.; Barmore, B.; Bussink, F. J.; Weitz, L.; and Dahlene, L.: *Fast-Time Evaluations Of Airborne Merging and Spacing In Terminal Arrival Operations*, AIAA-2005-6143, 2005.

10. Barmore, B.; Abbott, T. S.; and Capron, W. R.: *Evaluation of Airborne Precision spacing in a Human-in-the-Loop Experiment*, AIAA-2005-7402, 2005.

11. Hoffman, E.; Ivanescu, D.; Shaw, C.; and Zeghal, K.: *Analysis of Constant Time Delay Airborne Spacing Between Aircraft of Mixed Types in Varying Wind Conditions*, Paper 77, Proceedings of the 5th USA/Europe ATM Seminar at Budapest, Hungary, 2003.

12. Ivanescu, D.; Powell, D.; Shaw, C.; Hoffman, E.; and Zeghal, K.: *Effect Of Aircraft Self-Merging In Sequence On An Airborne Collision Avoidance System*, AIAA 2004-4994, 2004.

13. Weitz, L.; Hurtado, J. E.; and Bussink, F. J. L.: *Increasing Runway Capacity for Continuous Descent Approaches Through Airborne Precision Spacing*, AIAA 2005-6142, 2005.

14. Barmore, B. E.; Abbott, T. S.; and Krishnamurthy, K.: *Airborne-Managed Spacing in Multiple Arrival Streams*, Proceedings of the 24th Congress of the International Council of Aeronautical Sciences,, 2004.

15. Baxley, B.; Barmore, B.; Bone, R.; and Abbott, T. S.: *Operational Concept for Flight Crews to Participate in Merging and Spacing of Aircraft*, 2006 AIAA Aviation Technology, Integration and Operations Conference, 2006.

16. Lohr, G. W.; Oseguera-Lohr, R. M.; Abbott, T. S.; Capron, W. R.; and Howell, C. T.: *Airborne Evaluation and Demonstration of a Time-Based Airborne Inter-Arrival Spacing Tool*, NASA/TM-2005-213772, 2005.

17. Oseguera-Lohr, R. M.; and Nadler, E. D.: *Effects of an Approach Spacing Flight Deck Tool on Pilot Eyescan*, NASA/TM-2004-212987, 2004.

18. Lohr, G. W.; Oseguera-Lohr, R. M.; Abbott, T. S.; and Capron, W. R.: *A Time-Based Airborne Inter-Arrival Spacing Tool: Flight Evaluation Result*, ATC Quarterly, Vol 13 no 2, 2005.

19. Barmore, B.; Krishnamurthy, K.; Capron, W.; Baxley, B.; and Abbott, T. S.: *An Experimental Validation of Merging and Spacing by Flight Crew*, 2006 AIAA Aviation Technology, Integration and Operations Conference, 2006.

20. Krishnamurthy, K.; Barmore, B.; and Bussink, F. J. L.: *Airborne Precision Spacing in Merging Terminal Arrival Routes: A Fast-time Simulation Study*, Proceedings of the 6th USA/Europe ATM Seminar, 2005.

*21.* Abbott, T. S.: *A Trajectory Algorithm to Support En Route and Terminal Area Self-Spacing Concepts*, NASA CR-2007-214899, 2007.

22. Olson, Wayne M.: *Aircraft Performance Flight Testing*, AFFTC-TIH-99-01, 2000.

# Appendix Example Data Sets

## Input Trajectory Data

An example input trajectory data set is provided in Table A1.

The descent Mach is 0.82. The Mach-to-CAS transition speed for this example is 310 knots. Note that Waypoint-18 is the runway threshold at a 50 ft crossing height. No RF turns are shown.

Table A1. Example of trajectory input data.

