NASA/CR-2007-214899

A Trajectory Algorithm to Support En Route and Terminal Area Self-Spacing Concepts

Terence S. Abbott
Booz Allen Hamilton, McLean, Virginia

September 2007

The NASA STI Program Office . . . 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 Office plays a key part in helping NASA maintain this important role.

The NASA STI Program Office is operated by Langley Research Center, the lead center for NASA's scientific and technical information. The NASA STI Program Office provides access to the NASA STI Database, the largest collection of aeronautical and space science STI in the world. The Program Office is also NASA's institutional mechanism for disseminating the results of its research and development activities. These results are published 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 that complement the STI Program Office's diverse offerings include creating custom thesauri, building customized databases, organizing and publishing research results ... even providing videos.

For more information about the NASA STI Program Office, see the following:

- Access the NASA STI Program Home Page at ***http://www.sti.nasa.gov***

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

- Fax your question to the NASA STI Help Desk at (301) 621-0134

- Phone the NASA STI Help Desk at (301) 621-0390

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

NASA/CR-2007-214899

A Trajectory Algorithm to Support En Route and Terminal Area Self-Spacing Concepts

Terence S. Abbott
Booz Allen Hamilton, McLean, Virginia

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

Prepared for Langley Research Center
under Purchase Order L-70750D

September 2007

Available from:

NASA Center for AeroSpace Information (CASI)
7115 Standard Drive
Hanover, MD 21076-1320
(301) 621-0390

National Technical Information Service (NTIS)
5285 Port Royal Road
Springfield, VA 22161-2171
(703) 605-6000

Table of Contents

Nomenclature

2D: 2 dimensional

4D: 4 dimensional

ADS-B: Automatic Dependence Surveillance Broadcast

CAS: Calibrated Airspeed

DTG: Distance-To-Go

MSL: Mean Sea Level

STAR: Standard Terminal Arrival Route

TAS: True Airspeed

TCP: Trajectory Change Point

TTG: Time-To-Go

VTCP: Vertical Trajectory Change Point

Wpt: Waypoint

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 Route (STAR) with the augmentation in the form of altitude or speed crossing restrictions at waypoints on the route. Wind data at each waypoint are also inputs into this algorithm. The algorithm calculates the altitude, speed, along path distance, and along path time for each waypoint.

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 have 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 over-the-threshold runway 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. 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 to 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 groundspeeds along the path.

The major complexity in computing a 4D trajectory involves the interrelationship of groundspeed 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 then 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. 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 / CAS transition speed (CAS) value may also be input for profiles that involve a constant Mach / CAS descent segment.

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 provided better groundspeed (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.

