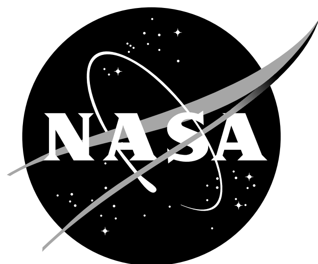
View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by NASA Technical Reports Server

NASA/TM-2017-219669



An Efficient Universal Trajectory Language

George E. Hagen, Nelson M. Guerreiro, Jeffrey M. Maddalon, Ricky W. Butler,
Langley Research Center, Hampton, Virginia

September 2017

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 NTRS Registered and its public interface, the NASA Technical Reports 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 counter-part of peer-reviewed formal professional papers but has 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:

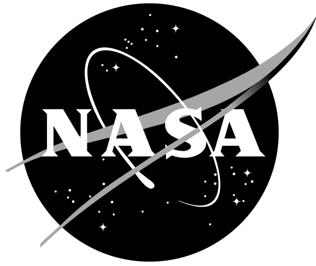
- Access the NASA STI program home page at http://www.sti.nasa.gov

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

- Phone the NASA STI Information Desk at 757-864-9658

- Write to:
  NASA STI Information Desk
  Mail Stop 148
  NASA Langley Research Center
  Hampton, VA 23681-2199

NASA/TM-2017-219669



An Efficient Universal Trajectory Language

George E. Hagen, Nelson M. Guerreiro, Jeffrey M. Maddalon, Ricky W. Butler,
Langley Research Center, Hampton, Virginia

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

September 2017

The use of trademarks or names of manufacturers in this report is for accurate reporting and does not constitute an official endorsement, either expressed or implied, of such products or manufacturers by the National Aeronautics and Space Administration.

Available from:

NASA STI Program / Mail Stop 148
NASA Langley Research Center
Hampton, VA 23681-2199
Fax: 757-864-6500

Abstract

The Efficient Universal Trajectory Language (EUTL) is a language for specifying and representing trajectories for Air Traffic Management (ATM) concepts such as Trajectory Based Operations (TBO). In these concepts, the communication of a trajectory between an aircraft and ground automation is fundamental. Historically, this trajectory exchange has not been done, leading to trajectory definitions that have been centered around particular application domains and, therefore, are not well suited for TBO applications. The EUTL trajectory language has been defined in the PVS formal specification language, which provides an operational semantics for the EUTL language. The hope is that EUTL will provide a foundation for mathematically verified algorithms that manipulate trajectories. Additionally, the EUTL language provides well-defined methods to unambiguously determine position and velocity information between the reported trajectory points. In this paper, we present the EUTL trajectory language in mathematical detail.

Contents

1 Introduction **1**

2 Related Work **3**

3 Trajectory Overview **6**
3.1 Trajectory Abstraction 7
3.2 Well-Formed Plans 9
3.3 Consistent Plans 10
3.3.1 Turn TCP Constraints 10
3.3.2 Ground Speed TCP Constraints 10
3.3.3 Vertical Speed TCP Constraints 11
3.4 Velocity-Continuous Plans 11

4 Language Elements **11**
4.1 The Plan Data Structure 11
4.2 EUTL Syntax 12

5 Informal Semantics of Position and Velocity Functions **13**
5.1 Non-accelerating Segments 13
5.2 Multiple Acceleration Zones 15
5.3 Path Distance Between Index Points 15
5.4 Interpolating Within Acceleration Zones 17
5.4.1 Computing Ground Speeds at Index Points 17
5.4.2 Determining Distance From a Time Within a Segment 18
5.4.3 Advancing a Specified Distance Within a Segment 18
5.5 Velocity at a Time 18
5.6 Vertical Component of Position 19
5.7 Summary of Calculation Steps 20

6 Operational Semantics **20**
6.1 Notation 20
6.2 Summary of Utility Functions 21
6.3 Calculating Path Distance 23
6.3.1 Straight Path 23
6.3.2 Circular Turn 23
6.3.3 Combined Function 24
6.4 Ground Speed Within a Segment 24
6.5 Determining Distance From a Time in a Segment 25
6.6 Advancing a Calculated Distance Within a Segment 26
6.6.1 Straight Linear Segment 26
6.6.2 Turn Segment 27
6.7 Full Position and Velocity Function 28
6.8 The `posVelWithinSeg` Function 28

6.9 The `initialVelocity` Function 29
6.10 The `interpolateAltVs` Function 29
6.11 Consistent Plans, Revisited 30
6.11.1 Track TCP Constraints 30
6.11.2 Ground Speed TCP Constraints 31
6.11.3 Vertical Speed TCP Constraints 31
6.12 Velocity-Continuous Plans, Revisited 32

7 TCP Generation **34**
7.1 `TrajGen.makeKinematicPlan` 35
7.2 `DistPlan.makePlan` 37

8 Generation of Medium Fidelity Trajectories **38**

9 Concluding Remarks **47**

A Support Functions **50**
A.1 TCP Functions 50
A.2 The getSegment Function 51
A.3 In Acceleration Zone Functions 51

B Great Circle Functions **52**
B.1 `angular_distance` 52
B.2 `distance` 52
B.3 `initial_course` and `final_course` 52
B.4 `angle_between` 53
B.5 `velocity_initial` 54
B.6 `linear_initial` 55

C Chordal Radius **56**
C.1 Function `chord_distance` 56

D Ground Speed vs. Airspeed **58**

Abbreviations

2D	2-Dimensional
3D	3-Dimensional
4D	4-Dimensional
AIDL	Aircraft Intent Description Language
ATN	Aeronautical Telecommunications Network
ATM	Air Traffic Management
BGS	Beginning of Ground Speed acceleration
BOT	Beginning Of Turn
BVS	Beginning of Vertical Speed acceleration
CAS	Calibrated Airspeed
EGS	End of Ground Speed acceleration
EOT	End Of Turn
EPP	Extended Projected Profile
EUTL	Efficient Universal Trajectory Language
EVS	End of Vertical Speed acceleration
FAA	Federal Aviation Administration
FMS	Flight Management Systems
IAS	Indicated Airspeed
RTA	Required Time of Arrival
RTCA	Radio Technical Commission for Aeronautics
TAS	True Airspeed
TBO	Trajectory Based Operations
TIGAR	Toolkit for Integrated Ground/Air Research
TCP	Trajectory Change Point

1 Introduction

In this paper we present a language for representing aircraft flight trajectories in a manner that allows for precision, simplicity, and efficiency. Although hundreds of papers have been written about trajectories and trajectory generation [1–8], we will not seek to provide a comprehensive treatment of the subject or even to develop a taxonomy of the plethora of approaches that have been proposed. Instead we will start from the desired basic mathematical properties and then proceed to develop the language along with a practical data structure that provides all of the needed functionality. We call this language the Efficient Universal Trajectory Language (EUTL). It is efficient because trajectories can be stored and communicated in a way that is reasonably compact and positions and velocities can be quickly calculated. The language is universal in that it does not require other aircraft systems or components, such as the flight management computer or flight control computer, to re-construct the trajectory for an aircraft. Universality is a important requirement because it allows trajectories to be exchanged between aircraft and ground-based systems without ambiguity. This paper presents a detailed, mathematical description of the EUTL language, which can be used to precisely define the trajectory for an aircraft. This paper does not focus on the problem of how to generate a trajectory to meet specified objectives or how to use trajectory information from a EUTL trajectory. Methods for constructing trajectories for different types of aircraft and various levels of fidelity are also not addressed here. This paper is only concerned with the problem of how to represent a trajectory in an unambiguous and efficient manner.

A moving object follows a *path*, or a sequence of positions, through space. When this path is augmented with a time for each position, it is referred to as a *trajectory* or a *four dimensional trajectory (4DT)*. In the literature, a path is sometimes referred to as a "3D trajectory." We avoid this terminology due to an intrinsic ambiguity. Does this 3D trajectory mean three spatial dimensions, without time or two spatial dimensions (typically latitude and longitude) and time? For this reason, we will refer to a sequence of positions, without reference to time, as a *path* and a path, of any dimension, that is augmented with times, as a *trajectory*.

Although the aviation and Air Traffic Management (ATM) communities have produced a large amount of work on trajectory generation and trajectory representation, there is a surprising lack of unity among the approaches. This could be because trajectories are fundamental to many different types of operations, each having a different focus. For instance, a pilot enters a sequence of waypoints into a flight management system (FMS), then adds a cruise speed and altitude. With this information, the flight management system can compute a reference path and/or a set of guidance commands necessary to meet the pilot's instruction. It is therefore not surprising that trajectories are sometimes defined as a series of these guidance commands along with their associated execution times. This approach works well if one is only concerned with communicating a trajectory between the flight management system and the flight guidance system. However, if a recipient external to this aircraft were to receive this sequence of guidance commands, it would have

to have a fairly detailed model of the flight guidance system of the sending aircraft to determine where the aircraft will be at any time. In other words, this abstract version of the trajectory must be converted to a more concrete version before it can be effectively used by the recipient. This process is typically called *trajectory prediction.*[1] All recipients of the abstract trajectory must have a detailed model of the aircraft dynamics in order to perform this trajectory prediction process. *We believe that trajectories that will be communicated to external agents should be represented in a form that does not require a complex trajectory prediction process.* It is for this reason that we introduce the EUTL language. Trajectories generated by any agent in the ATM system can be accurately translated to EUTL and then communicated consistently to all other participants. And these participants can interpret EUTL through use of the unambiguous mathematical semantics given in this paper. We have, therefore, sought to provide sufficient details so that readers of this paper could create their own implementation of EUTL on any desired platform.

A trajectory defined using EUTL only models the position and velocity state variables of an aircraft. Conceptually, a trajectory language could also model other state variables such as the attitude angles of the aircraft, the angular rates of these variables, or even higher order derivatives, such as positional or angular acceleration. We have deliberately chosen to not include these state variables in our language. The EUTL language is intended to be used for air traffic management systems that will be necessary for Trajectory Based Operations (TBO). In such systems, the accurate representation and communication of trajectories is the essential capability. Other state variables may be used in *generating* a trajectory, but we do not believe (at this time) that they will be needed by other systems that receive the trajectory. Future research can investigate if there is benefit to including other state variables in an air traffic management trajectory language.

In this paper we present a method for defining a relatively small set of data and associated functions, such that the trajectories can be computed in an accurate and efficient manner. Although many of these derivations may seem obvious, the choice was made to describe the data and functions completely instead of leaving often significant details unstated. This work builds on [9], where our trajectory definition did not allow ground speed acceleration zones to overlap with turns. The previous version also relied heavily on the use of Euclidean calculations and the use of mathematical projections to geodesic coordinates. Instead of projections, the new approach described in this paper uses spherical trigonometry because we have found this to be computationally faster and more accurate.

A formal semantics for EUTL has been developed in the PVS specification language [10, 11], which is presented in Section 6. We believe that this language can provide a foundation for mathematically verified algorithms that manipulate trajectories. There is much evidence that a major obstacle to the deployment of advanced software systems in the National Airspace (NAS) is the current lack of suitable

[1] Historically, the trajectory prediction process generates a trajectory from an aircraft's flight plan, its current state, and from wind and temperature data. A sequence of guidance commands provides additional information and enables better prediction.

verification technology that can establish the safety of software-intensive advanced systems [12]. Java and C++ implementations of EUTL have also been developed and have been used to store and manipulate trajectories in the Stratway program [13] and for fast-time conflict-detection studies [14, 15].

2 Related Work

In this section we will look at some of the past methods used to describe trajectories. We will not delve into the massive and complex set of approaches to *trajectory generation*. Instead, we will restrict our attention to methods used to specify the *output* from a trajectory generation process, whether by a person or a machine. According to the International Civil Aviation Organization, a trajectory is, “a description of the movement of an aircraft, both in the air and on the ground, including position, time and, at least via calculation, speed and acceleration” [16]. In the air traffic management literature, a wide number of approaches to trajectory representation have been described:

- a sequence of 3D points (latitude, longitude, altitude) with a small, fixed time interval between them (e.g., 1 second) and a start time. Trajectories like this are referred to as *trajectory traces* or *raw trajectories* and may be represented as a sequence of uniformly-sampled 4D points [17];

- a sequence of 4D points (latitude, longitude, altitude, time) with large and irregular time intervals between them (e.g., 15 minutes). There are many different approaches to interpolating between them including straight-line, Lagrange, Hermite, Bezier, homotopy, and various spline methods [18];

- a sequence of timed guidance commands [19];

- a sequence of 2D points with a separate speed profile and vertical profile that are functions of path distance [1];

- a sequence of 2D points with speed/altitude constraints and perhaps a final required time of arrival (RTA) [20];

- a sequence of 4D points (latitude, longitude, altitude, time) that has a continuous velocity vector by adding information about accelerations (e.g. EUTL);

Probably the most basic notion of a trajectory for air traffic management is the flight plan that is filed with the Federal Aviation Administration (FAA) for a particular flight. This flight plan route defines a set of waypoints that will be sequenced by the aircraft during its flight. This is augmented with some information about the expected departure time, cruise altitude, and target airspeed for that flight. This provides a very crude prediction of where an airplane will be at any given time in the future. At the other end of the spectrum is the *reference trajectory* that is calculated by a flight management system. This is the most precise, most detailed representation of the trajectory of any entity in the system. However, this

trajectory is contained in an internal data structure of the FMS and is traditionally not exported to other agents in this form.

Another notion of a trajectory is a sequence of guidance commands such as `HOLD TO AN ALTITUDE`, `CONSTANT RADIUS TO A FIX`, `COURSE TO AN INTERCEPT`, or `DIRECT TO FIX`. These guidance commands are expected to be generated by an FMS and sent to a flight guidance system/flight control system [21]. This notion of a trajectory can also be used to communicate aircraft trajectory information to other airspace users [22]. Determining the exact trajectory represented by these guidance commands requires a detailed model of the guidance and control systems of the aircraft. Without corresponding precise models, the exact trajectory is not known. This can be considered an advantage in that the guidance commands are independent of a specific manufacturer's implementation. Unfortunately this leads to a greater error between the trajectory as flown and the trajectory represented by the guidance commands. Depending on the intended purpose of the trajectory, this error may or may not be acceptable.

There have been many proposed mechanism to specify a trajectory in a manner that enables its communication to other agents in the ATM system. We will look at only a few. We begin with Paielli's approach presented in [1] because this approach provides a set of requirements that they believe should be met by any method that seeks to specify a trajectory for air traffic operations. They state that the basic requirements of a trajectory representation should be:

1. able to precisely specify any *reasonable* 4D reference trajectory;
2. able to precisely specify error tolerances relative to the reference trajectory;
3. based on a global earth-fixed coordinate system (e.g., the WGS84 geodetic coordinate system);
4. parametric and reasonably compact;
5. based on a text format readable by humans; and
6. suitable for an international standard.

We concur that these requirements move the field of trajectory specification towards a more solid foundation. We also add to that list the following requirement:

7. the method for calculating position and velocity at any point in the trajectory should be specified in detail.

Paielli then presents an XML-based language that can be used to specify a trajectory in a way that meets all of these requirements. The horizontal path of such a trajectory is specified using a sequence of straight, great circle segments connected by turns of a specified radius. The along-track position is specified as a polynomial function of time; altitude is a polynomial function of along-track position. These polynomials are obtained from trajectory traces by curve fitting and are not limited to any particular order of polynomial. This method is similar to ours in that

it enables a mathematically precise determination of where an aircraft is at any time. There is a close connection between his approach and ours for the 2D path; however, altitude and speed are handled very differently. He makes the following observation [1]:

> "Suppose, for example, that an aircraft is on approach for landing and is one minute behind schedule (but still within tolerance). If altitude is specified as a function of time, the aircraft will be required to land several miles before it reaches the runway! On the other hand, if altitude is a function of along-track position, the aircraft will be required to land at the runway regardless of its status with respect to its schedule."

This design choice has some domain specific advantages. However, there are some trade-offs. For example, one cannot perform conflict detection without having a precise mapping from time to 3D position. We also note that in his approach, along-track position is specified as a polynomial function of time. So this part of his trajectory specification must be recalculated in the very context he is describing. Nevertheless, this decomposition approach may provide some computational efficiencies. Here we see why there are so many different approaches to trajectory specification. What is given priority or what is ignored in a specification is dependent upon where it is used. Tools can be constructed that generate trajectories given a myriad of different combinations of constraints. But in the end, that resulting trajectory should be a mathematical function from time to position[2]. One of the uses for the EUTL is described later in this paper, where we developed a tool that takes altitude as a function of path distance and speed as a function of path distance and generates a trajectory in our specification language.