| Identifier | Latitude | Longitude | Crossing Altitude | Crossing Angle | Crossing CAS | Crossing Mach | Crossing Rate |
|---|---|---|---|---|---|---|---|
| Waypoint-01 | 31.87476 | -103.244 | 37000 | 0 | 0 | 0.78 | 0 |
| Waypoint-02 | 32.48133 | -99.8635 | 0 | 0 | 0 | 0 | 0 |
| Waypoint-03 | 32.20548 | -98.9531 | 0 | 0 | 0 | 0 | 0 |
| Waypoint-04 | 32.19398 | -98.6621 | 0 | 0 | 0 | 0 | 0 |
| Waypoint-05 | 32.17042 | -98.113 | 0 | 0 | 0 | 0 | 0 |
| Waypoint-06 | 32.15959 | -97.8777 | 0 | 0 | 0 | 0 | 0 |
| Waypoint-07 | 32.34026 | -97.6623 | 0 | 0 | 0 | 0 | 0 |
| Waypoint-08 | 32.46908 | -97.5079 | 0 | 0 | 0 | 0 | 0 |
| Waypoint-09 | 32.64444 | -97.2967 | 11700 | 3.0 | 0 | 0 | 0 |
| Waypoint-10 | 32.71448 | -97.2119 | 11000 | 1.1 | 0 | 0 | 0 |
| Waypoint-11 | 32.74948 | -97.1695 | 0 | 0 | 0 | 0 | 0 |
| Waypoint-12 | 32.97496 | -97.1783 | 0 | 0 | 0 | 0 | 0 |
| Waypoint-13 | 33.10724 | -97.1754 | 5300 | 2.3 | 220 | 0 | 0.5 |
| Waypoint-14 | 33.10658 | -97.0537 | 4300 | 1.8 | 0 | 0 | 0 |
| Waypoint-15 | 33.03645 | -97.0541 | 0 | 0 | 0 | 0 | 0 |
| Waypoint-16 | 33.00561 | -97.0542 | 2400 | 3.1 | 170 | 0 | 0.5 |
| Waypoint-17 | 32.95953 | -97.0544 | 1495 | 3.0 | 127 | 0 | 0.75 |
| Waypoint-18 | 32.91582 | -97.0546 | 660 | 3.0 | 127 | 0 | 0.75 |

## Input Wind Data

An example wind speed data set is provided in Table A2.

Table A2. Example of wind speed input data.

| Identifier | Altitude | Wind Speed | Wind Direction |
|---|---|---|---|
| Waypoint-01 | 0 | 20 | 180 |
| | 10000 | 50 | 270 |
| | 20000 | 60 | 340 |
| | 40000 | 70 | 350 |
| Waypoint-02 | 0 | 20 | 180 |
| | 10000 | 50 | 270 |
| | 20000 | 60 | 340 |
| | 40000 | 70 | 350 |
| Waypoint-03 | 0 | 20 | 180 |
| | 10000 | 50 | 270 |
| | 20000 | 60 | 340 |
| | 40000 | 70 | 350 |
| Waypoint-04 | 0 | 20 | 180 |
| | 10000 | 50 | 270 |
| | 20000 | 60 | 340 |
| | 40000 | 70 | 350 |
| Waypoint-05 | 0 | 20 | 180 |
| | 10000 | 50 | 270 |
| | 20000 | 60 | 340 |
| | 40000 | 70 | 350 |
| Waypoint-06 | 0 | 20 | 180 |
| | 10000 | 50 | 270 |
| | 20000 | 60 | 340 |
| | 40000 | 70 | 350 |
| Waypoint-07 | 0 | 20 | 160 |
| | 10000 | 50 | 240 |
| | 20000 | 60 | 320 |
| | 40000 | 70 | 330 |

Table A2 (continued). Example of wind speed input data.

| Identifier | Altitude | Wind Speed | Wind Direction |
|---|---|---|---|
| Waypoint-08 | 0 | 20 | 160 |
| | 10000 | 50 | 240 |
| | 20000 | 60 | 330 |
| | 40000 | 70 | 340 |
| Waypoint-09 | 0 | 20 | 160 |
| | 10000 | 50 | 240 |
| | 20000 | 60 | 330 |
| | 40000 | 70 | 340 |
| Waypoint-10 | 0 | 20 | 160 |
| | 10000 | 50 | 240 |
| | 20000 | 50 | 330 |
| | 40000 | 60 | 340 |
| Waypoint-11 | 0 | 20 | 160 |
| | 10000 | 50 | 240 |
| | 20000 | 50 | 330 |
| | 40000 | 60 | 340 |
| Waypoint-12 | 0 | 20 | 160 |
| | 10000 | 50 | 240 |
| | 20000 | 50 | 330 |
| | 40000 | 60 | 340 |
| Waypoint-13 | 0 | 20 | 160 |
| | 10000 | 50 | 240 |
| | 20000 | 50 | 330 |
| | 40000 | 60 | 340 |
| Waypoint-14 | 0 | 20 | 160 |
| | 10000 | 40 | 240 |
| | 20000 | 50 | 330 |
| | 40000 | 60 | 340 |

Table A2 (continued). Example of wind speed input data.

| Identifier | Altitude | Wind Speed | Wind Direction |
|---|---|---|---|
| Waypoint-15 | 0 | 20 | 160 |
| | 10000 | 40 | 240 |
| | 20000 | 50 | 330 |
| | 40000 | 60 | 340 |
| Waypoint-16 | 0 | 20 | 160 |
| | 10000 | 40 | 240 |
| | 20000 | 50 | 330 |
| | 40000 | 60 | 340 |
| Waypoint-17 | 0 | 20 | 160 |
| | 10000 | 40 | 240 |
| | 20000 | 50 | 330 |
| | 40000 | 60 | 340 |
| Waypoint-18 | 0 | 20 | 160 |
| | 10000 | 40 | 240 |
| | 20000 | 50 | 330 |
| | 40000 | 60 | 340 |

## Output Trajectory Data

An example of the data available from this trajectory algorithm is provided in Table A3. Not shown, but also available, are the latitude and longitude data for each TCP.