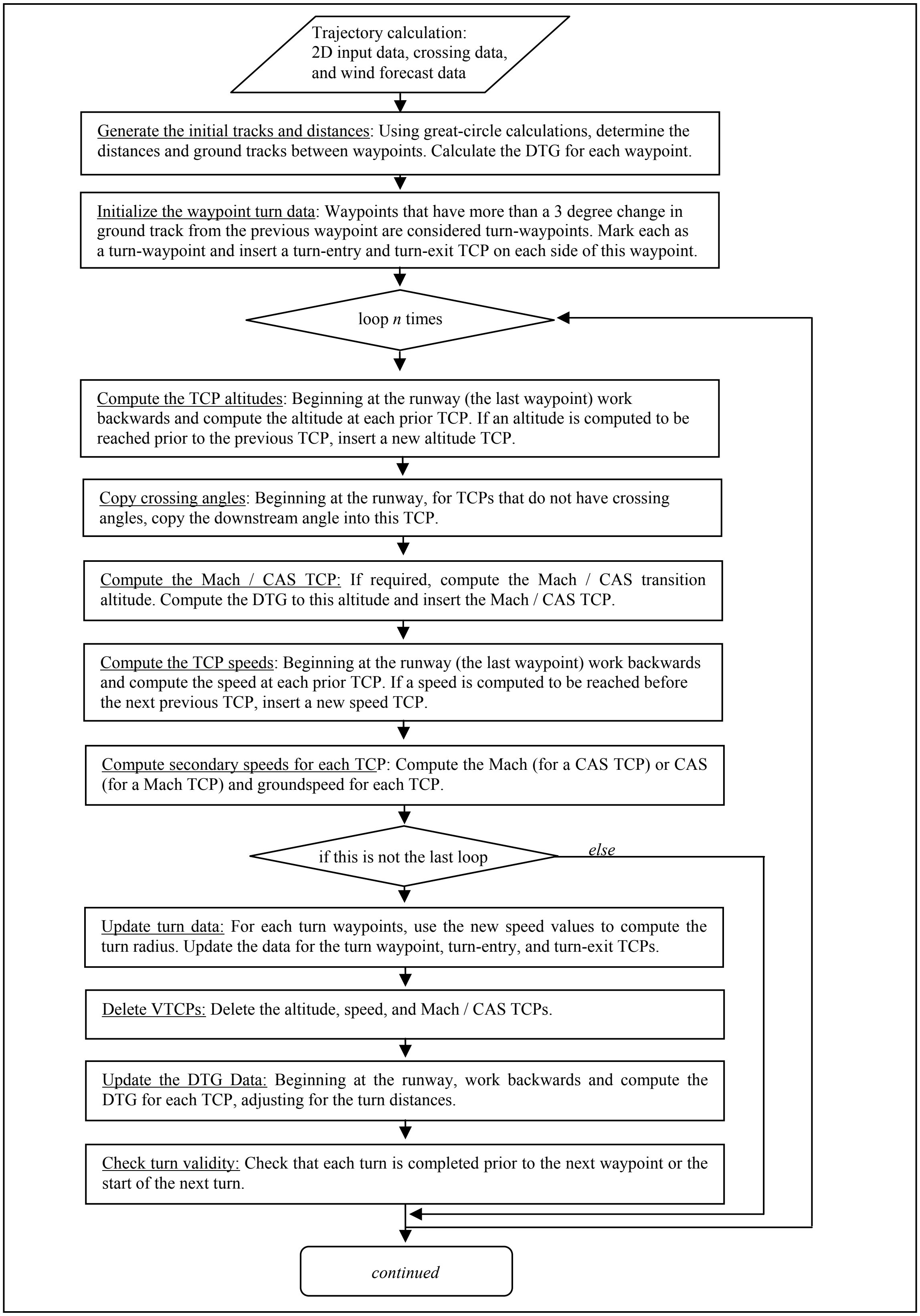

Figure 1. Basic functions.

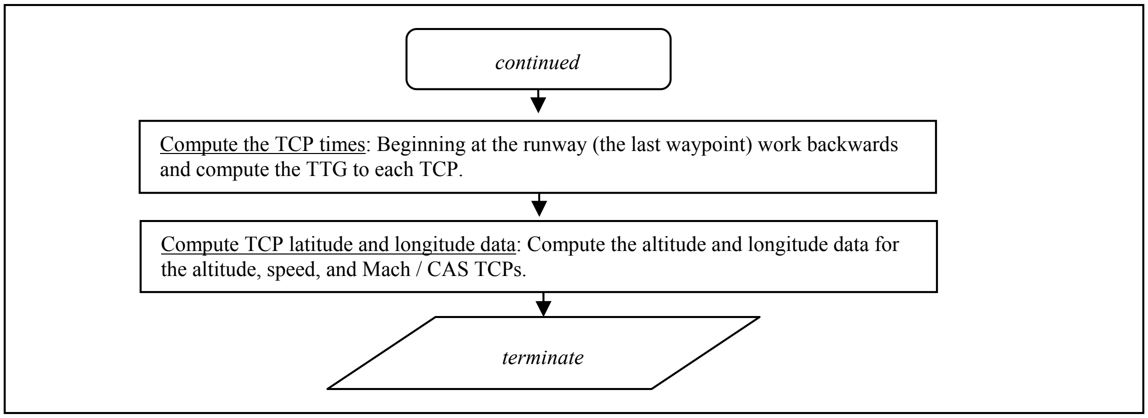

Figure 1 (continued). Basic functions.

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.