The next approach to trajectory specification that we would like to consider is the Aircraft Intent Description Language (AIDL) [19, 23]. AIDL is essentially an abstraction of the most common guidance commands that are available on a transport class aircraft. It is therefore the input to a guidance or control system and so does not meet requirement (1) of the Paielli requirements. This language is suitable for describing the output of an FMS, but reconstructing an accurate trajectory of the aircraft requires a detailed model of the guidance and/or control system of the aircraft.

Another example of a trajectory description that has been defined for air traffic operations is the Extended Projected Profile (EPP). The EPP has been defined in the RTCA Data Communications standards for the Aeronautical Telecommunications Network (ATN), Baseline 2 as a method for the exchange of trajectory information between an aircraft and ground automation [24,25]. Appropriately-equipped aircraft are expected to be able to translate their FMS reference trajectory into the EPP specification and provide that EPP trajectory to ground automation as needed or as required.

The EPP specification calls for a set of up to 128 trajectory points with information such as the 2D reference position, the reference altitude, the predicted time,

[2] The closely related function from time to velocity is also defined in our operational semantics.

the predicted airspeed, as well as other optional information, such as turn radius and speed, altitude, or time constraints. Although this specification is intended to be a detailed description of the planned trajectory for an aircraft, it has inherent ambiguities. First, the points provided in the EPP trajectory provide estimates for the times at those reference points but assumptions have to be made in order to estimate the position of the aircraft at times between available trajectory points. Even though the EPP points provide estimated speed information, these speeds are indicated airspeeds, which require the use of wind and temperature forecast tables to properly convert to ground speeds that can be used to estimate the positions between points. As a consequence, significant along-track errors may be present in an EPP-described trajectory, especially in trajectories containing long segments between turns. Lateral turn points are provided in terms of the reference 2D point (e.g., the sequenced waypoint), an estimated altitude and time, an expected airspeed, and the predicted turn radius. Depending on the coordinate system used, the computations to estimate other points of the turn, such as the expected start and end of turn, which are important when the aircraft is using a fly-by maneuver to sequence a waypoint, can produce somewhat different positions and estimated times. Because of potential significant along-track error and a lack of rigorous mathematical definition and the ambiguities it creates, the EPP trajectory specification may not be the best suited trajectory description for performing certain air traffic system functions, such as conflict detection.

In the following sections we describe EUTL as well as its associated mathematical properties and functions, which make EUTL a mathematically unambiguous description of a trajectory.

3 Trajectory Overview

A trajectory in EUTL is modeled as a function of time into position and denoted, $\mathbf{s}(t)$. The velocity is also defined along all points of the trajectory and it is denoted as $\mathbf{v}(t)$. Trajectories can come in one of two forms, one where the positions are in Euclidean coordinates (x, y, z) or one where the positions are in geodesic coordinates (latitude, longitude, and altitude). Euclidean coordinates are often more appropriate when modeling trajectories of aircraft that fly in small localized areas, such as small unmanned aircraft. Geodesic coordinates are more appropriate when modeling long distance trajectories in the National Airspace System. The specification of many of the properties of a trajectory does not rely on a particular coordinate system; however, the low-level functions that compute positions and velocities do.

The position and velocity functions are, in some sense, the essence of the specification of a trajectory. Nevertheless, the equations that define these functions, given a sequence of 3D positions and time, are non-trivial and hence are developed in two steps. First, we define a path function as a function from distance along the path to position and denote this function as $\mathbf{p}_{\mathsf{3D}}(d)$. Next, we define a function $d_{\mathsf{3D}}(t)$ that maps time to total path distance. These functions can then be composed to obtain $\mathbf{s}(t) = \mathbf{p}_{\mathsf{3D}}(d_{\mathsf{3D}}(t))$. One advantage of this decomposition is that the path of the air-

craft can be specified separately from the speed of the aircraft. With the complete specification of these two functions, then $\mathbf{s}(t)$ and ultimately $\mathbf{v}(t)$ can be defined. We note that this decomposition also enables us to use distance as an alternative index for trajectories.

Typically, navigation is divided between horizontal planning and vertical planning. Therefore, a single three dimensional path distance function is usually not the focus of ATM service providers. They typically measure distance between points on a map (horizontal path distance), which does not include distances obtained from altitude changes. We therefore separate the horizontal and vertical dimensions. The vertical dimension is simpler than the horizontal, so a decomposition into two subfunctions is not necessary. The vertical position $z(t)$ is defined directly as a function of time.

$$\mathbf{s}(t) = (\mathbf{p}(d(t)), z(t))$$

We make the assumption that the horizontal path of the aircraft only consists of combinations of straight and circular segments, so the horizontal path function, $\mathbf{p}$, can be defined in a straightforward manner (details are provided in Section 5.7).

Next we turn our attention to the horizontal path distance function, $d(t)$. This function fundamentally relies on the horizontal speed. As an example, if the ground speed is constant, then the path distance function could be defined as

$$d(t) = v_{\mathsf{gs}} \min(t, T)$$

where, v_{gs} is the ground speed and T is the end time of the trajectory. A realistic trajectory with many speed changes has a much more complicated $d(t)$ function. With this background, we can turn to how to encode the key information of a trajectory.

3.1 Trajectory Abstraction

Instead of arbitrary functions $\mathbf{s}$ and $\mathbf{v}$, we provide five key assumptions that enable the encoding of these functions to be efficient in both space (relatively few bytes of information are required to store or communicate them) and time (they can be reconstructed quickly). These assumptions, several of which were alluded to in the discussion above, are:

1. the trajectory is divided into segments—sections of the trajectory partitioned by time;

2. a segment is either an accelerating segment or a non-accelerating segment;

3. viewed from above, all segments are either straight or circular arcs, and arcs always have associated angles of less than 180°;[3]

4. all changes in ground speed are represented via constant acceleration segments; and

[3]Larger arcs can be created using a sequence of consecutive arcs each having associated angles of less that 180°.

5. all changes in vertical speed are represented via constant acceleration segments.

Real aircraft are subject to non-constant accelerations and may follow paths that are neither straight nor circular; however, with a sufficient number of segments, this structure can model very complex trajectories. Future work will characterize the accuracy of the EUTL approach in representing trajectories used in different application domains.

The EUTL trajectory abstraction is a time-sequence of 3D points where some of these points are labeled as Trajectory Change Points (TCPs). Since the abstraction relies on points and not functions, we introduce notation to describe these points. The symbols $\mathbf{s}_i, \mathbf{v}_i, t_i$ represent the position, velocity, and time of the ith point in the trajectory, respectively. A *segment* within a trajectory refers to the interval between points, so the ith segment refers to the interval between the ith and $i^{\text{th}}+1$ points in the trajectory. The presence of absolute time values in the trajectory implicitly defines an average ground speed over the segment. Although aircraft performance fundamentally relies on airspeed, many ATM functions depend on accurate predictions of aircraft locations relative to each other. For a discussion about the issues associated with ground speed vs. airspeed see Appendix D.

The TCPs represent the start or end of a segment over which there exists a constant acceleration. There are three types of acceleration: turns, ground speed (horizontal) acceleration, and vertical speed acceleration. Furthermore, we must indicate whether a TCP is the beginning or end of an acceleration segment.[4] To summarize, the TCPs defined in the EUTL language are:

`BOT`	beginning of turn
`EOT`	end of turn
`BGS`	beginning of ground speed acceleration
`EGS`	end of ground speed acceleration
`BVS`	beginning of vertical speed acceleration
`EVS`	end of vertical speed acceleration

The beginning TCP also encodes the exact acceleration value; in the case of a turn, this is the turn radius and center of turn position[5]. The end TCP provides the duration of acceleration, as well as a position reference for checking the acceleration calculations. This concept is illustrated in Figure 1. Each of the dots represent a 3D waypoint. The colored waypoints are TCPs. The black waypoints could represent established reference navigation fixes or other points of reference such as RTA points, or could be redundant and optionally eliminated.

In EUTL a trajectory is encoded as a sequence of points, with some designated as the beginning or end of an acceleration zone. We refer to this structure as a *plan*. A trajectory is an abstract concept, whereas a plan is a structure that can be encoded in software. Much of the discussion in the rest of this paper is about plans.

[4]The inclusion of both a beginning and end point, as opposed to only a beginning point with a duration parameter, allows for additional easy consistency checking of an encoded trajectory.

[5]The center of turn is stored for computational efficiency.

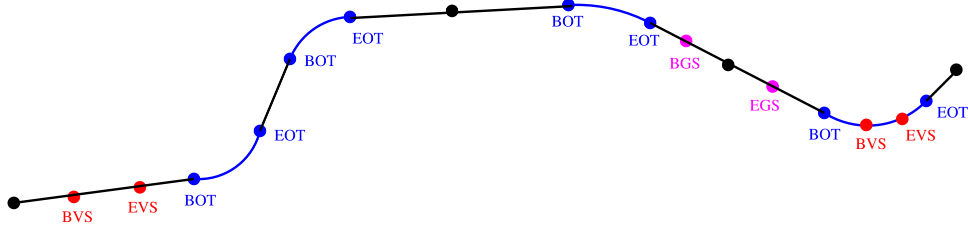


Figure 1: Example Trajectory with TCPs

If a plan does not contain any acceleration segments we say it is a *linear* plan. If it contains at least one acceleration segment, then it is said to be a *kinematic* plan. Because there is a time associated with each point in a plan, an average velocity is implicitly defined for each segment; however, for kinematic plans the velocity may vary over an accelerating segment. A linear plan typically has a discontinuous velocity at the index points, whereas a kinematic plan (if constructed properly) will have a continuous velocity vector throughout the plan.

3.2 Well-Formed Plans

Not all sequences of points form an arrangement that properly defines a trajectory. Certain restrictions need to be placed on a sequence of points to ensure that a trajectory can be formed from them. We refer to these conditions as: well-formed, consistent, and velocity-continuous. These conditions are described in this section and the next two sections.

A plan is *well-formed* if:

- there are at least two points in the plan;
- each point is appropriately ordered by time (that is, $\texttt{time}(i) < \texttt{time}(i+1)$);
- every beginning TCP has a corresponding ending TCP; and
- there are no two acceleration zones of the same type that are overlapping in time.

Given the three types of TCP-pair segments (turn, ground speed, and vertical speed), we allow different segment types to overlap in time, but segments of the same category cannot overlap. For example, a beginning of turn TCP (`BOT`) must be followed at some point later in the plan by a corresponding end of turn TCP (`EOT`) and this zone must not overlap another `BOT`/`EOT` pair. This must also be true of all (`BGS`/`EGS`) and (`BVS`/`EVS`) pairs. As a special case, one acceleration zone can end at precisely the same point where another acceleration zone of the same type begins. This is accomplished through use of the special combination TCPs: `EOTBOT`, `EGSBGS`, and `EVSBVS`.

3.3 Consistent Plans

A plan is *consistent* when:

- it is well-formed; and
- every end TCP point matches the position kinematically-calculated from its corresponding beginning TCP point and its acceleration value.[6]

In essence, consistent plans ensure that the mathematical relationship between the beginning TCP point and the ending TCP point is maintained. The trajectory generation process can be quite complex and latent errors in this complexity may cause the formation of inconsistent plans. The mathematical constraints that define consistency for each of the three types of TCPs are presented in the next three subsections.

3.3.1 Turn TCP Constraints

The BOT and EOT positions and associated acceleration data must be consistent with each other. A constant-acceleration turn is described by a radius and center of turn, which defines the turn arc from the BOT point to the EOT point. The radius is stored as a signed value, where the sign encodes the turn direction (a positive radius is clockwise, negative counterclockwise) and the magnitude is the actual radius value. For a plan to be considered consistent, it is necessary that both the BOT and EOT points are on this circle. In addition all other points between BOT and EOT should be on this circle. Each of these tests is accomplished by measuring the distance from the point to the center of turn and ensuring that this distance is the same as the radius.

Because the only information stored about a turn is the radius and turn center, not all information contained in the BOT and EOT is tested in this consistency check. Specifically, neither the altitude, nor the time components are checked because these values are not constrained by the turn structure.

3.3.2 Ground Speed TCP Constraints

The information stored for a ground speed change in segment i is the ground speed acceleration value, denoted $a_{\mathtt{gs}_i}$. The path distance between BGS and EGS must satisfy the following equation:

$$\text{path distance} = v_{\mathtt{gs}_i}\Delta_t + \frac{1}{2}a_{\mathtt{gs}_i}\Delta_t^2,$$

where Δ_t is the time between BGS and EGS and $v_{\mathtt{gs}_i}$ is the ground speed at the beginning of the segment, that is, the ground speed coming out of BGS. The path distance refers to the two-dimensional path distance and this path distance is defined assuming the plan is turn consistent, per Section 3.3.1.

[6] We note that this calculation can be complicated if this TCP pair overlaps with other TCP pairs of a different type.

3.3.3 Vertical Speed TCP Constraints

The information stored for a vertical speed change is the vertical speed acceleration value, denoted a_{vs_i}. The vertical path distance between the BVS/EVS pair must satisfy the following equation:

$$\text{vertical distance} = v_{\mathsf{vs}_i}\Delta_t + \frac{1}{2}a_{\mathsf{vs}_i}\Delta_t^2,$$

where Δ_t is the time between BVS and EVS and v_{vs_i} is the vertical speed at the beginning of the segment, that is, the vertical speed coming out of BVS. The vertical distance is simply the change in altitude from BVS to EVS.

3.4 Velocity-Continuous Plans

A *linear* plan can contain instantaneous velocity changes at the index points; however, conventional aircraft cannot make instantaneous velocity changes. Although low momentum vehicles, such as rotorcraft or quadcopters, can perform nearly instantaneous velocity changes (i.e., direction changes with no turn radius), we constrain our discussion in this section to trajectories with continuous velocity vectors at all points. We call such plans *kinematic* plans. We note, however, that the trajectories of low momentum vehicles can easily be be described using the EUTL language.

The velocity of a trajectory is said to be continuous at some point i provided the velocity into point i is the same as the velocity coming out of point i. The requirement that the velocity be the same means that every component of velocity (track angle, ground speed, and vertical speed) must be the same.

We note that linear plans typically have many velocity discontinuities whereas properly-constructed kinematic plans are velocity-continuous. Observe that plan consistency is primarily a restriction on the positions of points, sometimes called the structural aspects of the plan, whereas velocity-continuous plans are a restriction on the velocity at points, and velocity relates to times of the points.

4 Language Elements

Section 3 provides an overview of trajectories and how trajectories can be encoded into a form that is suitable for storing in a computer or communicating between computers. This section provides a more detailed description of how a plan is stored and the basic functions used with a plan. The key functions of position and velocity are given sections of their own. This section provides many of the important details necessary to understand the sections describing position and velocity.

4.1 The Plan Data Structure

We will now use Plan to refer to the pseudo-code operational specification of a plan. The notation used here is based on the PVS specification language. A Plan consists of these lists:

```
points: [nat -> NavPoint]
data:   [nat -> TcpData]
```

A `NavPoint` is a data structure that consists of 3D position, a time, and a string name. It can be defined as follows:

```
(p:Position, time: real, name: String)
```

A `Position` can be defined with either Euclidean coordinates, with the function `newPositionXYZ`, or geodesic coordinates, with the function `newPositionLLA`, which thus provides a common interface and allows most other software to ignore the detail of the true nature of a position. `Position` is effectively a disjoint union of the two types. In the software, the type coercions to the appropriate type are needed, but in this paper, we leave these out to simplify the presentation. Similarly, given a `NavPoint np`, the expression `position(np)` is often just written as `np`. However, `time(np)` will always be explicit. The method `isLatLon(p)` returns a boolean value indicating whether the position is geodesic (latitude, longitude) or not. If a position `p` is Euclidean the accessors `x(p)`, `y(p)` and `z(p)` can be used to retrieve the components. If the position is geodesic, then the accessors `lat(p)`, `lon(p)`, `alt(p)` are used.

In addition to the `NavPoint`, each point also has a `TcpData` data structure. This structure contains the information about whether the point is a TCP point or not. Furthermore, if the point is a TCP point, then this structure stores whether it is a beginning or ending TCP, and what type of TCP it is (track, ground speed, or vertical speed). Finally, if the point is a `BOT`, then the radius and center of turn are also stored, if the point is a `BGS` then the ground speed acceleration is stored, and if the point is a `BVS` then the vertical speed acceleration is stored in `TcpData`.

One observation made from this definition of `Plan` is that the velocity is not explicitly stored. Instead, each point contains a time and, given knowledge of the acceleration regions in a `Plan`, the velocity is *implicitly* defined for all positions between these points. The computations used for velocity are foundational to the definition of a trajectory and thus the specification of these equations becomes critical to the definition of the trajectory.

4.2 EUTL Syntax

In this section we present a simple syntax of the Efficient Universal Trajectory Language. While we are not concerned with a particular encoding of the elements of the language, we provide a representative syntax to illustrate the main aspects of this information. The syntax of EUTL is extremely simple: a semicolon-separated list of points, with point names and TCP data being included if appropriate. The language grammar is shown in Figure 2.