Table A3. Example of the trajectory output data.

| TCP type | Identifier | Altitude | Mach | CAS | Mach Segment | Ground Speed | Track | DTG | TTG |
|---|---|---|---|---|---|---|---|---|---|
| Input | Waypoint-01 | 37000 | 0.78 | 252.5 | true | 450.7 | 77.1 | 366.06 | 3214.8 |
| Turn-entry | | 37000 | 0.78 | 252.5 | true | 450.7 | 77.1 | 192.89 | 1831.4 |
| Input | Waypoint-02 | 37000 | 0.78 | 252.5 | true | 469.9 | 93.3 | 190.64 | 1813.8 |
| Turn-exit | | 37000 | 0.78 | 252.5 | true | 487.5 | 109.5 | 188.39 | 1796.9 |
| Turn-entry | | 37000 | 0.78 | 252.5 | true | 487.5 | 109.5 | 142.90 | 1461.0 |
| Input | Waypoint-03 | 37000 | 0.78 | 252.5 | true | 478.6 | 101 | 141.68 | 1451.9 |
| Turn-exit | | 37000 | 0.78 | 252.5 | true | 469.1 | 92.6 | 140.46 | 1442.6 |

Table A3 (continued). Example of the trajectory output data.

| TCP type | Identifier | Altitude | Mach | CAS | Mach Segment | Ground Speed | Track | DTG | TTG |
|---|---|---|---|---|---|---|---|---|---|
| Input | Waypoint-04 | 37000 | 0.78 | 252.5 | true | 469.1 | 92.8 | 126.90 | 1338.6 |
| VTCP | | 37000 | 0.78 | 252.5 | true | 469.3 | 93 | 125.46 | 1327.5 |
| VTCP | | 36306 | 0.82 | 271.2 | true | 494.5 | 93 | 123.28 | 1311.2 |
| VTCP | | 30337 | 0.82 | 310 | false | 509.6 | 93 | 104.53 | 1176.8 |
| Input | Waypoint-05 | 28569 | 0.793 | 310 | false | 497.2 | 93 | 98.98 | 1137.1 |
| Turn-entry | | 25777 | 0.751 | 310 | false | 478.5 | 93 | 90.21 | 1072.4 |
| Input | Waypoint-06 | 24818 | 0.737 | 310 | false | 446.6 | 69.1 | 87.20 | 1048.9 |
| Turn-exit | | 23858 | 0.723 | 310 | false | 415.4 | 45.2 | 84.19 | 1023.8 |
| Input | Waypoint-07 | 19976 | 0.672 | 310 | false | 393.4 | 45.3 | 72.00 | 915.2 |
| Input | Waypoint-08 | 16474 | 0.629 | 310 | false | 404.6 | 45.4 | 61.00 | 816.0 |
| Input | Waypoint-09 | 11700 | 0.576 | 310 | false | 409.4 | 45.5 | 46.01 | 683.4 |
| VTCP | | 11432 | 0.574 | 310 | false | 408.5 | 45.5 | 43.71 | 663.1 |
| Input | Waypoint-10 | 11000 | 0.524 | 284.6 | false | 378.1 | 45.5 | 40.01 | 629.3 |
| VTCP | | 11000 | 0.519 | 282 | false | 375.1 | 45.5 | 39.65 | 625.8 |
| Turn-entry | | 10811 | 0.507 | 276.4 | false | 368.4 | 45.5 | 38.87 | 618.3 |
| Input | Waypoint-11 | 10382 | 0.479 | 262.9 | false | 340.6 | 21.8 | 37.12 | 600.5 |
| VTCP | | 10000 | 0.453 | 250 | false | 324.7 | 19.3 | 35.55 | 583.5 |
| Turn-exit | | 9954 | 0.452 | 250 | false | 308.9 | 358.1 | 35.36 | 581.4 |
| Input | Waypoint-12 | 7105 | 0.429 | 250 | false | 307.7 | 1.1 | 23.69 | 445.1 |
| VTCP | | 6474 | 0.424 | 250 | false | 307.3 | 1.1 | 21.10 | 414.8 |
| Turn-entry | | 5793 | 0.391 | 233.1 | false | 286.5 | 1.1 | 18.31 | 381.0 |
| Input | Waypoint-13 | 5300 | 0.366 | 220 | false | 270 | 45.7 | 16.29 | 354.9 |
| Turn-exit | | 4909 | 0.363 | 220 | false | 245 | 90.3 | 14.27 | 326.6 |
| Turn-entry | | 4556 | 0.361 | 220 | false | 242 | 90.3 | 12.42 | 299.3 |
| Input | Waypoint-14 | 4300 | 0.359 | 220 | false | 215.4 | 135.3 | 11.08 | 278.2 |
| VTCP | | 3987 | 0.357 | 220 | false | 204.1 | 164.4 | 10.21 | 263.2 |
| Turn-exit | | 3831 | 0.35 | 215.9 | false | 197 | 180.3 | 9.74 | 254.7 |
| Input | Waypoint-15 | 3009 | 0.305 | 191.2 | false | 170.7 | 180.2 | 7.24 | 205.8 |
| Input | Waypoint-16 | 2400 | 0.268 | 170 | false | 148.8 | 180.2 | 5.39 | 164.1 |
| VTCP | | 2140 | 0.267 | 170 | false | 148.9 | 180.2 | 4.65 | 146.2 |