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; turn-entry, identifying a TCP that marks the start of a turn; turn-exit, identifying a TCP that marks the end of a turn; vertical TCPs (VTCPs), denoting a change in the altitude or speed profile; and a Mach / CAS TCP, denoting the Mach / CAS transition point. TCPs are also denoted relative to the associated speed value, whether the crossing speed is Mach or CAS derived.

Description of Major Functions

The functions shown in figure 1 are described in detail in this section. The functions are presented in the order shown in the figure. 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 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.

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

This is an initialization function that determines 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. This 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 135 degrees.

if (absolute(a) > 135)

Set an error and ignore the turn.

a = 0

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

if (absolute(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*

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-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-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 (absolute(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 equal to the inbound ground track; and inserts a co-distance turn-exit TCP after this turn-waypoint with the ground track angle for this turn-exit TCP set equal to the outbound ground track. An example illustrating the inserted turn-start and turn-end TCPs is shown in figure 2.

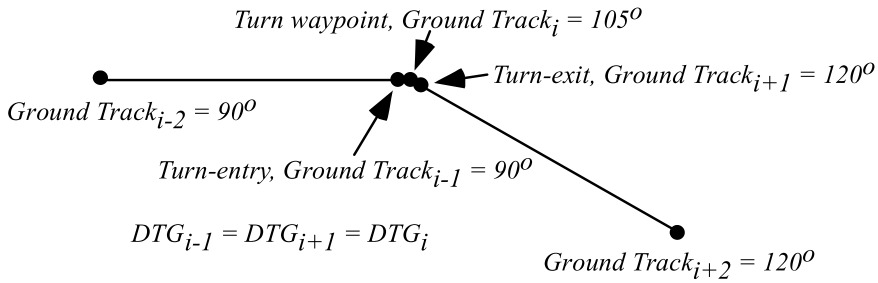

Figure 2. Initialized turn waypoint.

Compute TCP Altitudes

Beginning with the last waypoint, this 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. 3), searches backward for the previous constraint (TCP_{i-3} in fig. 3), 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 4. This function is performed in the following steps:

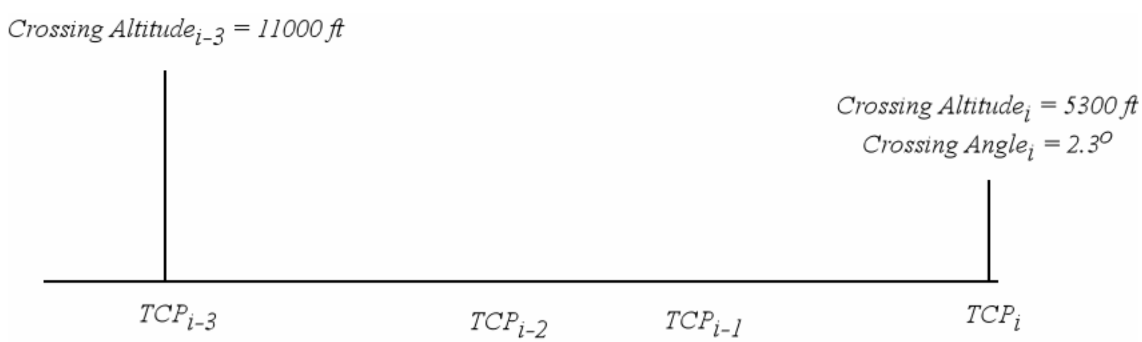

Figure 3. Input altitude crossing constraints.