Quoted terms along with parentheses, semicolons, and commas indicate literal values. Terms in all capital letters represent numeric or string values. Expressions contained in square brackets ([and]) are optional. This data can easily be represented in various formats, a human-readable one being a simple text table. An example of such is shown in Section 8 and Figure 20.

plan	::=	LATLON? *point_list*
point_list	::=	*point* *point_list*
	::=	*point* *point*
point	::=	*navpoint* *tcpdata*
navpoint	::=	TIME (*position*) [NAME]
position	::=	LAT, LON, ALT | X, Y, Z
tcpdata	::=	[(*trk_data*)] [(*gs_data*)] [(*vs_data*)]
trk_data	::=	*trk_type* RADIUS | "EOT"
gs_data	::=	*gs_type* GS_ACCEL | "EGS"
vs_data	::=	*vs_type* VS_ACCEL | "EVS"
trk_type	::=	"BOT" | "EOTBOT"
gs_type	::=	"BGS" | "EGSBGS"
trk_type	::=	"BVS" | "EVSBVS"

Figure 2: EUTL Grammar.

5 Informal Semantics of Position and Velocity Functions

In this section we provide an informal introduction to defining position and velocity as functions of time for any point within a trajectory. This section is intended to provide the high-level mathematical overview, whereas in Section 6 detailed descriptions of algorithms that can be used to compute position and velocity are presented.

5.1 Non-accelerating Segments

In non-accelerating segments, the velocity is constant over the segment. Recall that accelerating sections of a plan begin and end with TCP points. These sections can extend over several segments. Any segments of well-formed plans that do not fall within any TCP regions are non-accelerating segments.

In a non-accelerating segment, for all points within the segment, the velocity is constant. As observed in Section 4.1, the velocity is not explicitly stored, so it must be computed. The velocity is defined as follows:

$$\mathbf{v}_i = \frac{\mathbf{s}_{i+1} - \mathbf{s}_i}{t_{i+1} - t_i}$$

when $t_i < t_{i+1}$. Since the velocity is constant over the segment, velocity as a function of time is defined as

$$\mathbf{v}(t) = \mathbf{v}_i, \text{ for } t_i \leq t < t_{i+1}. \tag{1}$$

In a similar way, for a segment with a constant velocity, the position $\mathbf{s}(t)$ at absolute time t within segment i is given as follows

$$\mathbf{s}(t) = \mathbf{s}_i + (t - t_i)\mathbf{v}_i, \text{ for } t_i \leq t < t_{i+1}. \tag{2}$$

These definitions are valid when the positions are Euclidean. If the coordinates are geodesic, then spherical trigonometry must be used instead of conventional vector algebra. In this formulation, aircraft travel in great circle paths. The position $\mathbf{s}(t)$ is given by

$$
\begin{array}{l}
\texttt{LET} \\
\quad \Delta_t = t_{i+1} - t_i \\
\quad \mathbf{v}_i = \texttt{velocity_initial}(\mathbf{s}_i, \mathbf{s}_{i+1}, \Delta_t), \\
\quad d = (t - t_i) \cdot \texttt{gs}(\mathbf{v}_i) \\
\texttt{IN} \\
\quad \texttt{linear_initial}(\mathbf{s}_i,\ \texttt{trk}(\mathbf{v}_i),\ d)
\end{array}
\tag{3}
$$

assuming, $t_i \leq t < t_{i+1}$. This definition relies on several other functions, including:

- `trk`($\mathbf{v}$) is the track component of velocity $\mathbf{v}$;
- `gs`($\mathbf{v}$) is the ground speed component of velocity $\mathbf{v}$;
- `velocity_initial` is the initial great circle velocity coming out of the initial point (see definition in Appendix B); and
- `linear_initial` is the position starting at the initial point and traveling d distance along the great circle in the given direction (see definition in Appendix B).

We note that the ground speed and vertical speed components of the velocity are constant (i.e., the same as $\mathbf{v}_i$ above), but the track will vary over the great circle path (in essence, over time). The velocity, $\mathbf{v}(t)$, at time t assuming, $t_i \leq t < t_{i+1}$ is given by

$$
\begin{array}{l}
\texttt{LET} \\
\quad \texttt{trk} = \texttt{final_course}(\mathbf{s}_i, \mathbf{s}(t)) \\
\texttt{IN} \\
\quad (\texttt{trk},\ \texttt{gs}(\mathbf{v}_i),\ \texttt{vs}(\mathbf{v}_i))
\end{array}
$$

This definition relies on two other functions:

- `vs`($\mathbf{v}$) is the vertical speed component of velocity $\mathbf{v}$; and
- `final_course` is the final track angle[7] going into the given point (see definition in Appendix B).

[7] For the purposes of this paper, the navigational concepts of a *course angle* and a *track angle* are interchangeable.

5.2 Multiple Acceleration Zones

Determining the position and velocity for segments within acceleration zones is more challenging and quite complicated in the presence of overlapping acceleration regions. Consider the trajectory shown in Figure 3. We want to find the position and velocity at any arbitrary time t in the plan. If time t falls within segment 3 (indicated by the dashed arrow), then one must deal with three different accelerations at the same time. Each of the points contains a 3D position and a time, though in the

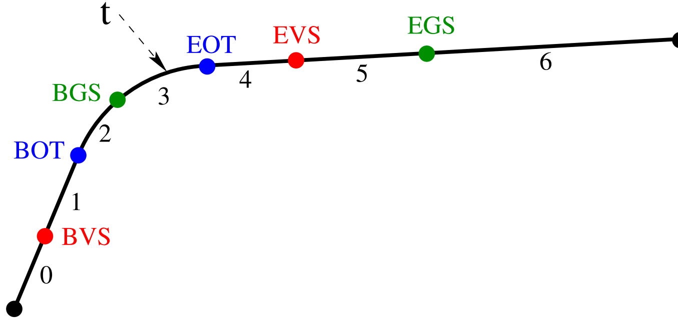


Figure 3: Overlapping Acceleration Zones

figure the altitude profile is not shown. It is clear from the figure that point t falls within segment 3; however, an algorithm must determine the segment number by scanning the times at the points. The order of computation is critical when determining the position and velocity at time t when there are multiple overlapping acceleration regions in the plan. A straightforward calculation, similar to equation 2 is not possible because the position in a turn is based on the path distance, the path distance is based on time and ground speed profile, and the ground speed profile is a function of path distance. To break this circular dependency, we start with a function that computes the circular path distance between the index points, rather than at an arbitrary point t. Next, we define a function for path distance by time, based on the initial ground speed at the beginning index point and the given ground speed acceleration. Combining these two results gives the horizontal position; the vertical position can be overlaid on this position. This method will only work within the segment containing the time point of interest. This calculation is described in more detail in the following sections.

5.3 Path Distance Between Index Points

If the points are straight (i.e., non-turning) segments, then the distance is simply the Euclidean distance or the great circle distance between two 2D points. If the segment is a turn segment, the path distance can be calculated as shown in Figure 4. The path distance is $d_i = \theta R$ where θ is the angle at the center of the turn. Recall that the turn radius and center position are stored for each turn. We note that the positions $\mathbf{s}_i$ and $\mathbf{s}_{i+1}$ may be points within a turn and therefore can be points other

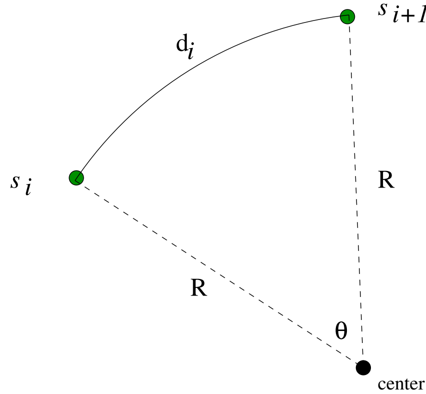


Figure 4: Path Distance of a Turn Segment

than a `BOT` or `EOT`.

There are two different notions of radius that arise when dealing with geodesic coordinates. They are:

- *Surface Radius*: the along-surface radius that is calculated using the great circle distance from the center of the turn to a point on the turn; and
- *Chordal Radius*: the part of a chord that is in the plane of the turn itself (which is an arc of a small circle) and hence passes through the sphere's volume.

For small-radius turns these are approximately the same, but for very large radius turns (on the order of hundreds of nautical miles), there can be a significant difference between the two values. These are illustrated in Figure 5.

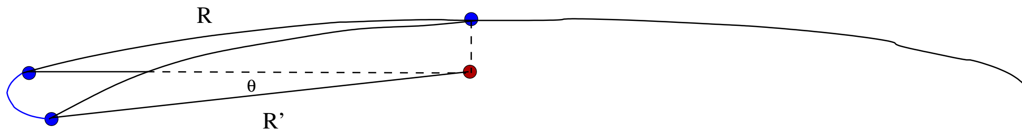


Figure 5: Example Surface Radius, R, and Chordal Radius, R'.

The surface radius is labeled R and the chordal radius is labeled R' in the figure. The turn circle is shown in blue. Note that turn circle is on the surface of the sphere but it is not a great circle (it may be referred to as a *small circle*). This small circle is also on the same plane as the two R' segments. Therefore the length (i.e., path distance) of the blue turn is $\theta R'$.

The radius used in the path distance calculation above should be the **chordal** radius. Appendix C provides the equations that can be used to convert the surface radius to the chordal radius and vice versa.

5.4 Interpolating Within Acceleration Zones

Next, we need a means to interpolate within two specified points. This requires that we develop a method to translate from distance to time. First we establish a way to compute the ground speed at each index point, see Section 5.4.1. If the segment is not within a BGS-EGS region, the speed over this segment is simply the path distance between the end points divided by the difference in times at the end points, as in equation 1.

5.4.1 Computing Ground Speeds at Index Points

We begin by recognizing that the ground speed in an EUTL plan is not necessarily continuous. In particular, the ground speed can be discontinuous at the segment boundaries in a linear plan. Therefore, we cannot rely on any ground speed information outside of the segment of interest. We distinguish between the ground speed at the beginning of the segment, denoted, v_{gs_i}, and the ground speed at the end of the segment, denoted, v_{final_i}. An example of these is shown in Figure 6. The speed v_{gs_i} can be thought of as the speed coming out of segment i and $v_{\mathsf{final}_{i-1}}$ can be thought of as the speed coming into segment i.

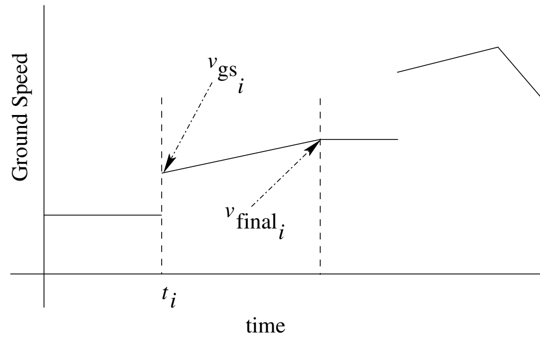


Figure 6: Relationship between Ground Speeds

If we define

$$\begin{aligned}
D_i &= \text{horizontal path distance between point } i \text{ and } i+1, \\
\Delta_t &= t_{i+1} - t_i, and \\
a_{\mathsf{gs}_i} &= \text{ground speed acceleration (possibly 0) over segment } i,
\end{aligned}$$

then the ground speed at the beginning of segment i is:

$$v_{\mathsf{gs}_i} = \frac{D_i}{\Delta_t} - \frac{1}{2} a_{\mathsf{gs}_i} \Delta_t, \tag{4}$$

and the ground speed at the end of segment i is:

$$v_{\mathsf{final}_i} = \frac{D_i}{\Delta_t} + \frac{1}{2} a_{\mathsf{gs}_i} \Delta_t \ .$$

These equations are easily obtained by solving for v_{gs_i} and v_{final_i} in the fundamental distance and velocity kinematic equations:

$$D_i = v_{\mathsf{gs}_i}\Delta_t + \frac{1}{2}a_{\mathsf{gs}_i}\Delta_t^2,$$
$$v_{\mathsf{final}_i} = v_{\mathsf{gs}_i} + a_{\mathsf{gs}_i}\Delta_t,$$

where v_{gs_i} is the initial velocity and v_{final_i} is the final velocity. In section 6, these functions are given longer, textual names: $\texttt{gsOut}(i)$ and $\texttt{gsIn}(i)$:

$$\texttt{gsOut}(i) = v_{\mathsf{gs}_i},$$
$$\texttt{gsIn}(i) = v_{\mathsf{final}_{i-1}}.$$

The plan is *continuous* at index point i if $\texttt{gsOut}(i) = \texttt{gsIn}(i)$.

5.4.2 Determining Distance From a Time Within a Segment

With the ground speed at the start of the segment (see Section 5.4.1), we can define the path distance d within segment i relative to $\mathbf{s}_i$ as follows

$$d_i(t) = v_{\mathsf{gs}_i} \cdot (t - t_i) + \frac{1}{2}a_{\mathsf{gs}_i} \cdot (t - t_i)^2, \tag{5}$$

where v_{gs_i} is defined by equation 4. This relative path distance in a segment can be used to find the final position by moving this distance from $\mathbf{s}_i$. If time t is not within a ground speed acceleration zone, then $a_{\mathsf{gs}_i} = 0$.

5.4.3 Advancing a Specified Distance Within a Segment

Finding a point within a straight segment by a specified distance is essentially an interpolation function. In the Euclidean case, linear interpolation accomplishes the task. In the geodesic case, there are great circle functions that interpolate between points on a great circle arc. The challenge is to define a function to find a point from a distance in a turn. Consider the turn segment shown in Figure 7. Here the distance $d_i(t)$ is the relative path distance, defined by equation 5, from the start of segment i to the point at time t. The angle α is simply $d_i(t)/R'$ and hence the direction from the center is easily determined. If one then calculates the point that is R units away from the center in the direction determined by α, then the final point is obtained. We note that the radius used in the angle calculation is the chordal radius R', while the distance from the center calculation uses the surface radius, R. See section 6.6.2 for details about this `turnByDist` method.

5.5 Velocity at a Time

Next, we turn to defining velocity as a function of time. Clearly it should be equal to the time-derivative at time t. However, we do not calculate it as a formal derivative. Instead we calculate the velocity components separately. That is, we calculate the track, ground speed, and vertical speed at point t.

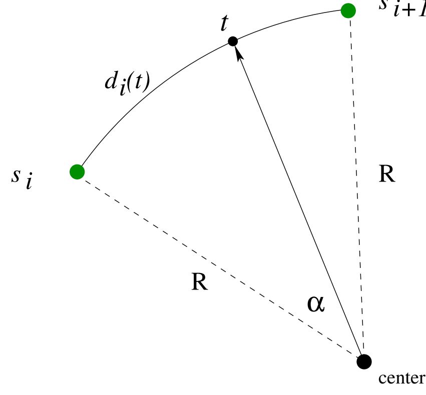


Figure 7: Advancement Within a Turn Segment.

The ground speed at t is derived from the kinematic function for velocity with a constant acceleration. Since we have the ground speed at the start of segment i (see section 5.4.1) the ground speed at time t is defined as:

$$\mathtt{gs}_i(t) = v_{\mathsf{gs}_i} + a_{\mathsf{gs}_i} \cdot (t - t_i),$$

where v_{gs_i} is defined by equation 4. Recall that v_{gs_i} is based on the current segment number, i, and it is assumed that $t_i \leq t < t_{i+1}$.

The track at time t is computed differently for the two cases. In straight segments, the final track is defined by the track angle when $\mathbf{s}_i$ is moved linearly by path distance d. In turn segments, the track angle is defined by the perpendicular to the line from the turn center to the point at t (see Figure 7).

The vertical speed at time t within segment i is given by:

$$\mathtt{vs}_i(t) = v_{\mathsf{vs}_i} + a_{\mathsf{vs}_i} \cdot (t - t_i)$$

where

$$v_{\mathsf{vs}_i} = \frac{z_{i+1} - z_i}{\Delta_t} - \frac{1}{2} a_{\mathsf{vs}_i} \Delta t$$

and z_{i+1} and z_i are the altitudes at the segment end points and $\Delta_t = t_{i+1} - t_i$. It is assumed that $t_i \leq t < t_{i+1}$. If the segment is not a vertical speed acceleration segment, then the vertical acceleration is zero, i.e., $a_{\mathsf{vs}_i} = 0$.

5.6 Vertical Component of Position

The definition of position in Section 5.4 only computes the 2D position (x and y or latitude and longitude). The vertical component of position is computed separately from the 2D position and then the components are integrated together. Since the vertical component is independent of the horizontal components, the function to perform the vertical calculations can be a linear interpolation. Given a time t

within a segment i (i.e., $t_i \leq t < t_{i+1}$), the altitude at this time is

$$z_i(t) = z_i + v_{\mathsf{vs}_i} \cdot (t - t_i) + \frac{1}{2} a_{\mathsf{vs}_i} \cdot (t - t_i)^2,$$

where a_{vs_i} is the vertical acceleration.