Table A3 (continued). Example of the trajectory output data.

| TCP type | Identifier | Altitude | Mach | CAS | Mach Segment | Ground Speed | Track | DTG | TTG |
|---|---|---|---|---|---|---|---|---|---|
| Input | Waypoint-17 | 1495 | 0.197 | 127 | false | 105.5 | 180.2 | 2.62 | 88.9 |
| Input | Waypoint-18 | 660 | 0.194 | 127 | false | 106.9 | 180.2 | 0.00 | 0.0 |

| **REPORT DOCUMENTATION PAGE** | *Form Approved OMB No. 0704-0188* |
|---|---|



**1. REPORT DATE** *(DD-MM-YYYY)*
01- 07 - 2014

**2. REPORT TYPE**
Contractor Report

**3. DATES COVERED** *(From - To)*

**4. TITLE AND SUBTITLE**
A Trajectory Algorithm to Support En Route and Terminal Area Self-Spacing Concepts: Third Revision

**5a. CONTRACT NUMBER**
NNL10AA14B

**5b. GRANT NUMBER**

**5c. PROGRAM ELEMENT NUMBER**

**6. AUTHOR(S)**
Abbott, Terence S.

**5d. PROJECT NUMBER**

**5e. TASK NUMBER**

**5f. WORK UNIT NUMBER**
305295.02.31.07.01.03

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**
NASA Langley Research Center
Hampton, Virginia 23681

**8. PERFORMING ORGANIZATION REPORT NUMBER**

**9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)**
National Aeronautics and Space Administration
Washington, DC 20546-0001

**10. SPONSOR/MONITOR'S ACRONYM(S)**
NASA

**11. SPONSOR/MONITOR'S REPORT NUMBER(S)**
NASA/CR-2014-218288

**12. DISTRIBUTION/AVAILABILITY STATEMENT**
Unclassified - Unlimited
Subject Category 03
Availability: NASA CASI (443) 757-5802

**13. SUPPLEMENTARY NOTES**
This document is a revision to NASA-CR-2010-216204,
dated February 2010.
Langley Technical Monitor: Bryan E. Barmore

**14. ABSTRACT**
This document describes an algorithm for the generation of a four dimensional trajectory. Input data for this algorithm are similar to an augmented Standard Terminal Arrival (STAR) with the augmentation in the form of altitude or speed crossing restrictions at waypoints on the route. This version of the algorithm accommodates constant radius turns and cruise altitude waypoints with calibrated airspeed, versus Mach, constraints. The algorithm calculates the altitude, speed, along path distance, and along path time for each waypoint. Wind data at each of these waypoints are also used for the calculation of ground speed and turn radius.

**15. SUBJECT TERMS**
Algorithm; Calibrating; Crossing; Ground speed; Trajectories; Wind measurements

| 16. SECURITY CLASSIFICATION OF: a. REPORT | b. ABSTRACT | c. THIS PAGE | 17. LIMITATION OF ABSTRACT | 18. NUMBER OF PAGES | 19a. NAME OF RESPONSIBLE PERSON |
|---|---|---|---|---|---|
| | | | | | STI Help Desk (email: help@sti.nasa.gov) |
| | | | | | **19b. TELEPHONE NUMBER** *(Include area code)* |
| U | U | U | UU | 93 | (443) 757-5802 |