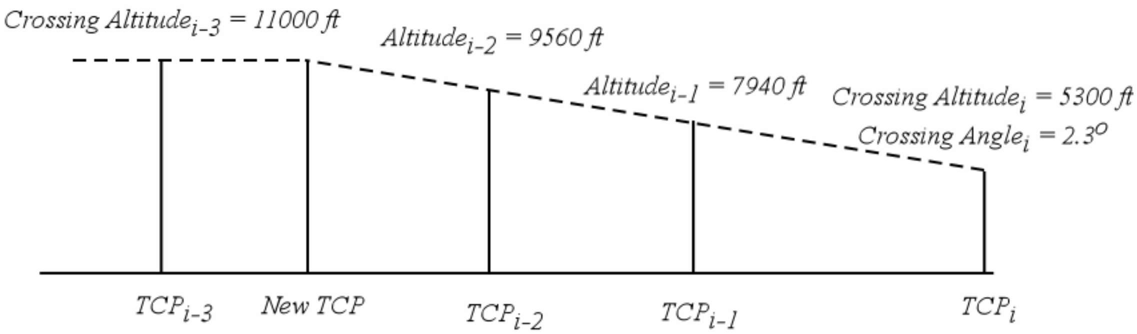

Figure 4. Computed altitude profile with TCP added.

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,

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

While (cc > index number of the first waypoint)

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, set 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.

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

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

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 $(Prior\ Altitude \leq Test\ Altitude)$

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* $(\ absolute(z - Prior\ Altitude) < some\ small\ value)\ )$

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

else $Altitude_{k-1} = z$

Check to see if the constraint has been reached, if not, set an error condition.

if ((k-1) = pc)

if (absolute($Altitude_{pc}$ - Crossing $Altitude_{pc}$) > 100ft) set an error here

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 (absolute(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.

$DTG_k = d$

$Altitude_k$ = Prior Altitude

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

if (WptInTurn(k)) Ground $Track_k$ *= ComputedGndTrk(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

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

Compute Mach / CAS TCP

If required, compute the Mach / CAS altitude and insert a TCP at this point. 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 / CAS TCP is then inserted into the TCP list at this point.

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 / CAS transition speed input, use this value for the *First CAS* value.

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

Compute the Mach / CAS transition altitude.

$$z = (1.0 - (((((0.2 * ((FirstCas/661.48)^{2.0}) + 1.0)^{3.5}) - 1.0) / (((0.2 * (LastMach^{2.0}) + 1.0)^{3.5}) - 1.0))^{0.19026})) / 0.00000687535$$

For an actual implementation, it would be beneficial to check for an error at this point. If *z* greater than the altitude associated with the *Last Mach* TCP or if *z* 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 / CAS transition TCP.

Add the data for this new TCP.

$Crossing\ Mach_i = Last\ Mach$

$Crossing\ CAS_i = First\ CAS$

$CAS\ Rate_i = CAS\ Rate$

$DTG_i = d$

$Altitude_i = z$

$Ground\ Track_i = Saved\ Ground\ Track$

$Mach_i = Last\ Mach$

$CAS_i = First\ CAS$

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 TCP Speeds

This 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 / CAS transition point.

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

if (Doing Mach) ComputeTcpMach(cc)

else ComputeTcpCas(cc)

end of while cc > index number of the first waypoint

Compute Secondary Speeds

This function adds the Mach values to CAS TCPs, the CAS values to Mach TCPs, and the groundspeed values to all TCPs. This function is preformed in the following steps:

Doing Mach = false

Working backwards form 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 / CAS transition point.

if (TCP_i *= Mach CAS Transition) 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 groundspeed. Compute 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

This 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} *≠ 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 *≠ 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 groundspeeds 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,

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

else

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

$AvgGsFirstHalf = 0$

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

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

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

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 groundspeed.

AvgGsLastHalf = 0

for (j = index; j ≤ (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 <= (end - 1); j = j + 1)

end of else if (d ≤ 0)

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)*

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

half turn = full turn / 2

Compute the outputs from the average groundspeed.

Average Ground Speed = (AvgGsFirstHalf + AvgGsLastHalf) / 2

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

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

w = c0 / Average Ground Speed

The time to make the turn is,

$Turn\ Data\ Turn\ Time_{index} = absolute(full\ turn) / w$

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,

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

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 5.