5.7 Summary of Calculation Steps

The approach for finding the position and velocity at time t can be summarized as follows, given a time t:

- find the segment number which contains time t;
- calculate the ground speed at the beginning of the segment and determine the value of the ground speed acceleration of this segment (possibly 0);
- using the ground speed data, determine the distance that will be traveled from the starting location of the segment;
- advance this distance into the segment to obtain the horizontal 2D position (the two cases are: (1) straight path and (2) circular path);
- calculate the altitude and vertical speed; and
- combine the horizontal and vertical components of position.

6 Operational Semantics

Section 5 provides a high-level, conceptual overview of how position and velocity functions are defined. In this section, we provide detailed definitions of these functions and thus provide an *operational semantics* for EUTL plans. By defining these functions in detail, we hope that our readers should be able to produce their own software implementation in a relatively straightforward manner.

The operational semantics of EUTL has been captured in the PVS formal specification language [10, 11]. We are also in the process of verifying our Java and C++ versions of EUTL against this formal specification using the model animation technique [26].

6.1 Notation

The basic syntax for expression definition is typical of functional programming languages and is as follows:

```
LET
    <id_1> = <expression_1>
    <id_2> = <expression_2>
        .
```

```
        .
        .
    <id_n> = <expression_n>
IN
    <expression>
```

The <expression> after the keyword IN defines the body of the function. It must be a function of previously defined functions and local values <id_X> that are defined between the relevant keywords LET and IN. The following illustrates how this is used to define a function:

```
circumference(radius): real =
   LET
      pi = 3.14159
   IN
      2.0*pi*radius
```

In this case the definition of the function circumference(radius) $=$ 2.0*3.14159*radius. Additionally, a function may return a tuple value consisting of more than one element:

```
  f(r: real): [real,real] =
      LET
         a = 100
         b = 200
      IN
         (5*a*r, b*r*r)
```

Subvalues of a tuple are extracted by implicit functions referring to their position in the tuple. In this case the value of $f(10).\texttt{first} = 5 \cdot 100 \cdot 10 = 5000$ and $f(1).\texttt{second} = 200 \cdot 1 \cdot 1 = 200$.

6.2 Summary of Utility Functions

The algorithms in Chapter 6 depend upon several utility functions whose meaning is straightforward, so only a short description is presented in this section. More complete definitions are given in Appendix A or B.

The set of utility methods in Table 1 retrieve information out of the Plan data structures.

The set of utility methods in Table 2 check if a point is of a particular TCP type.

The methods of Table 3 are those that search a plan for points of a particular type.

The final section of utility functions (Table 4) relate a particular time to the corresponding segment. All such functions return a negative value if no such segment exists within the given plan.

`time(p: Plan, i: int) : real`	Time of point i
`point(p: Plan, i:int) : NavPoint`	NavPoint at the given index
`signedRadius(p: Plan, i: int): real`	Chordal radius of turn, sign indicates turn direction
`turnCenter(p: Plan, i: int): Position`	Center of turn at point i
`gsAccel(p: Plan, i: int): real`	Ground speed acceleration
`vsAccel(p: Plan, i: int): real`	Vertical speed acceleration
`isLatLon(p: Plan): bool`	True iff plan is a geodesic plan
`size(p: Plan): int`	Number of points in the plan

Table 1: Basic Plan Functions.

`isTrkTCP(p:Plan,i:int):bool`	Is this point a turn change point?
`isBOT(p:Plan,i:int):bool`	Is this point a beginning of turn?
`isEOT(p:Plan,i:int):bool`	Is this point an ending of turn?
`isGsTCP(p:Plan,i:int):bool`	Is this a ground speed change point?
`isBGS(p:Plan,i:int):bool`	Is this a beginning of ground speed change pt?
`isEGS(p:Plan,i:int):bool`	Is this an ending ground speed change point?
`isVsTCP(p:Plan,i:int):bool`	Is this a vertical speed change point?
`isBVS(p:Plan,i:int):bool`	Is this a beginning vertical speed change point?
`isEVS(p:Plan,i:int):bool`	Is this an ending vertical speed change point?

Table 2: Determining Point Type.

`prevBOT(p: Plan, i: int): int`	Previous `BOT`	strictly before point i
`prevBGS(p: Plan, i: int): int`	Previous `BGS`	strictly before point i
`prevBVS(p: Plan, i: int): int`	Previous `BVS`	strictly before point i

Table 3: TCP Navigation Functions.

`getSegment(p:Plan,t:real):int`	Return the segment # that contains t
`inTrkChange(p:Plan,double t):bool`	True iff time t falls within a `BOT-EOT`
`inGsChange(p:Plan,double t):bool`	True iff time t falls within a `BGS-EGS`
`inVsChange(p:Plan,double t):bool`	True iff time t falls within a `BVS-EVS`

Table 4: Basic Plan Functions.

We note that our algorithms operate on plans defined with either Euclidean or geodesic coordinates. Although this complicates the implementation of our position and velocity functions, the resulting flexibility for higher level algorithms more than compensates for this complexity.

6.3 Calculating Path Distance

Based on the discussion in Section 5.3, there are two cases that must be covered to define the horizontal path distance, denoted d_i, between index points: straight path and circular turn.

6.3.1 Straight Path

The Euclidean distance is given by

$$\texttt{pathDistance}(i) = ||\mathbf{s}_{i+1} - \mathbf{s}_i||.$$

The geodesic distance is given by

$$\texttt{pathDistance}(i) = \texttt{GreatCircle.distance}(\mathbf{s}_i, \mathbf{s}_{i+1}).$$

The great circle functions are defined in Appendix B. The combined function is:

```
distanceH(p1: Position, p2: Position): nnreal =
   IF isLatLon(p1) THEN
      GreatCircle.distance(lla(p1),lla(p2))
   ELSE
      norm(vect2(p1) - vect2(p2))
   ENDIF
```

6.3.2 Circular Turn

We seek to calculate the horizontal path distance d_i of the segment shown in Figure 4. The path distance is given by:

$$d_i = \texttt{pathDistance}(i) = \theta R',$$

where R' is the chordal radius and $\theta = \texttt{angle_between}(\mathbf{s}_i, \texttt{center}, \mathbf{s}_{i+1})$ is defined by:

```
angle_between(p1: Position, p2: Position, p3: Position): real =
   IF isLatLon(p1) THEN
        GreatCircle.angle_between(lla(p1), lla(p2), lla(p3))
   ELSE
       LET A = p2.vect2() - p1.vect2()
           B = p2.vect2() - p3.vect2()
       IN
           acos((A*B)/(norm(A)*norm(B)))
   ENDIF
```

We note that the geodesic case is handled using the `GreatCircle` function `angle_between` defined in Appendix B.

6.3.3 Combined Function

The function that combines the linear and circular cases is

```
pathDistance(p: Plan, i:int, linear: bool): real =
   IF i < 0 OR i+1 >= size(p) THEN 0
   ELSE
     IF NOT linear AND inTrkChange(p, time(p,i)) THEN
        LET
          ixBOT = prevBOT(p, i+1)
          center = turnCenter(p, ixBOT)
          R' = signedRadius(p,ixBOT)
          theta = angle_between(point(p,i), center, point(p,i+1))
        IN
          abs(theta * R')
     ELSE
        distanceH(point(p,i), point(p,i+1))
     ENDIF
   ENDIF
```

The function `inTrkChange` determines whether this segment is within a turn. See Appendix A for its definition. The chordal radius is retrieved using the `signedRadius` function.

6.4 Ground Speed Within a Segment

In Section 5.4.1, equation 4, we developed the semantics of ground speed at an index point. In a full algorithm we must deal with a few technicalities. First, if i is the last point in a plan, we assume that the future velocity vector is equal to the previous ground speed in, that is, `gsIn`(i). Also, an algorithmic implementation must determine whether the segment is in a ground speed acceleration zone or not (i.e., between a `BGS-EGS` pair). If the segment is in such a zone, then the acceleration must be recovered from an earlier `BGS` point. More complete versions of `gsFinal` and `gsOut` are:

```
gsFinal(p: Plan, i:int, linear: bool): real =
   IF i < 0 OR i > size(p)-1 THEN -1
   ELSE
       LET
         dist = pathDistance(p, i, i+1, linear)
         dt = time(p,i+1) - time(p,i)
```

```
         a = IF inGsChange(p, time(p,i)) AND NOT linear THEN
                gsAccel(p,prevBGS(p,i+1))
             ELSE
               0.0
             ENDIF
       IN
         dist / dt + 0.5 * a * dt
   ENDIF

gsOut(p: Plan, i:int, linear: bool): real =
   IF i < 0 OR i > size(p)-1 THEN -1
   ELSIF i = size(p) - 1 THEN
      gsIn(p, i)
   ELSE
      LET dist = pathDistance(p, i, i+1, linear)
          dt = time(p, i+1) - time(p, i)
          a = IF inGsChange(p, time(p, i)) AND NOT linear THEN
                    gsAccel(p, prevBGS(p,i+1))
                 ELSE
                    0
                 ENDIF
      IN
          dist / dt - 0.5 * a * dt
   ENDIF
```

We note that numerical inaccuracies may lead to small negative ground speeds when implementing this function in floating point arithmetic. It may be necessary to guard against these undesirable return values. For convenience we define `GsIn` as follows:

```
gsIn(p: Plan, i:int, linear: bool): real =
   Gsfinal(p,i-1,linear);
```

6.5 Determining Distance From a Time in a Segment

Based on the discussion in Section 5.4.2, our approach to finding distance within a segment is based upon using path distance within a trajectory. Since we ultimately want to find the position and velocity in a plan at a specific time, we need to find the distance from the start of a segment to that time point. This is accomplished with the `distFromPointToTime` function:

```
distFromPointToTime(p: Plan, seg: int, t: real, linear: bool):real=
  LET  gs0 = gsOut(p, seg, linear)
       dt = t - time(p, seg)
  IN
```

```
IF inGsChange(p, t) AND NOT linear THEN
  LET a = gsAccel(p,prevBGS(p, seg + 1)) IN
      gs0 * dt + 0.5 * a * dt * dt
ELSE
      gs0 * dt
ENDIF
```

6.6 Advancing a Calculated Distance Within a Segment

To find a new point d distance from a given point, and based on the discussion in Section 5.4.3, there are two cases that must be covered: the segment is linear, and the segment is a turn.

6.6.1 Straight Linear Segment

```
linearDist2D(so: Position, track:real, d: real, gsAt_d: real):
                                              [Position,Velocity] =
   IF isLatLon(so) THEN
      LET sEnd = GreatCircle.linear_initial(so, track, d)
          finalTrk = IF d > minDist THEN
                       GreatCircle.final_course(so, sEnd)
                     ELSE
                       track
                     ENDIF
          vEnd = Velocity.mkTrkGsVs(finalTrk, gsAt_d, 0.0)
      IN
          (newPositionLLA(sEnd),vEnd)
   ELSE
      LET sNew = linearByDist2D(so,track,d)
          vNew = Velocity.mkTrkGsVs(track,gsAt_d,0.0)
      IN
          (newPositionXYZ(sNew),vNew)
   ENDIF
```

We note that for small distances d, the `GreatCircle` computations may produce inaccurate results. We have left out the details of how this is compensated for in the Java and C++ implementations. We also note that the `2D` in the name indicates that altitude and vertical speed are not computed by this function. These values are computed by the `interpolateAltVs` function. The function `linearByDist2D` is defined as follows

$$\texttt{linearByDist2D}(s:\texttt{Vect3},\ \ \texttt{track: real},\ d:\texttt{real}):\texttt{Vect3} = (s_x + d\cdot\sin(\texttt{track}),\ s_y + d\cdot\cos(\texttt{track}),\ s_z).$$

It calculates the 2D position after moving distance d units in the direction `track` and the altitude is not calculated.

6.6.2 Turn Segment

The parameters to the turn are:

- `so`: the position at the start of the turn (i.e., the `BOT`);
- `center`: the center of the turn stored in the `BOT` point;
- `dir`: the direction of the turn; and
- `gsAtd`: the ground speed at the end of the turn.

The last parameter `gsAtd` is passed in as a parameter because it usually has already been computed and this prevents a redundant calculation. The geodesic turn by path distance function is:

```
turnByDist2D(so, center: Position, dir:int, d: real, gsAtd: real):
                                                  [Position,Velocity] =
      LET R = GreatCircle.distance(so, center)
          R’ = GreatCircle.to_chordal_radius(R)
          alpha = dir*d/R’
          trkFromCenter = GreatCircle.initial_course(center,so)
          nTrk = trkFromCenter + alpha
          sn = LatLonAlt.mkAlt(GreatCircle.linear_initial(center,
                                 nTrk,R),0)
          final_course = GreatCircle.final_course(center,sn)
          finalTrk = final_course + dir*PI/2
          vn = Velocity.mkTrkGsVs(finalTrk,gsAtd,0.0)
      IN
          (sn,vn)
```

It returns a pair containing the final position and velocity. We note that the conversion of the radius to a chordal radius is optional, unless high accuracy is necessary. The Euclidean function is:

```
turnByDist2D(so, center: Vect3, dir: int, d: real, gsAtd: real):
                                                  [Vect3, Velocity] =
  LET R = distanceH(so, center) IN
    IF R = 0 THEN (so, INVALID)
    ELSE
      LET alpha = dir*d/R
          trkFromCenter = track(center, so)
          nTrk = trkFromCenter + alpha
          sn = mkZ(linearByDist(center, nTrk, R), 0.0)
          finalTrk = nTrk + dir*PI/2
          vn = Velocity.mkTrkGsVs(finalTrk,gsAtd, 0.0)
      IN
```

```
            (sn,vn)
      ENDIF
```

The function `track(center, so)` returns the track angle from point `center` to `so`. The function `LatLonAlt.mkAlt(lla,z)` returns a geodesic coordinate with latitude and longitude the same as in `lla`, but with altitude `z`. The function `mkZ` performs the same operation on a Euclidean coordinate frame.

6.7 Full Position and Velocity Function

```
 positionVelocity(p: Plan, t: real, linear: bool):
                                                   [Position, Velocity] =
   IF (t < time(p,0)) THEN (INVALID, Velocity.ZEROV)
   ELSE
      LET seg = getSegment(p,t)
          np1 = point(p,seg)
      IN
        IF (seg + 1 > size(p) - 1) THEN
           LET v = finalVelocity(p, seg - 1) IN
               (np1, v)
        ELSE
           LET gs0 = gsOut(p,seg, linear)
               gsAt_d = gsAtTime(p, seg, gs0, t, linear)
               adv = posVelWithinSeg(p, seg, t, linear, gsAt_d)
               altPair = interpolateAltVs(p,seg,t-time(p,seg),linear)
               sNew = mkAlt(adv.first, altPair.first)
               vNew = Velocity.mkVs(adv.second, altPair.secong)
           IN
               (sNew, vNew)
        ENDIF
   ENDIF
```

This key function relies on two subfunctions: `posVelWithinSeg` and `interpolateAltVs` described below. The function `Velocity.mkVs(v,z)` returns a velocity with the same horizontal information as `v`, but with vertical speed `z`.

6.8 The `posVelWithinSeg` Function

The function `posVelWithinSeg` determines if the horizontal motion is linear or a turn and then calculates the appropriate new 2D position:

```
posVelWithinSeg(p: Plan, seg:int, t:real, linear:bool, gsAt_d: real):
                                                   [Position, Velocity] =
      LET  np1 = point(p,seg)
           so = position(np1)
      IN
```

```
IF NOT linear AND inTrkChange(p,t) THEN
   LET ixPrevBOT = prevBOT(p, seg + 1)
       center = turnCenter(p, ixPrevBOT)
       dir = turnDir(p, ixPrevBOT)
       distFromSo = distFromPointToTime(p, seg, t, linear)
   IN
       turnByDist2D(so, center, dir, distFromSo, gsAt_d)
ELSE
   LET np2 = point(p, seg+1)
       vo = initialVelocity(np1,np2)
       distFromSo = distFromPointToTime(p, seg, t, linear)
   IN
       linearDist2D(so, trk(vo), distFromSo, gsAt_d)
ENDIF
```

6.9 The `initialVelocity` Function

The `initialVelocity` function calculates the initial velocity between two `NavPoints`

```
initialVelocity(s1: NavPoint, s2: NavPoint): Velocity =
   LET pp = position(s1)
       tt = time(s1)
       dt = time(s2) - tt
   IN
     IF dt = 0 THEN
        zero
     ELSIF dt > 0 THEN
        IF isLatLon(s1) THEN
            GreatCircle.velocity_initial(s1, s2, dt)
        ELSE
           (1/dt)*(s2 - s1)
        ENDIF
     ELSE
       IF isLatLon(s1) THEN
           GreatCircle.velocity_initial(s2, s1, -dt)
       ELSE
           (-1/dt)*(s1 - s2)
       ENDIF
     ENDIF
```

6.10 The `interpolateAltVs` Function

The `interpolateAltVs` function makes the vertical calculations of altitude and vertical speed:

```
interpolateAltVs(p: Plan, seg: int, dt: real, linear: bool):
                                                          [real,real] =
    LET tSeg = time(p,seg)
        vsAccel_d = IF ( NOT linear AND inVsChange(p, tSeg)) THEN
                      vsAccel(p, prevBVS(p, seg+1))
                    ELSE
                      0.0
                    ENDIF
        alt1 = alt(point(p,seg))
        vsInit = vsOut(p,seg,linear)
        newAlt = alt1 + vsInit*dt + 0.5 * vsAccel_d*dt*dt
        newVs = vsInit + vsAccel_d*dt
    IN
        (newAlt,newVs)
```

6.11 Consistent Plans, Revisited

Consistent plans must first be well-formed. Furthermore, all TCP begin and end locations must be consistent with the acceleration values in order for a plan to be usable. When the calculations are done using floating point arithmetic, the agreement will not be perfect. Therefore, approximate agreement within bounds is used. This section provides a formal description of plan consistency that was introduced in Section 3.3. To allow for floating point implementations, the functions in this section contain a tolerance parameter `distH_Epsilon` that specifies how accurate the calculated positions must be.

6.11.1 Track TCP Constraints

The function `isTurnConsistent` is applied to a `BOT` to test whether all the points in the turn (until the `EOT`) are the proper distance from the center of turn. Recall that the radius of the turn represents the surface distance from the center of the turn to the `BOT`, or the curved distance over the earth. The mathematical definition of `isTurnConsistent`, split over two functions, is:

```
isTurnConsistent(p:Plan, i:int, distH_Epsilon: double): bool =
   IF p.isBOT(i) THEN
      LET
        center = p.turnCenter(i)
        radius = p.getTcpData(i).turnRadius()
      IN
        isTurnConsistentRec(p,i,distH_Epsilon,radius,center)
    ELSE false
    ENDIF
```

```
isTurnConsistentRec(p:Plan, i:int, distH_Epsilon: double,
                    radius: double, center: Position): bool =
   LET
     dist = p.point(i).position().distanceH(center)
     rtn = abs(dist - radius) > distH_Epsilon
   IN
     IF i >= p.size() THEN false
       ELSIF p.isEOT(i) THEN rtn
       ELSE rtn AND
            isTurnConsistentRec(p,i+1,distH_Epsilon,radius,center)
     ENDIF
```

6.11.2 Ground Speed TCP Constraints

The function `isGsConsistent` is applied to a `BGS` to test whether the ground speed acceleration into the next `EGS` is properly defined. Its mathematical definition is:

```
isGsConsistent(p: Plan, ixBGS: int,  distEpsilon: double): bool =
     IF p.isBGS(ixBGS) THEN
         LET BGS = p.point(ixBGS)
             ixEGS = p.nextEGS(ixBGS)
             EGS = p.point(ixEGS)
             gsOutBGS = p.gsOut(ixBGS)
             dt = time(EGS) - time(BGS)
             a_BGS = p.gsAccel(ixBGS)
             ds = gsOutBGS*dt + 0.5*a_BGS*dt*dt
             distH = p.pathDistance(ixBGS,ixEGS)
             absDiff = abs(ds-distH)
         IN
             absDiff < distEpsilon
     ELSE
        true
     ENDIF
```

6.11.3 Vertical Speed TCP Constraints

The function `isVsConsistent` is applied to a `BVS` to test whether the ground speed acceleration into the next `EVS` is properly defined. Its mathematical definition is:

```
isVsConsistent(p: Plan, ixBVS: int, distEpsilon: double): bool =
    IF p.isBVS(ixBVS) THEN
       LET
          ixEVS = p.nextEVS(ixBVS)
          VSCBegin = p.point(ixBVS)
          VSCEnd = p.point(ixEVS)
          a = p.vsAccel(ixBVS)
```

```
            dt = time(VSCEnd) - time(VSCBegin)
            ds = p.vsOut(ixBVS)*dt + 0.5*a*dt*dt
            deltaAlts = VSCEnd.alt - VSCBegin.alt()
            absDiff = abs(ds-deltaAlts)
         IN
            absDiff <= distEpsilon
        ELSE
           true
        ENDIF
```

6.12 Velocity-Continuous Plans, Revisited

We define the function `isVelocityContinuous`:

```
   isVelocityContinuous(p: Plan, i:int): bool =
            trkIn(p,i) = trkOut(p,i)
        AND gsIn(p,i)  = gsOut(p,i)
        AND vsIn(p,i)  = vsOut(p,i)
```

which checks for continuity at index i. This function depends upon the following subfunctions which have not been defined yet:

```
  trkInTurn(p: Plan, pos:Position, center: Position, dir:int): real =
     IF isLatLon(pos) THEN
        LET final_course = GreatCircle.final_course(center,pos) IN
           final_course + dir*PI/2
     ELSE
        LET trkFromCenter = track(center, pos)
            nTrk = trkFromCenter
        IN
            nTrk + dir*PI/2
     ENDIF

  track(p1:Vect3, p2:Vect3): real =
      atan2(p2.x-p1.x, p2.y-p1.y)

  trkFinal(p: Plan, seg: int, linear: bool): real =
     IF seg < 0 OR seg >= size(p) - 1 THEN -1
     ELSE
       IF inTrkChange(p, time(p,seg)) AND NOT linear THEN
          LET ixBOT = prevBOT(p,seg+1)
              dir = turnDir(p,ixBOT)
              center = turnCenter(p, ixBOT)
          IN
              trkInTurn(p, point(p,seg+1), center, dir)
       ELSE
```

```
        IF isLatLon(p) THEN
           GreatCircle.final_course(point(p, seg), point(p,seg+1))
        ELSE
           trk(finalVelocity(point(p, seg), point(p, seg+1)))
        ENDIF
     ENDIF
   ENDIF

trkOut(p: Plan, seg:int, linear: bool): real =
   IF (seg = size(p) - 1) THEN
      trkFinal(p, seg-1, linear)
   ELSE
      IF inTrkChange(p, time(p,seg)) AND NOT linear THEN
         LET ixBOT = prevBOT(p, seg + 1)
             center = turnCenter(p, ixBOT)
             dir = turnDir(p, ixBOT)
         IN
             trkInTurn(p, point(p, seg), center, dir)
      ELSE
         IF isLatLon(p) THEN
            GreatCircle.initial_course(point(p,seg),point(p,seg+1))
         ELSE
            trk(initialVelocity(point(p, seg), point(p, seg+1))
        ENDIF
      ENDIF
  ENDIF

trkIn(p: Plan, seg:int): real =
     trkFinal(p,seg-1,false);

trkOut(p: Plan, seg:int): real =
     trkOut(p,seg,false);

vsFinal(p: Plan, i: int, linear: bool): real =
   IF i < 0 OR i > size(p) - 1 THEN -1
   ELSE
      LET
         dist = alt(point(p,i+1)) - alt(point(p,i))
         dt = time(p, i+1) - time(p, i)
         a = IF inVsChange(p, time(p,i)) AND NOT linear THEN
                 vsAccel(p, prevBVS(p, i + 1))
             ELSE
                0.0
             ENDIF
```

```
      IN
          dist / dt + 0.5 * a * dt
   ENDIF

vsFinal(p: Plan, i: int): real = vsFinal(p,i,false)

vsOut(p: Plan, i: int, linear: bool): real =
   IF i < 0 OR i >= size(p) - 1 THEN -1
   ELSIF i = size(p) - 1 THEN
       vsFinal(p, i - 1)
   ELSE
      LET
         dist = alt(point(p,i+1)) - alt(point(p,i))
         dt = time(p, i+1) - time(p, i)
         a = IF inVsChange(p, time(p, i)) AND NOT linear THEN
                vsAccel(p, prevBVS(p,i + 1))
             ELSE
                0
             ENDIF
      IN
        dist / dt - 0.5 * a * dt
   ENDIF


vsIn(p: Plan, i: int): real =
   vsFinal(p,i-1,false)

vsOut(p: Plan, i: int): real =
   vsOut(p,i,false)
```

7 TCP Generation

The plan data structure efficiently stores the information needed to retrieve the position and velocity of a trajectory at any time point t. We have described this functionality in detail. However, we have not yet explained how one creates a well-defined consistent trajectory. A linear trajectory is easy to construct; however, a consistent and velocity-continuous kinematic plan with multiple TCP regions (possibly overlapping) can be challenging. We have developed two major algorithms for doing this:

- `TrajGen.makeKinematicPlan`
- `Distplan.makePlan`

We note that these algorithms are not used in the operational semantics of the EUTL language. These algorithms produce EUTL trajectories as an output. As

these algorithms are outside the scope of this paper, they are only broadly described here.

7.1 `TrajGen.makeKinematicPlan`

The `makeKinematicPlan` algorithm accepts a linear plan as input and creates a consistent and velocity-continuous plan from it. This trajectory generator is a low-fidelity generator and does not use aircraft performance data or wind data as input. This is a multi-pass algorithm that first marks significant vertical speed change points. It then generates circular turns between non-collinear segments. It next calculates areas of horizontal acceleration between segments with different velocities. Because changing ground speed can also alter the vertical profile, it then corrects the vertical profile to have a similar behavior to the original based on the previously marked points. It finally calculates vertical acceleration zones. More details on how this is achieved can be found in [9].

Consider the horizontal view of a linear plan show in Figure 8.

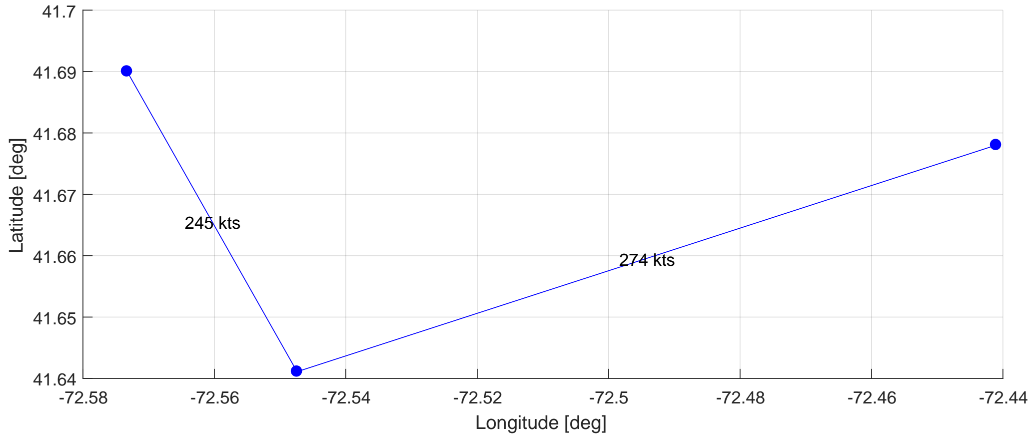


Figure 8: Horizontal View of Linear Plan.

Point	Latitude (°)	Longitude (°)	Altitude (ft)	Time (s)
1	41.690007	-72.573280	5000.0	36130.0
2	41.641105	-72.547418	6000.0	36176.3
3	41.678012	-72.441028	6000.0	36245.4

Table 5: Linear Plan Input

This linear plan was generated by the input from Table 5. In this case the generation used default acceleration values of 4 m/s horizontal and 2 m/s vertical. The bank angle used to calculate turns was 25 degrees.

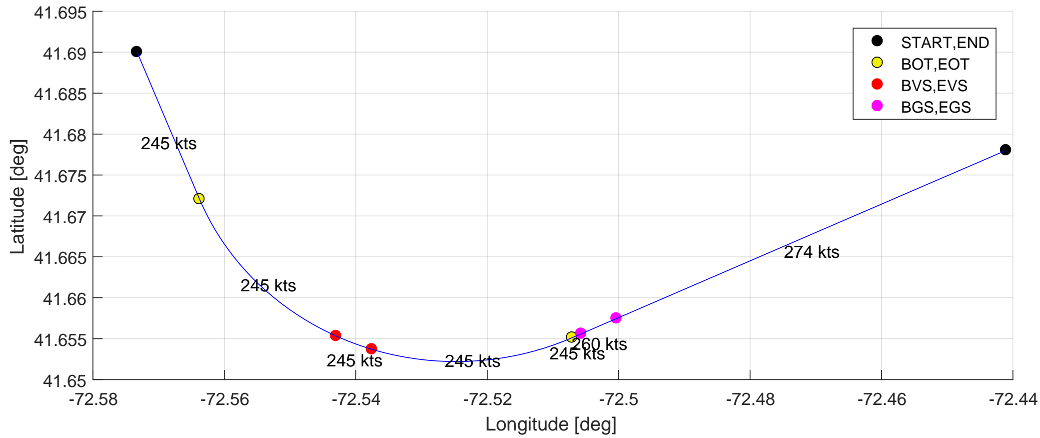


Figure 9: Horizontal View of Generated Plan.

The horizontal profile of the generated kinematic plan is show in Figure 9. The yellow waypoints are the `BOT` and `EOT` TCPs. The magenta waypoints are the `BGS` and `EGS` TCPs. The red waypoints are the `BVS` and `EVS` TCPs. The vertex point of the turn has been replaced with a turn section. In this figure, the ground speed change that occurs at the vertex in the linear plan has been deferred until after the completion of the turn. This makes the maneuver a constant bank-angle turn, which may be easier to fly. Alternatively, the generator can make the ground speed change occur at the mid point of the turn. Note that there are red vertical TCPs within the turn and that the speed values displayed are average speeds in each segment. The vertical profile of this kinematic plan is show in Figure 10.

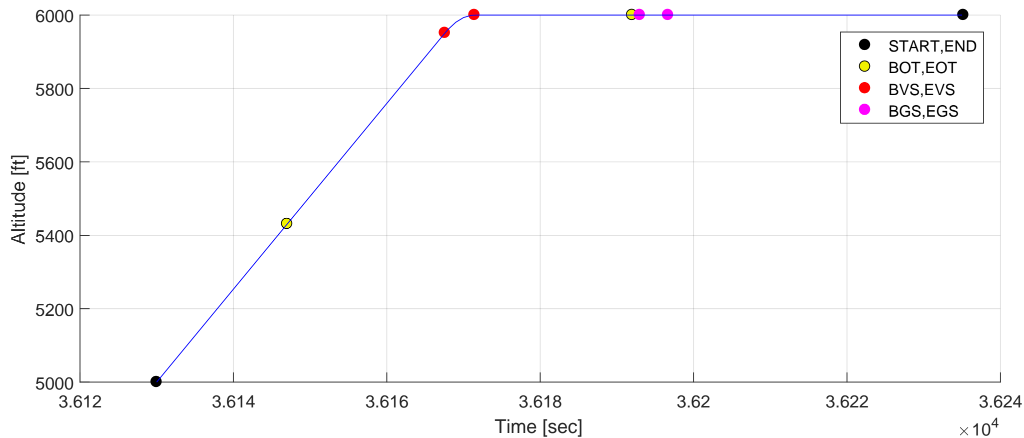


Figure 10: Vertical View of Generated Plan.

7.2 `DistPlan.makePlan`

The `TrajGen` algorithm is a means of transforming a linear plan into a kinematic plan. In it, speeds are inferred from point times and acceleration values are user-specified. The `DistPlan` utility synthesizes a kinematic plan from a 2D profile (a distance-indexed speed profile and a distance-indexed altitude profile) using a multi-pass process. With the `DistPlan` utility, speeds and distances are specified and speed changes are assumed to be gradual; acceleration values and times are inferred. It is intended to facilitate incorporating non-constant wind data and higher fidelity aircraft dynamic models, allowing these to be abstracted into kinematic plans, such as presented in Section 8.

More formally, the function `DistPlan.makePlan` receives three inputs and produces a consistent and velocity-continuous, kinematic plan from them as output. The three inputs are:

- a list of 2D waypoints with radius information (i.e., a route);
- ground speeds as a function of path distance; and
- altitudes as a function of path distance.

The speed profile indicates the boundaries of acceleration zones.

The route is transformed into a kinematic plan that describes all turns and horizontal segments using the standard turn generation algorithm and a default ground speed. This plan (Plan 1) defines the turns and distances for the final kinematic plan. This allows us to directly map speed and altitude data to it using path distance as a reference.

Each point in the ground speed profile indicates the desired speed at that path distance in the route, and it is assumed that between points there is a constant (possibly zero) acceleration. It is required that the first point, at distance 0, is always assigned an initial speed.