Turn Data Straight Distance1$_{index}$ = *Turn Data Turn Radius* $_{index}$ * *tangent(absolute(half turn))*

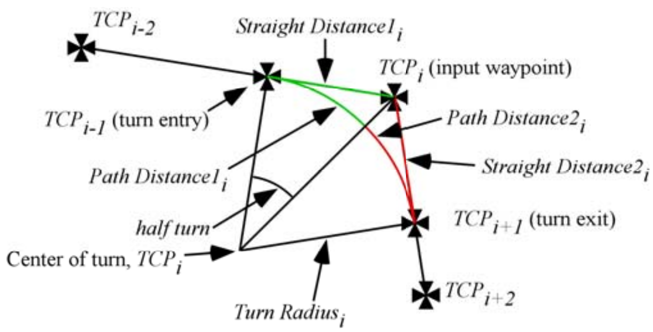

Figure 5. Turn distances for waypoint$_i$.

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 5.

Turn Data Path Distance1$_{index}$ = *absolute(half turn) * Turn Data Turn Radius*$_{index}$ */ 57.3*

w = c0 / AvgGsFirstHalf

Turn Data Turn Time1$_{index}$ = *absolute(half turn) / w*

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 calculates are recomputed here for clarity.

Turn Data Straight Distance2$_{index}$ = *Turn Data Turn Radius* $_{index}$ * *tangent(absolute(half turn))*

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

$w = c0 / AvgGsLastHalf$

$Turn\ Data\ Turn\ Time2_{index} = absolute(half\ turn) / w$

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.

GenerateWptWindProfile(DTG_{start}, TCP_{start})

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

This function simply deletes the altitude, speed, and Mach / CAS TCPs. The remaining TCPs will only consist of input waypoints, turn-entry, and turn-exit TCPS.

Update DTG Data

This 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.

$DTG_{first\ waypoint} = 0$

i = index number of the last waypoint

while (i > 0)

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

if (WptInTurn(i))

$DTG_{i-1} = DTG_i + Turn\ Data\ Path\ Distance1_i$

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.

nn = i - 1

while (TCP$_{nn}$ $\neq$ input waypoint) nn = nn - 1

if (WptInTurn(nn))

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

DistanceOffset = Turn Data Straight Distance2$_{nn}$ - TurnData.PathDistance2$_{nn}$

The DTG to the input waypoint is then:

DTG$_{nn}$ = (Center to Center Distance$_i$ - PriorDistanceOffset - DistanceOffset) + DTG$_i$

The turn-exit DTG is then,

DTG$_{nn+1}$ = DTG$_{nn}$ - Turn Data Path Distance2$_{nn}$

else

The next waypoint is not in a turn.

DTG$_{nn}$ = Center to Center Distance$_i$ - PriorDistanceOffset + DTG$_i$

i = nn

end of while (i > 0)

Check Turn Validity

This 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.

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

Compute TCP Times

Beginning at the runway (the last waypoint), work backwards and compute the TTG to each TCP.

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

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

$Average\ Ground\ Speed = (Ground\ Speed_{i-1} + Ground\ Speed_i) / 2$

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

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

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

Compute TCP Latitude and Longitude Data

With the exception of the input waypoints, this functions computes the latitude and longitude data for all of the TCPs.

In Turn = false

Past Center = 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 $(TCP_i == turn\text{-}entry)$

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

Find the center of the turn.

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 $Track_{Turn\ Index}$ - 90.0

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)

RelativeLatLong(Center Latitude, Center Longitude, a1, x), returning Turn Data $Latitude_i$ and Turn Data $Longitude_i$

Compute the location for the center of the turn.

a2 = DeltaAngle(Turn Data $Track1_i$, Turn Data $Track2_i$)

if (a2 > 0) b = Ground $Track_i$ + 90

else b = Ground $Track_i$ - 90

Compute the latitude and longitude from *Turn Data Latitude*$_i$, *Turn Data Longitude*$_i$, the angle *b*, and the distance, *Turn Data Turn Radius*$_i$.

RelativeLatLon(Turn Data Latitude$_i$, *Turn Data Longitude*$_i$, *b*, *Turn Data Turn Radius*$_i$*), returning Turn Data Center Latitude*$_i$ *and Turn Data Center 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)

Secondary Function Descriptions

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 21. 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.

ComputeGndSpeedUsingTrack

This function computes a ground speed from track angle (versus heading), CAS, altitude, and wind data.