The ground speed profile may be divided into 3 sections representing take-off/climb, cruise, and descent/landing. If this is so, then the various sections are combined into a single unified profile before it is processed. If the distances are consistent with the total distance defined by Plan 1 and the sections do not overlap, they are simply concatenated into a single list, indexed from the start of the plan.[8] If there is some overlap, then the cruise section is truncated or expanded to allow the climb and descent portions to fit within Plan 1 exactly.

Vertical profile points are considered "fixed altitude" points, with vertical speeds inferred between them. For greater accuracy, additional points may be included to better describe an actual climb or descent profile. Similar to the speed profile, it is assumed that the first point will have a specified altitude.

If the vertical profile is provided in three parts, it is processed in a similar way to the three-part ground speed profile, though altitudes are indexed against path distance. Any major mismatch in altitude between sections is registered as an error.

[8] For ease of computation, the climb and cruise sections are indexed from the start of the trajectory, while the descent portion is initially calculated backwards from the end of the trajectory.

During processing, ground speeds are not calculated using the normal trajectory generation algorithm. Instead, the unified ground speed profile is applied to Plan 1. At each distance a new, marked point is added (if necessary), and the speed of each section between marked points is modified to match the profile's speeds. This results in a Plan 2, with regions of constant speed.

Plan 2 is scanned, searching for speed-marked points. At each of these points, the previous segment speed is compared to the next segment speed. If there is a discontinuity, then the segments between the previous two speed-marked points is interpreted as a non-zero acceleration zone and its start and end are marked as `BGS` and/or `EGS`, and an appropriate acceleration value is calculated. If there is no change, then the region simply retains the previously defined constant ground speed. When complete, each segment will have been assigned a constant ground speed acceleration, possibly zero. This results in a new Plan 3.

Next, the vertical profile is applied. All points in the vertical profile are added to Plan 3 as "fixed altitude" points. This results in Plan 4. Finally the normal vertical TCP generation algorithm is applied to Plan 4, resulting in the output of a final consistent kinematic plan.

8 Generation of Medium Fidelity Trajectories

In this section we describe the use of the EUTL within a low- to medium-fidelity trajectory generator. The purpose of this trajectory generator is to generate reasonable, consistent and velocity-continuous trajectories for commercial aircraft given the flight plan information, the filed cruise true airspeed (TAS) and cruise altitude, and an aircraft performance model. The trajectory generator uses the `DistPlan` utility described in Section 7.2 . These trajectories have been used within the TBO Toolkit for Integrated Air/Ground Research (TBO-TIGAR), which is a tool intended for early-stage TBO concept exploration. We illustrate the trajectory generator using the following flight:

```
Call Sign: NASA001
Aircraft Type: B712 (Boeing 717)
Flight Plan: KBWI.TERPZ.WONCE.JIMME.ADDEK.HAFNR.GVE.LYH.KELLS
             .MAYOS.MAJIC.SUDSY.GIZMO.AMOBE.INNOR.JATAB.KCLT
Cruise TAS: 455 kts
Cruise Altitude: 34000 ft (FL340)
```

The flight plan defines the origin airport (KBWI), the destination airport (KCLT), and the waypoints that must be sequenced between them. The cruise altitude and cruise speed provide the target states for the cruise portion of the flight but the trajectory generator still needs to compute the full vertical speed and altitude profiles for this flight. To do so, we make assumptions about how the aircraft will execute the vertical profile, primarily in terms of indicated airspeeds (IAS) within certain altitude regions. Using a performance model for the B712 aircraft type, a standard atmosphere table, the nominal range for this flight, and the speed

constraints below 10,000 feet, we end up with the climb, cruise, and descent speed profile in Table 6, in terms of calibrated airspeed (CAS)[9] or Mach number.

Table 6: Calibrated/Mach Speed Profile Versus Altitude.

Phase	Starting Altitude (ft)	Ending Altitude (ft)	Target CAS (kts/Mach)
Climb	0.0	10000.0	250.0
Climb	10000.0	30160.6	270.0
Climb	30160.6	34000.0	0.72
Cruise	34000.0	34000.0	0.786
Descent	34000.0	33221.1	0.74
Descent	33221.1	10000.0	260.0
Descent	10000.0	0.0	250.0

The speed profile shown in Table 6 provides the calibrated airspeed targets within an altitude region for this flight. These speeds are useful for describing the profile from the aircraft operational point-of-view but are not sufficient for generating the full vertical profile for this flight. Even though the calibrated speed is being held constant within these regions, the true airspeed is changing with altitude due to the changing atmosphere (pressure, density and temperature). Also, as the aircraft climbs or descends, the climb or descent rate is also variable with altitude. Additionally, the effects of wind will impact the ground speed (GS) of this trajectory. Thus, the next step in this trajectory generation is to compute the vertical speed and altitude profile that takes all of these effects into account.

The calibrated airspeed profile is converted into a ground speed and altitude profile as a function of range. We begin with the reference *linear* plan generated from the flight plan waypoints, as seen in Figure 11[10]. This reference linear plan is used as a way to query a given wind field for the magnitude and direction of the wind vector at any given location and altitude along this flight plan. The trajectory generator computes climb ground speed and altitude profiles in 3000 foot altitude increments from the departure airport until reaching the cruise airspeed and cruise altitude using the following algorithm:

1. determine the target calibrated airspeed based on the current altitude;
2. determine the target true airspeed at the current range given the target calibrated airspeed and the current range altitude;
3. determine the ground speed at the current range given the true airspeed and the wind at the current range;
4. determine the climb rate for this aircraft at the current altitude from the performance data;

[9]Calibrated airspeed is indicated airspeed that has been corrected for instrumentation error.

[10]The point "$IF" is a virtual point inserted by the trajectory generator that lines up the trajectory with the final approach segment.

5. compute the target true airspeed and target altitude at the next altitude increment;

6. use the current vertical rate to determine the time required to execute the step change in altitude;

7. use the time estimate to determine the range required to execute the step change in altitude;

8. update the current range, current altitude, and current true airspeed based on the step change; and

9. repeat above steps until reaching cruise altitude and cruise airspeed.

Note that the change in indicated airspeed executed at 10,000 feet is done via a short level segment. Also, after crossover altitude, the target speed becomes a constant indicated Mach instead of a constant indicated airspeed.

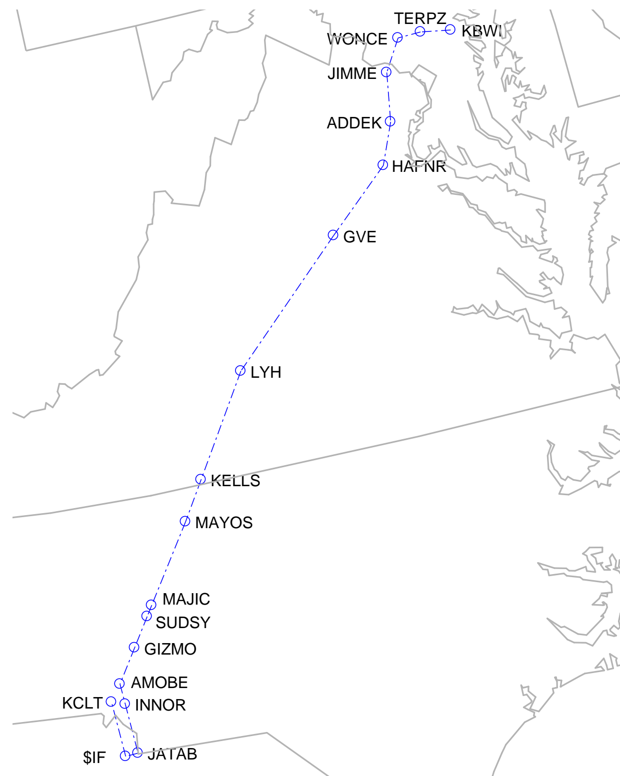


Figure 11: Reference Linear Plan Generated from the Flightplan.

The descent profile is computed using the same algorithm as the climb profile generator with two exceptions. First, the descent rate is broken up into three vertical rate regions; vertical rates above 30,000 feet; vertical rates above 10,000 feet; and vertical rates below 10,000 feet. The vertical rates within these regions target a

geometric descent path of roughly 3 degrees, 2.5 degrees, and 2 degrees, respectively. Second, the descent profile is generated in reverse from the destination airport back to cruise altitude in order to properly account for the wind as a function of range distance from the destination[11]. The computed speed and altitude profiles versus range for the example flight can be seen in Figure 12. When generating a trajectory in the presence of winds, the trajectory generator will also generate a ground speed profile from the end of the climb profile to the start of the descent profile, in increments of five nautical miles of range, to take into account the winds along the en route portion of the trajectory.

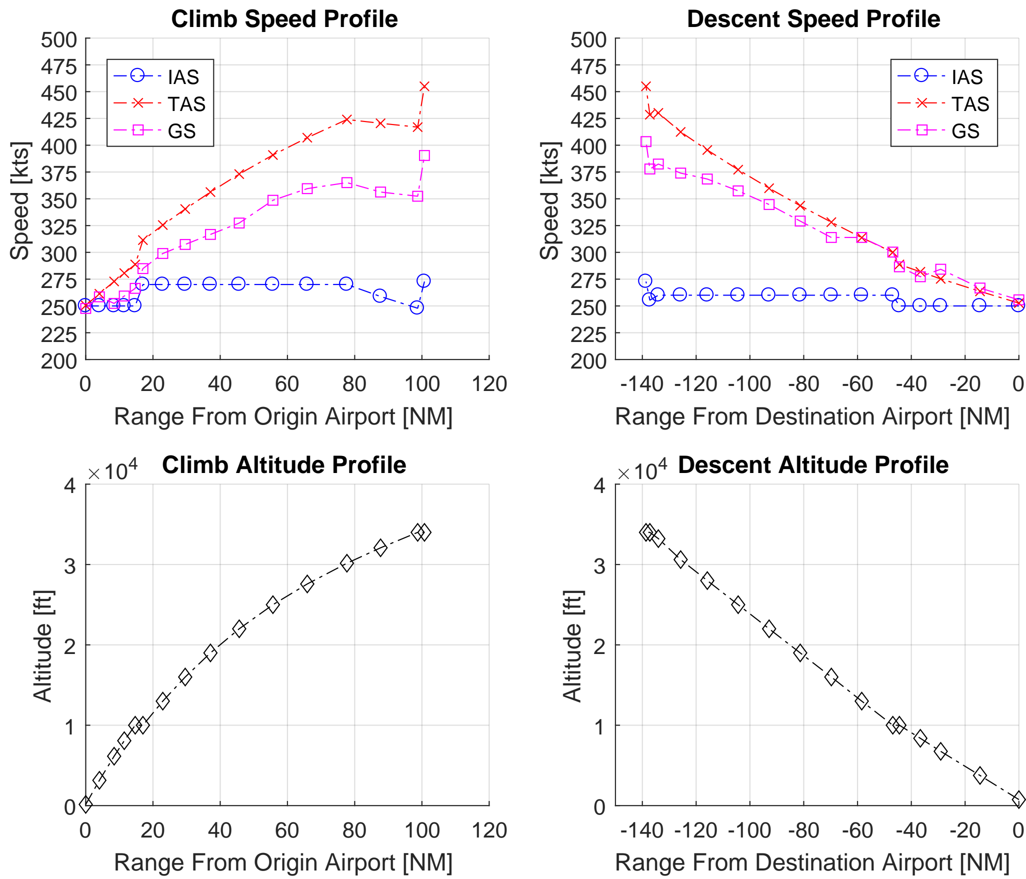


Figure 12: Speed and Altitude Profiles as a Function of Range.

The final step in trajectory generation is to use the DistPlan method described in Section 7.2 to generate a trajectory in the EUTL Plan format. A 2D Route is generated using the linear flight plan and annotated with the turn radii at each

[11]The top of descent location or range is not known until the full descent profile has been calculated, making it difficult to determine the appropriate wind information to use within the profile; this motivates the reverse computation of the descent profile.

waypoint, using the speed and altitude estimates at the range for those waypoints; turn radii are calculated using a default assumed bank angle (25 degrees is used in this example). The ground speed and altitude profiles as a function of range, along with this 2D Route, are used to compute a kinematic plan. The final kinematic trajectory for the example flight can be seen in Figures 13 and 14.

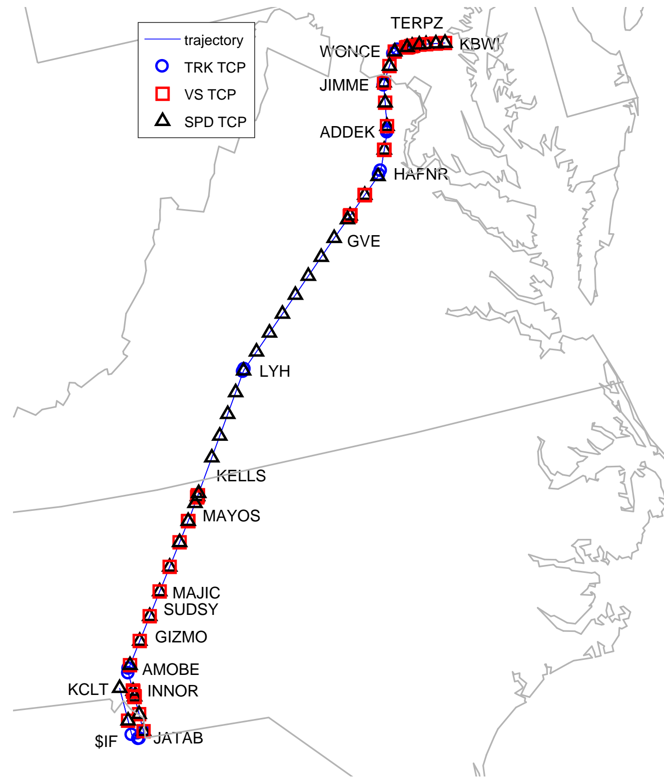


Figure 13: Kinematic Trajectory Showing the TCP Points.

The resulting altitude, vertical speed, ground speed, and track angle profiles for this example kinematic trajectory can be seen in Figures 15-18. The headwind experienced along this plan can also be seen in Figure 19. The textual version of the first 17 points of the generated plan is shown in Figure 20 with time in seconds, latitude and longitude in degrees, and altitude in feet. Turn radii are in nautical miles and horizontal and vertical accelerations are in meters per second. The total plan has 95 points.

In future work we hope to document the generation algorithm in more detail. We note that it is also possible to generate lower fidelity trajectories into the EUTL language. The trajectory shown in Figure 21 was created by a simpler trajectory generator for the same route. This trajectory has half the number of points in its Plan.

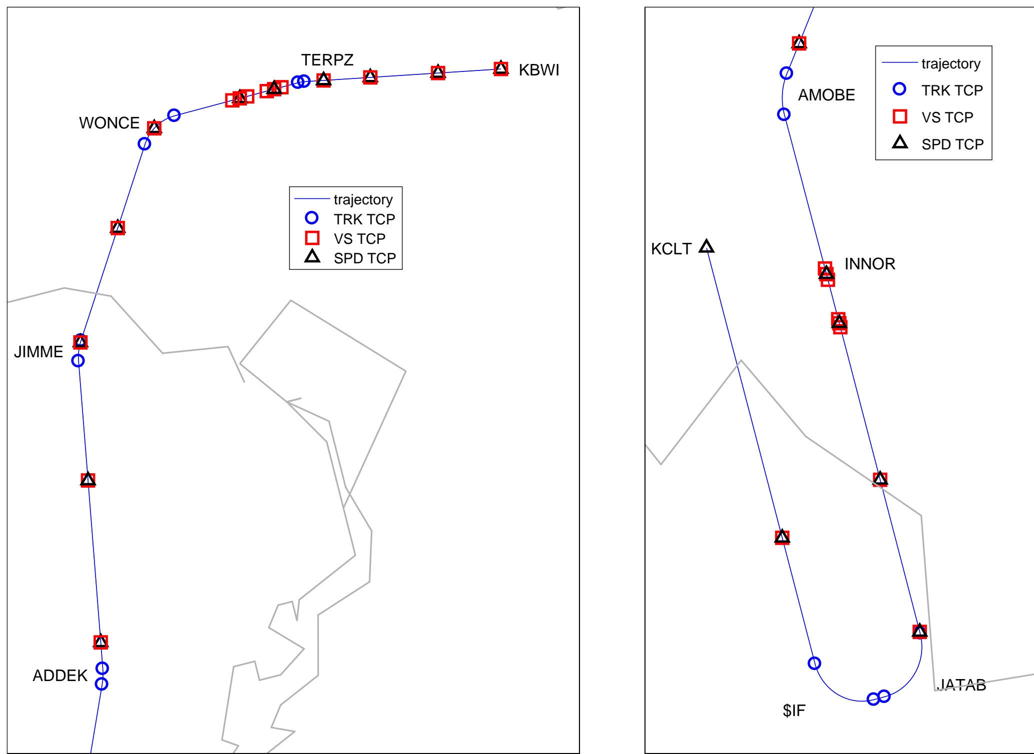


Figure 14: Kinematic Trajectory Detail for Departure (Left) and Arrival (Right).