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 (absolute(r) > 0.8) r = 0.8 * r / absolute(r)*

heading = track + arcsine(r)

a = DeltaAngle(heading, Wind Direction)

TAS = CasToTas Conversion(CAS, Altitude)

$Ground\ Speed = (Wind\ Speed^2 + TAS^2 - 2.0 * Wind\ Speed * TAS * cosine(a)\ )^{0.5}$

ComputeGndSpeedUsingMachAndTrack

This function computes a ground speed from track angle (versus heading), Mach, altitude, and wind data.

CAS = MachToCas(Mach,Altitude)

Ground Speed = ComputeGndSpeedUsingTrack

ComputedGndTrk

This 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.

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

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

else

$a = (1.0 - (distance - \text{DTG}_{i+1}) / d) * DeltaAngle(Ground\ Track_{i-1}, Ground\ Track_{i+1})$

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

ComputeTcpCas

The variable *cc* is passed into and out of this 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} *≠ Mach CAS Transition))*

Determine if the previous constraint cannot be met.

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

If this is the last pass through the algorithm, set 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.

Initial condition is the previous TCP.

$pc = cc - 1$

while ((pc > index number of the first waypoint) and (TCP_{pc} *≠ Mach CAS Transition)*
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,

$if\ (WptInTurn(k))\ Track = Ground\ Track_k$

$else\ Track = Ground\ Track_{k-1}$

$Current\ Ground\ Speed = ComputeGndSpeedUsingTrack(Test\ Speed,\ Track,\ Altitude_k, Wind\ Speed1,\ Wind\ Direction1)$

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*

Recompute 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 * d) * tangent(Crossing* $Angle_k$*) +* $Altitude_k$

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

InterpolateWindAtDistance(AltD, dx, Wind Speed2, Wind Direction2)

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

Track2 = GetTrajGndTrk(DTG_k *- dx)*

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 (absolute(*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 headwinds 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.

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

Determine if the constraint is met.

if ((k-1) = pc)

Was the crossing speed met within 1 kt? If not, set this as an error.

if (absolute(CAS_{pc} *- Crossing* CAS_{pc}*) > 1.0) 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.

$DTG_k = d$

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 *ComputedGndTrk* are described in this sections.

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

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 variable cc is passed into and out of this 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 $(TCP_{pc} \neq Mach\ CAS\ Transition)$
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 $(Prior\ Speed \leq Test\ Speed)$

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.

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)*

The ground speed at the prior altitude and speed.

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 $(Altitude_k >= Altitude_{k-1})\ AltD = Altitude_k$

else $AltD = (6076 * d) * tangent(\ Crossing\ Angle_k) + Altitude_k$

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

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

InterpolateWindAtDistance(AltD, dx, Wind Speed2, Wind Direction2)

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

$Track2 = GetTrajGndTrk(DTG_k - dx)$

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

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

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

    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,

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

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

Compute the average Mach rate.

$MRate1 = CasToMach(Crossing\ Rate_{cc}, Altitude_k)$

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

$Test\ Rate = (MRate1 + MRate2) / 2$

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 $(absolute(Mach_{pc} - Crossing\ Mach_{pc}) > 0.002)$
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.

$DTG_k = d$

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 *ComputedGndTrk* are described in this sections.

if (WptInTurn(k)) $Ground\ Track_k$ *= ComputedGndTrk(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

This functions 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 ≥ a ≥ 360

if (a > 180) a = a - 360

EstimateNextCas

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

CAS = 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 ((absolute(d - dx) > 0.001) && (count < 10))

if (d > dx) CAS = CAS - size

else CAS = CAS + size

size = size / 2

The estimated time t to reach this speed,

t = (CAS - Test Speed) / Test Rate

The new ground speed,

Gs2 = CasToTas Conversion(guess, Altitude) - Head Wind2

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

count = count + 1

end of the while loop

Limit the computed *CAS*, if necessary.

if (CAS > Prior Speed) CAS = Prior Speed

EstimateNextMach

This is an iterative function to estimate the Mach value, *Mach*, at the next TCP. Note that this is no closed-form solution for this calculation. The input variable names in this description are from the calling function. 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 ((absolute(d - dx) > 0.001) && (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 each wind 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.

$$Wind\ Speed_{k,\ Altitude1} = ((1.0 - r) * Wind\ Speed_{Temporary,\ Altitude1}) + (r * Wind\ Speed_{i,\ Altitude1})$$

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

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

GetTrajectoryData

This 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

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

if (distance < 0) Ground Track = Ground Track$_{last\ waypoint}$

else if (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 ≤ 0.0) r = 0

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

*dx = r * DeltaAngle(Ground Track*$_i$*, Ground Track*$_{i+1}$*)*

Ground Track = Ground Track$_i$ *+ dx*

InterpolateWindAtDistance

This function 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

while ((i0 < (index number of the last waypoint - 1)) and (TCP$_{i0}$ *≠ input waypoint) and (Distance > DTG*$_{i0+1}$*)) i0 = i0 + 1*

i1 = i0

while ((i1 < index number of the last waypoint) and (TCP$_{i1}$ *≠ input waypoint) and (Distance > DTG*$_{i1}$*)) i1 = i1 + 1*

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.

$r = (DTG_{i0} - Distance) / (DTG_{i0} - DTG_{i1})$

$Wind\ Speed = ((1.0 - r) * Spd0) + (r * Spd1)$

$a = DeltaAngle(Dir0, Dir1)$

$Wind\ Direction = Dir0 + (r * a)$

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 <= Number of Wind $Altitudes_i$; k = k + 1)

if (Wind $Altitude_{i,\ k}$ <= Altitude) p0 = k

if ((Wind $Altitude_{i,\ k}$ >= Altitude)and (p1 = 0)) p1 = k

if (p1 = 0) p1 = Number of Wind $Altitudes_i$

If *Altitude* = *Wind $Altitude_{p0}$* or if *Altitde* = *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:*

$r = (Altitude - Wind\ Altitude_{p0}) / (Wind\ Altitude_{p1} - Wind\ Altitude_{p0})$

$Wind\ Speed = ((1 - r) * Wind\ Speed_{p0}) + (r * Wind\ Speed_{p1})$

$a = DeltaAngle(Wind\ Direction_{p0}, Wind\ Direction_{p1})$

Wind Direction = Wind Direction$_{p0}$ *+ (r * a)*

RelativeLatLon

This function 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, this is an error.

*else Longitude = BaseLon + (180 / pi *(tangent(Angle)* (log(r1) - log(r2))))*

WptInTurn

This 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.

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 Appendix A.

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. Olson, Wayne M.: *Aircraft Performance Flight Testing*, AFFTC-TIH-99-01, 2000.

Appendix A Example Data Sets

Input Trajectory Data

An example input trajectory data set is provided below. The Mach / CAS transition speed for this example is 300 knots. Note that Waypoint-18 is the runway threshold.

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	0.82	0
Waypoint-02	32.48133	-99.8635	0	0	0	0.8	0.25
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	240	0	1.0
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	0.75
Waypoint-14	33.10658	-97.0537	4300	1.8	190	0	0.75
Waypoint-15	33.03645	-97.0541	0	0	0	0	0
Waypoint-16	33.00561	-97.0542	2400	3.1	170	0	0.75
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 below.

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 below. 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	0.82	266.9	true	461.7	77.1	366.2696	3230.593
VTCP		37000	0.82	266.9	true	461.7	77.1	194.0326	1887.718
Turn-entry		37000	0.814	264.8	true	458.4	77.1	193.1277	1880.637
Input	Waypoint-02	37000	0.8	259.7	true	469.7	93.3	190.8595	1863.04
Turn-exit		37000	0.8	259.7	true	488.5	109.5	188.5913	1845.996
Turn-entry		37000	0.8	259.7	true	488.5	109.5	143.1244	1510.896
Input	Waypoint-03	37000	0.8	259.7	true	478.8	101	141.9039	1501.811

Table A3 (continued). Example of the trajectory output data.