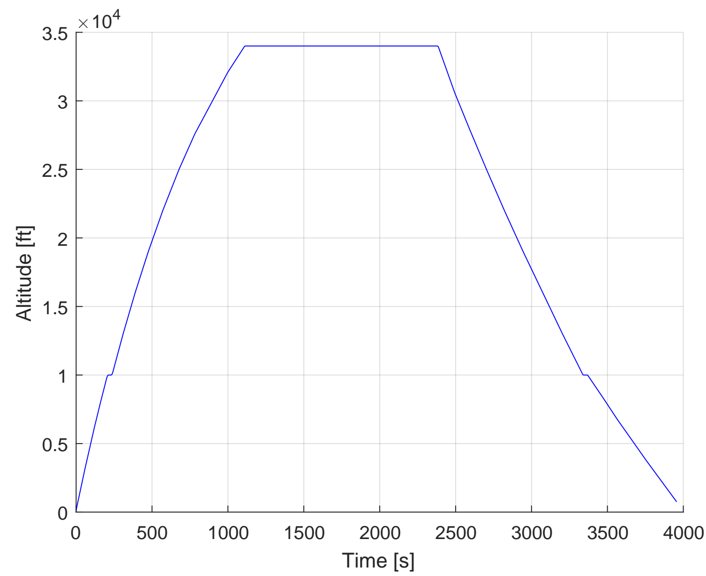


Figure 15: Kinematic Trajectory Altitude Profile with Time.

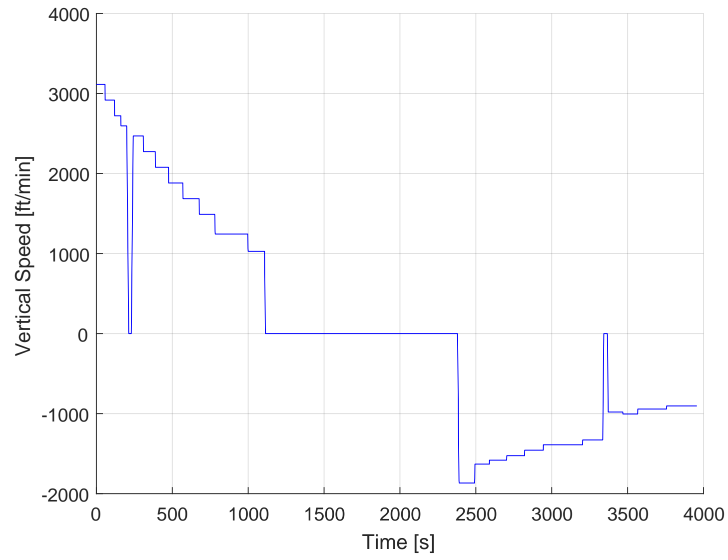


Figure 16: Kinematic Trajectory Vertical Speed Profile with Time.

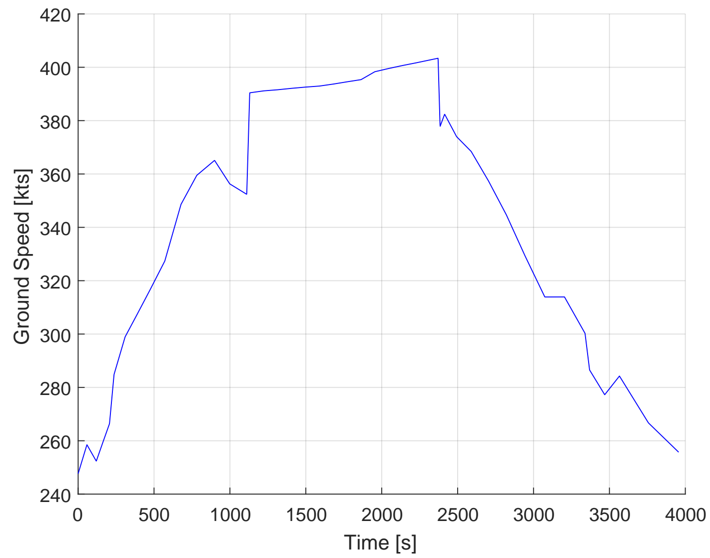


Figure 17: Kinematic Trajectory Ground Speed Profile with Time (in the Presence of Wind).

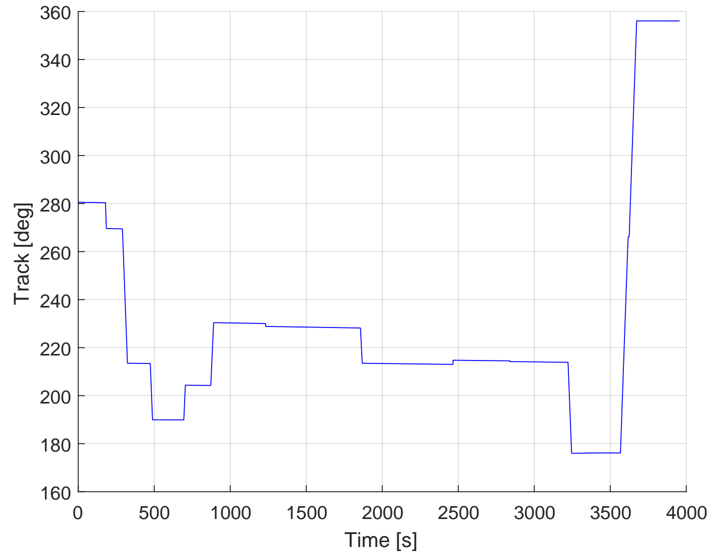


Figure 18: Kinematic Trajectory Track Angle Profile with Time.

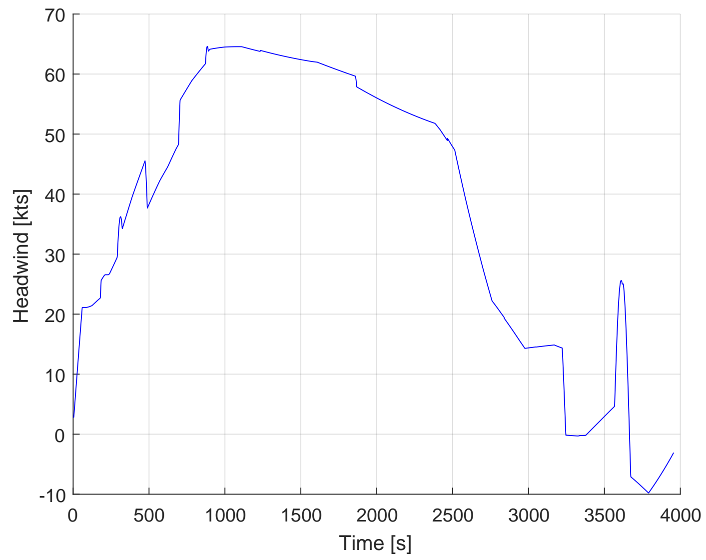


Figure 19: Headwind Along the Plan's Altitude versus Plan Range.

```
  TIME    LATITUDE    LONGITUDE  ALTITUDE    NAME     TCP_DATA
------   --------------------------------   ----     --------------
  0.00   (39.175361, -76.668333,   146.00)  KBWI     (BGS 0.097)     ;
 57.81   (39.187720, -76.754260,  3146.00)           (EGSBGS -0.051);
119.51   (39.200960, -76.846841,  6146.00)           (EGSBGS 0.084)  ;
161.99   (39.210048, -76.910704,  8073.00)           (EGSBGS 0.082)  ;
179.09   (39.213764, -76.936894,  8812.64)  TERPZ    (BOT -2.152)    ;
181.86   (39.214214, -76.941201,  8932.67)                           ;
184.63   (39.214349, -76.945543,  9052.49)           (EOT)           ;
199.94   (39.214216, -76.969703,  9714.78)           (BVS -1.000)    ;
206.53   (39.214157, -76.980177,  9928.69)           (EGSBGS 0.317)  ;
213.13   (39.214097, -76.990751, 10000.00)           (EVS)           ;
230.26   (39.213931, -77.018979, 10000.00)           (BVS 1.000)     ;
236.53   (39.213867, -77.029584, 10064.55)           (EGSBGS 0.099)  ;
242.81   (39.213801, -77.040284, 10258.22)           (EVS)           ;
290.62   (39.213258, -77.123337, 12226.44)  WONCE    (BOT -2.790)    ;
307.14   (39.207625, -77.151425, 12906.38)                           ;
309.41   (39.206032, -77.154924, 13000.00)           (EGSBGS 0.055)  ;
323.51   (39.192436, -77.172812, 13534.26)           (EOT)           ;
388.58   (39.116075, -77.237932, 16000.00)           (EGSBGS 0.056)  ;
...
```

Figure 20: Textual Version of Generated Plan.

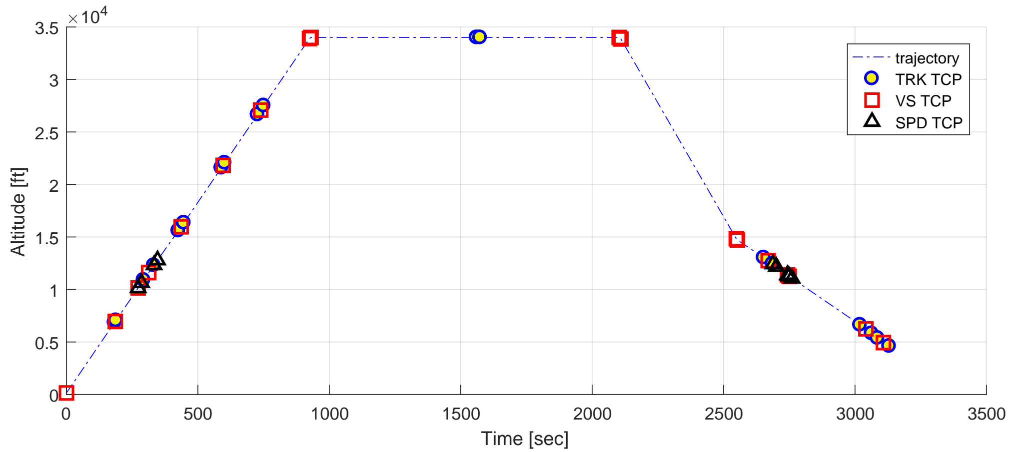


Figure 21: Vertical Profile of Low Fidelity Trajectory (Altitude versus Time).

9 Concluding Remarks

This paper presents a new language for specifying trajectories called the Efficient, Universal Trajectory Language (EUTL). This language is defined for ATM applications where trajectories are communicated between air and ground systems. Therefore, a high priority was placed on efficiently communicating unambiguous trajectory descriptions. The language defines a trajectory as a sequence of 3D positions and times where there is a constant velocity or constant acceleration between each point. The semantics of the language are given in mathematical detail so that position and velocity are precisely defined for all time points within a plan's starting and ending time. The language was also defined in a manner that does not depend upon any particular control system or aircraft dynamics model. For this reason we consider it a universal trajectory language.

References

1. Russell A. Paielli. Trajectory specification for high-capacity air traffic control. *Journal Of Aerospace Computing, Information, And Communication*, 2, Sept 2005.

2. Intent project: The transition towards global air and ground collaboration in traffic separation assurance. www.intentproject.org.

3. R.C.J. Ruigrok and M.S.V. Valenti Clari. The impact of aircraft intent information and traffic separation assurance responsibility on en-route airspace capacity. In *5th FAA/EUROCONTROL ATM R&D Seminar*, Jun 2003.

4. Robert A Vivona, Karen T Cate, and Steven M Green. Abstraction techniques for capturing and comparing trajectory predictor capabilities and requirements. In *AIAA Guidance, Navigation and Control Conference, Honolulu, HI*, 2008.

5. Sip Swierstra and Steven Green. Common trajectory prediction capability for decision support tools. In *5th USA/Eurocontrol ATM R&D Seminar, Budapest, Hungary*, 2003.

6. Stéphane Mondoloni and Daniel Kirk. Proposed trajectory prediction and exchange information items for flight information exchange model (fixm). Technical report, MITRE, 2012.

7. Bill Gill and Bob Maddock. Prediction of optimal 4D trajectories in the presence of time and altitude constraints. Technical Report DOC 97-70-09, PHARE, European Organization For the Safety of Air Navigation, Feb 1997.

8. Ben Musialek, Carmen F. Munafo, Hollis Ryan, and Mike Paglione. Literature survey of trajectory predictor technology. Technical Report DOT/FAA/TC-TN11/1, Federal Aviation Administration William J. Hughes Technical Center, 2010.

9. George E. Hagen and Ricky W. Butler. Towards a formal semantics of flight plans and trajectories. Technical Memorandum NASA/TM2014-218662, NASA, Dec 2014.

10. S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *Proc. 11th Int. Conf. on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, June 1992.

11. N. Shankar, S. Owre, and J. M. Rushby. *PVS Tutorial*. Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993. Also appears in Tutorial Notes, *Formal Methods Europe '93: Industrial-Strength Formal Methods*, pages 357–406, Odense, Denmark, April 1993.

12. United States Air Force Chief Scientist. Technology Horizons: A vision for Air Force science and technology 2010-30. Technical Report AF/ST-TR-10-01-PR, http://www.af.mil/information/technologyhorizons.asp, May 2010.

13. George E. Hagen, Ricky W. Butler, and Jeffrey M. Maddalon. The Stratway program for strategic conflict resolution: Users guide. Technical Memorandum NASA/TM2016-219196, NASA, May 2016.

14. Nelson M. Guerreiro, Ricky W. Butler, George E. Hagen, Jeffrey M. Maddalon, and Timothy A. Lewis. Parametric analysis of surveillance quality and level and quality of intent information and their impact on conflict detection performance. Technical Memorandum NASA/TM-2016-219177, NASA, March 2016.

15. Nelson M. Guerreiro, Ricky W. Butler, Jeffrey M. Maddalon, George E. Hagen, and Timothy A. Lewis. Conflict detection performance analysis for function allocation using time-shifted recorded traffic data. In *15th AIAA Aviation Technology, Integration, and Operations Conference Dallas, TX*, June 22-26 2015.

16. ICAO. Global air traffic management operational concept. Technical Report 9854 AN/458, International Civil Aviation Organization, 2005.

17. Samet Ayhan and Hanan Samet. Aircraft trajectory prediction made easy with predictive analytics. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, August 13-17, 2016.

18. Daniel Delahaye, Stéphane Puechmorel, Panagiotis Tsiotras, and Eric Feron. Mathematical models for aircraft trajectory design: A survey. *Lecture notes in Electrical Engineering*, 10.1007/978-4-431-54475-3:205–247, 2014. hal-00913243.

19. Guillermo Frontera, Juan A. Besada, Ana M. Bernardos, Enrique Casado, and Javier López-Leonés. Formal intent-based trajectory description languages. *IEEE Transactions On Intelligent Transportation Systems*, 15(4), Aug 2014.

20. Terence S. Abbott. An overview of a trajectory-based solution for en route and terminal area self-spacing: Seventh revision. Technical Report NASA/CR2015-218794, Stinger Ghaffarian Technologies, Hampton, Virginia, August 2015.

21. Randy Walter. Flight management systems. In Cary R . Spitzer, editor, *Digital Avionics Handbook, Second Edition*. CRC Press, 2000.

22. RTCA SC-186. Minimum aviation system performance standards for automatic dependent surveillance broadcast (ADS-B), 2002.

23. J. Lopez-Leones, M. A. Vilaplana, E. Gallo, F. A. Navarro, and C. Querejeta. The aircraft intent description language: A key enabler for air-ground synchronization in trajectory-based operations. In *26th IEEE/AIAA Digit. Avionics Systems Conference*, 2007.

24. RTCA Inc. Safety and performance requirements standard for baseline 2 ATS data communications (baseline 2 spr standard). Technical Report Volume 1, DO-350A, Washington, DC, 2016.

25. RTCA Inc. Interoperability requirements standard for baseline 2 ATS data communications (baseline 2 interop standard). Technical Report Volume 1 and 2, DO-351A, Washington, DC, 2016.

26. Aaron Dutle, César Muñoz, Anthony Narkawicz, and Ricky Butler. Software validation via model animation. *Lecture Notes in Computer Science*, 9254:92–108, 2015.

27. Ed Williams. Aviation formulary. http://www.edwilliams.org/avform.htm. Accessed, May 2, 2017.

28. United States Department of Defense. World geodetic system 1984: Its definition and relationships with local geodetic systems, 2014. NGA.STND.0036_1.0.0_-WGS84.