TCP type	Identifier	Altitude	Mach	CAS	Mach Segment	Ground Speed	Track	DTG	TTG
Turn-exit		37000	0.8	259.7	true	468.8	92.6	140.6834	1492.538
Input	Waypoint-04	37000	0.8	259.7	true	468.8	92.8	127.1251	1388.423
VTCP		37000	0.8	259.7	true	469	93	125.6414	1377.032
MACH CAS		30595	0.8	300	false	486	93	105.528	1225.392
Input	Waypoint-05	28581	0.769	300	false	472.4	93.1	99.20118	1177.863
Turn-entry		25687	0.727	300	false	453.8	93.1	90.11265	1107.212
Input	Waypoint-06	24824	0.715	300	false	422.2	69.1	87.40335	1084.944
Turn-exit		23961	0.703	300	false	396.5	45.2	84.69404	1061.117
Input	Waypoint-07	19976	0.651	300	false	390.6	45.3	72.17835	946.627
Input	Waypoint-08	16474	0.61	300	false	392.3	45.4	61.18281	845.5085
Input	Waypoint-09	11700	0.558	300	false	397.8	45.5	46.18899	708.8793
VTCP		11648	0.558	300	false	397.7	45.5	45.74832	704.8911
Input	Waypoint-10	11000	0.443	240	false	326.6	45.5	40.19145	649.6558
VTCP		11000	0.443	240	false	326.6	45.5	39.80241	645.3679
Turn-entry		10743	0.441	240	false	326.4	45.5	38.74742	633.7369
Input	Waypoint-11	10385	0.438	240	false	314.3	21.8	37.28263	617.277
Turn-exit		10028	0.435	240	false	297.3	358.1	35.81784	600.0319
Input	Waypoint-12	7104	0.412	240	false	296.7	1	23.83597	454.794
VTCP		6312	0.406	240	false	295.9	1	20.59182	415.378
Turn-entry		5799	0.402	240	false	294	1	18.4906	389.7323
Input	Waypoint-13	5300	0.366	220	false	270	45.7	16.44533	363.6217
Turn-exit		4918	0.363	220	false	244.7	90.3	14.40006	335.0103
VTCP		4759	0.362	220	false	243.2	90.3	13.56449	322.682
Turn-entry		4500	0.333	203.3	false	223.1	90.3	12.20674	301.7185
Input	Waypoint-14	4300	0.31	190	false	186	135.3	11.1612	283.3168
Turn-exit		3956	0.308	190	false	173.7	180.2	10.11566	262.3908
Input	Waypoint-15	3009	0.303	190	false	172.4	180.2	7.238161	202.5426
VTCP		2794	0.302	190	false	172.2	180.2	6.583648	188.8699
Input	Waypoint-16	2400	0.268	170	false	151.2	180.2	5.387746	162.2466
VTCP		2147	0.267	170	false	151.1	180.2	4.670449	145.1618
Input	Waypoint-17	1495	0.197	127	false	107	180.2	2.622742	88.03505
Input	Waypoint-18	660	0.194	127	false	107.5	180.2	0	0

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE *(DD-MM-YYYY)*
01- 09 - 2007

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

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Abbott, Terence S.

5d. PROJECT NUMBER
L-70750D

5e. TASK NUMBER

5f. WORK UNIT NUMBER
411931.02.07.07

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
NASA Langley Research Center
Hampton, VA 23681-2199

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-2007-214899

12. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified - Unlimited
Subject Category 01
Availability: NASA CASI (301) 621-0390

13. SUPPLEMENTARY NOTES
Langley Technical Monitor: Anthony M. Busquets
An electronic version can be found at http://ntrs.nasa.gov

14. ABSTRACT
This document describes an algorithm for the generation of a four dimensional aircraft trajectory. Input data for this algorithm are similar to an augmented Standard Terminal Arrival Route (STAR) with the augmentation in the form of altitude or speed crossing restrictions at waypoints on the route. Wind data at each waypoint are also inputs into this algorithm. The algorithm calculates the altitude, speed, along path distance, and along path time for each waypoint.

15. SUBJECT TERMS
Aircraft operations; Approach spacing; Aircraft systems

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	56	(301) 621-0390