Appendix A

Support Functions

Functions that calculate position and velocity for a specified time are presented in Section 6. These functions provide the data structure with a tremendous amount of utility and are essential to defining the *semantics* of the trajectory encoded in the plan. There are many basic utility functions that were presented but not fully specified.

A.1 TCP Functions

The methods of Table 2 check if a point is of a particular TCP type.

```
isBOT(p: Plan, i: int) : bool =
   data(p,i).tcp_trk = BOT OR data(p,i).tcp_trk = EOTBOT

isEOT(p: Plan, i: int) : bool =
   data(p,i).tcp_trk = EOT OR data(p,i).tcp_trk = EOTBOT

isGsTCP(p: Plan, i: int) : bool =
   data(p,i).tcp_gs /= NONE

isBGS(p: Plan, i: int) : bool =
   data(p,i).tcp_gs = BGS OR data(p,i).tcp_gs = EGSBGS

isEGS(p: Plan, i: int) : bool =
   data(p,i).tcp_gs = EGS OR data(p,i).tcp_gs = EGSBGS

isVsTCP(p: Plan, i: int) : bool = data(p,i).tcp_vs /= NONE

isBVS(p: Plan, i: int) : bool =
   data(p,i).tcp_vs = BVS OR data(p,i).tcp_vs = EVSBVS

isEVS(p: Plan, i: int) : bool =
   data(p,i).tcp_vs = EVS OR data(p,i).tcp_vs = EVSBVS
```

The methods of Table 3 search for previous and subsequent occurrences of TCP types:

```
prevSearch(p: Plan, property:[Plan,int -> bool], i: int): int =
   IF (i < 0) THEN -1
   ELSE IF (p.point(i).property) THEN i
   ELSE prevSearch(p,property, i-1)
   ENDIF
```

```
prevBOT = prevSearch(p,isBOT,i-1)
prevBGS = prevSearch(p,isBGS,i-1)
prevBVS = prevSearch(p,isBVS,i-1)
```

A.2 The getSegment Function

Informally, `getSegment` is a function that, for a given plan p and time t, returns an integer value in the range $[-1, \texttt{size}(p) - 1]$. It returns the value -1 if the time falls outside p's bounds, or the index in p such that $t_i \leq t < t_{i+1}$. There are many possible implementations for this function, e.g., linear search or binary search. Rather than present a particular search method we specify the output formally. The necessary conditions for `getSegment(t):int` for a given plan p are:

$$\texttt{getSegment}(p,t) < 0 \iff t < \texttt{time}(p,0) \vee t > \texttt{time}(p,\texttt{size}(t)-1)$$
$$\texttt{getSegment}(p,t) = i \iff \texttt{time}(i) \leq t \wedge t < \texttt{time}(i+1)$$

A.3 In Acceleration Zone Functions

The functions `inTrkChange`, `inGsChange`, and `inVsChange` are true if, for the given plan p and time t within p, the segment containing time t is in an acceleration zone of that type. This can be accomplished through the use of the `prevBOT` function and a similar `nextEOT` function (and likewise for horizontal speed and vertical speed accelerations). The necessary conditions for `inTrkChange(p,t):bool` for a given plan p are:

$$\begin{aligned}\texttt{inTrkChange}(p,t) \iff & \exists i \exists j \forall k : (i \geq 0) \wedge (j > i) \wedge (i < k) \wedge (k < j) \wedge \\ & \texttt{isBOT}(p,i) \wedge \texttt{isEOT}(p,j) \wedge \neg\texttt{isBOT}(p,k) \wedge \\ & \texttt{getSegment}(p,t) \geq i \wedge \texttt{getSegment}(p,t) < j\end{aligned}$$

Similarly for `inGsChange(p,t):bool` and `inVsChange(p,t):bool`:

$$\begin{aligned}\texttt{inGsChange}(p,t) \iff & \exists i \exists j \forall k : (i \geq 0) \wedge (j > i) \wedge (i < k) \wedge (k < j) \wedge \\ & \texttt{isBGS}(p,i) \wedge \texttt{isEGS}(p,j) \wedge \neg\texttt{isBGS}(p,k) \wedge \\ & \texttt{getSegment}(p,t) \geq i \wedge \texttt{getSegment}(p,t) < j\end{aligned}$$

$$\begin{aligned}\texttt{inVsChange}(p,t) \iff & \exists i \exists j \forall k : (i \geq 0) \wedge (j > i) \wedge (i < k) \wedge (k < j) \wedge \\ & \texttt{isBVS}(p,i) \wedge \texttt{isEVS}(p,j) \wedge \neg\texttt{isBVS}(p,k) \wedge \\ & \texttt{getSegment}(p,t) \geq i \wedge \texttt{getSegment}(p,t) < j\end{aligned}$$

Appendix B

Great Circle Functions

The formulae in this section are based on relatively standard developments in spherical trigonometry. Many are based on an excellent summary of key results especially relevant to navigation problems presented by Williams in [27].

B.1 `angular_distance`

The `angular_distance` function computes the great circle distance between the two points in terms of the angle at the center of the sphere. The calculation applies to any sphere, not just a spherical Earth. This implementation uses the haversine formula.

```
angular_distance(lat1: real, lon1: real, lat2: real, lon2: real):
                                                          nnreal =
   asin(sqrt(sq(sin((lat1 - lat2) / 2))
     + cos(lat1)*cos(lat2)* sq(sin((lon1-lon2) / 2)))) * 2.0
```

Using the `LatLonAlt` data structure we have:

```
angular_distance(p1: LatLonAlt, p2: LatLonAlt): nnreal =
   angular_distance(lat(p1), lon(p1), lat(p2), lon(p2))
```

B.2 `distance`

Compute the great circle distance between the two geodesic points. The calculation assumes the Earth is a sphere. This algorithm ignores the altitudes.

```
distance(p1: LatLonAlt, p2: LatLonAlt): nnreal =
    distance_from_angle(angular_distance(lat(p1),lon(p1),lat(p2),
                                         lon(p2)),0)

distance_from_angle(angle: nnreal, h: nnreal): nnreal =
   (spherical_earth_radius + h) * angle
```

B.3 `initial_course` and `final_course`

The initial true course (course relative to true north) at point `p1` on the great circle route from point `p1` to point `p2`. The value is in internal units of angles (radians), and is a compass angle $[0..2\pi]$: clockwise from true north. If point `p1` and `p2` are close to each other, then the initial course may become unstable. In the extreme case when point `p1` equals point `p2`, then the initial course is undefined.

```
initial_course(p1: LatLonAlt, p2: LatLonAlt): real =
   LET d = angular_distance(lat(p1), lon(p1), lat(p2), lon(p2)) IN
        initial_course_impl(p1, p2, d);
```

```
final_course(p1: LatLonAlt, p2: LatLonAlt): real =
   initial_course(p2, p1) + pi;
```

```
initial_course_impl(p1: LatLonAlt, p2: LatLonAlt): real =
    LET lat1 = lat(p1)
        lon1 = lon(p1)
        lat2 = lat(p2)
        lon2 = lon(p2)
    IN
        IF cos(lat1) = 0 THEN % at either pole
          IF lat1 > 0 THEN
            pi        % north pole, all directions are south
          ELSE
            2.0 * pi  % south pole, all directions are north
          ENDIF
        ELSE
          to2pi(atan2(sin(lon2-lon1)*cos(lat2),
                          cos(lat1)*sin(lat2)
                          - sin(lat1)*cos(lat2)*cos(lon2-lon1)))
    ENDIF
```

Note that in a floating-point implementation it is necessary to include a small buffer on the test if the first point is at a pole.

B.4 angle_between

The `angle_between` function calculates the angle between two great circles that intersect at a point `b`. The first great circle also goes through point `a`, while the second great circle goes through point `b`. Assuming `b` is not one of the poles, this function is defined as follows:

```
angle_between(a:LatLonAlt, b:LatLonAlt, c:LatLonAlt): real =
    LET ang1 = initial_course(b,a),
        ang2 = initial_course(b,c)
    IN
        turnDelta(ang1,ang2);
```

where `turnDelta` calculates the difference between two angles as follows

```
turnDelta(alpha: real, beta: real): real =
   LET a = to2pi(alpha)
       b = to2pi(beta)
       delta = abs(a-b)
   IN
     IF (delta<=pi) THEN delta
     ELSE 2*pi-delta
     ENDIF
```

and

```
to2pi(a: real) : real =
  LET n = floor(a/(2*pi)) IN
     a - 2*n*pi
```

Note that if the input location `b` is precisely at one of the poles, an alternate version of the `angle_between` function is required:

```
angle_between(a:LatLonAlt, b:LatLonAlt, c:LatLonAlt): real =
   LET
      a1 = angular_distance(c,b)
      b1 = angular_distance(a,c)
      c1 = angular_distance(b,a)
      d = sin(c1)*sin(a1)
   IN
      IF d = 0.0 THEN PI
      ELSE
         acos( (cos(b1)-cos(c1)*cos(a1)) / d )
      ENDIF
```

This function is based on the spherical law of cosines. It depends upon the `angular_distance` function.

B.5 velocity_initial

The `velocity_initial` function computes the initial velocity on the great circle from point p1 to point p2 with the given amount of time.

```
velocity_initial(p1:LatLonAlt, p2: LatLonAlt p2, double t):
                                                       Velocity =
     LET d = angular_distance(p1, p2),
         gs = distance_from_angle(d, 0.0) / t
         crs = initial_course_impl(p1, p2, d)
     IN
         Velocity.mkTrkGsVs(crs, gs, (alt(p2) - alt(p1)) / t)
```

where the function `Velocity.mkTrkGsVs` returns a `Velocity` with the given track, ground speed, and vertical speed values. If points p1 and p2 are essentially the same

(about 1 meter apart), then a zero vector is returned. Also if the absolute value of time is less than 1 [ms], then a zero vector is returned. These details have been left out of the above definition.

B.6 linear_initial

The `linear_initial` function determines a point from the given lat/lon with an initial angle of `track` at a distance `dist`. This calculation follows the great circle.

```
  linear_initial(s: LatLonAlt, track: real, dist: real): LatLonAlt =
     linear_initial_impl(s, track, angle_from_distance(dist), 0.0)
```

where

```
linear_initial_impl(s: LatLonAlt, track: real, d: real, vert: real):
                                                               LatLonAlt =
           LET cosd = cos(d)
               sind = sin(d)
               sinslat = sin(lat(s))
               cosslat = cos(lat(s))
               lat = asin(sinslat*cosd + cosslat*sind*cos(track))
               dlon = atan2(sin(track)*sind*cosslat,
                            cosd - sinslat*sin(lat))
               lon = to_pi(lon(s) + dlon)
           IN
              LatLonAlt.mk(lat, lon, alt(s) + vert)
```

The function `LatLonAlt.mk` is a function that returns a `LatLonAlt` (geodesic coordinate) based on latitude, longitude, and altitude values.

Appendix C

Chordal Radius

There are two different notions of radius that arise when dealing with geodesic coordinates. They are:

- *Surface Radius* (R): the along-surface radius that is calculated using the great circle distance from the center of the turn to a point on the turn.
- *Chordal Radius* (R'): the part of a chord that is in the plane of the turn itself (which is a small circle) and hence passes through the sphere's volume.

For small-radius turns these are approximately the same, but for very large radius turns (on the order of hundreds of nautical miles), there can be a more noticable difference between the two values. Here are typical differences as a function of surface radius R;

R [NM]	$R - R'$ [NM]
1.00	0.00000001
2.00	0.00000011
3.00	0.00000038
4.00	0.00000090
5.00	0.00000176
6.00	0.00000305
7.00	0.00000484
8.00	0.00000722
9.00	0.00001028
10.00	0.00001410
50.00	0.00176281
100.00	0.01410206

The chordal radius can be obtained from the surface, great-circle radius using the following function:

```
to_chordal_radius(surface_radius: real): real =
    GreatCircle.chord_distance(surface_radius*2)/2
```

where `chord_distance` is defined as follows:

```
chord_distance(surface_dist: real): real =
    LET theta = angle_from_distance(surface_dist,0.0) IN
        2.0*sin(theta/2.0)*GreatCircle.spherical_earth_radius
```

C.1 Function `chord_distance`

The function `chord_distance` returns the straight-line chord distance (through a spherical earth) from two points on the surface of the earth.

```
chord_distance(double lat1, double lon1, double lat2, double lon2):
                                                              real =
     LET v1 = spherical2xyz(lat1,lon1)
         v2 = spherical2xyz(lat2,lon2)
     IN
         || v1 - v2 ||
```

where

```
 spherical2xyz(double lat, double lon): Vect3 =
    LET r = GreatCircle.spherical_earth_radius
        theta = PI/2 - lat
        x = r*sin(theta)*cos(lon)
        y = r*sin(theta)*sin(lon)
        z = r*cos(theta)
    IN
        (x,y,z)
```

and `GreatCircle.spherical_earth_radius` is computed as follows

```
    spherical_earth_radius: real =
       1/ (0.000539957 * pi / (180.0 * 60.0)) = 6366707.019493707 m
```

This definition of the radius of a spherical earth in meters assumes one nautical mile is 1852 meters (as defined by various international committees) and that one nautical mile is equal to one minute of arc (traditional definition of a nautical mile) on the Earth's surface. This value lies between the major and minor axis as defined by the "reference ellipsoid" in WGS84 [28].

Appendix D

Ground Speed vs. Airspeed

Because aircraft fly with respect to airspeed, many trajectory definitions have been defined with respect to airspeed. However, there is a serious problem with communicating trajectories based on airspeed: the future location of the aircraft relative to the ground and most likely to other aircraft, can not be known unless one has access to the wind field used by the creator of this trajectory.[D1] Consequently important ATM functions such as conflict detection and resolution cannot be performed without first converting the trajectories in terms of airspeed into trajectories in terms of ground speed. We believe that the original creator of a trajectory should perform this translation rather than the recipients.

Because times are stored in the internal data structures of EUTL, it is natural to focus on times rather than speeds. But, if one's focus is changed to speed rather than time, then some additional flexibility can be obtained from the EUTL language. With this focus, the trajectory defined in EUTL can be thought of as a two-dimensional path augmented by a speed profile and an altitude profile. In fact, if one suspends the idea of using time in an absolute sense, the speed profile can be either an airspeed profile or a ground speed profile. Admittedly, this can be awkward, but with care, some useful additional capabilities can be created.

If we let P_{as} be a plan where the speeds are interpreted as airspeed and P_{gs} be a plan where the speeds are interpreted as ground speed, then the following conversion can be accomplished.

$$P_{\mathsf{as}} + W \longrightarrow P_{\mathsf{gs}}$$

where W is a wind field. This conversion can be accomplished by only modifying the times if one makes the *assumption* that the wind only affects the along-track position of an aircraft. This is a reasonable assumption because the control system and pilot work to maintain lateral conformance to the flight plan. We also note that W can represent the difference between wind field used to create P_{as} and the wind field of the recipient P_{gs}. This is the approach used in the medium fidelity trajectory generator in TBO-TIGAR, see Section 8.

[D1] It would, of course, be possible to transmit this data, but this could introduce a significant communication bandwidth cost.

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

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.
PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
01-09-2017	Technical Memorandum	

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
An Efficient Universal Trajectory Language	
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	**5d. PROJECT NUMBER**
George E. Hagen, Jeffrey M. Maddalon, Nelson M. Guerreiro, Ricky W. Butler,	
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
NASA Langley Research Center Hampton, Virginia 23681-2199	L–20870
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	**10. SPONSOR/MONITOR'S ACRONYM(S)**
National Aeronautics and Space Administration Washington, DC 20546-0001	NASA
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)
	NASA/TM-2017-219669

12. DISTRIBUTION/AVAILABILITY STATEMENT

Unclassified-Unlimited
Subject Category 63
Availability: NASA STI Program(757) 864-9658

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The Efficient Universal Trajectory Language (EUTL) is a language for specifying and representing trajectories for Air Traffic Management (ATM) concepts such as Trajectory Based Operations (TBO). In these concepts, the communication of a trajectory between an aircraft and ground automation is fundamental. Historically, this trajectory exchange has not been done, leading to trajectory definitions that have been centered around particular application domains and, therefore, are not well suited for TBO applications. The EUTL trajectory language has been defined in the PVS formal specification language, which provides an operational semantics for the EUTL language. The hope is that EUTL will provide a foundation for mathematically verified algorithms that manipulate trajectories. Additionally, the EUTL language provides well-defined methods to unambiguously determine position and velocity information between the reported trajectory points. In this paper, we present the EUTL trajectory language in mathematical detail.

15. SUBJECT TERMS

air traffic management, trajectory, flight plan, trajectory specification, trajectory-based operations

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			STI Help Desk (email: help@sti.nasa.gov)
					19b. TELEPHONE NUMBER *(Include area code)*
U	U	U	UU	69	(757) 864-9658