NASA/TM–2013-218030

The Chorus Conflict and Loss of Separation Resolution Algorithms

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

August 2013

NASA STI Program . . . in Profile

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

NASA/TM–2013-218030

The Chorus Conflict and Loss of Separation Resolution Algorithms

Ricky W. Butler, George E. Hagen, and Jeffrey Maddalon
Langley Research Center, Hampton, Virginia

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

August 2013

Acknowledgments

The authors want to thank Anthony Narkawicz and César Muñoz for their help in developing and tuning these algorithms. Their expertise in the mathematical criteria theory was essential to us achieving a set of algorithms that conform to this criteria.

Available from:

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

Abstract

The Chorus software is designed to investigate near-term, tactical conflict and loss of separation detection and resolution concepts for air traffic management. This software is currently being used in two different problem domains: en-route self-separation and sense and avoid for unmanned aircraft systems. This paper describes the core resolution algorithms that are part of Chorus. The combination of several features of the Chorus program distinguishes this software from other approaches to conflict and loss of separation resolution. First, the program stores a history of state information over time which enables it to handle communication dropouts and take advantage of previous input data. Second, the underlying conflict algorithms find resolutions that solve the most urgent conflict, but also seek to prevent secondary conflicts with the other aircraft. Third, if the program is run on multiple aircraft, and the two aircraft maneuver at the same time, the result will be implicitly coordinated. This implicit coordination property is established by ensuring that a resolution produced by Chorus will comply with a mathematically-defined criteria whose correctness has been formally verified. Fourth, the program produces both instantaneous solutions and kinematic solutions, which are based on simple acceleration models. Finally, the program provides resolutions for recovery from loss of separation. Different versions of this software are implemented as Java and C++ software programs, respectively.

Contents

1 Introduction

The Chorus software is designed to investigate near-term, tactical conflict detection and resolution concepts for air traffic management. This software is used in two air traffic domains, self-separation concepts [10, 11] and unmanned aircraft systems [3]. This paper describes the core conflict resolution algorithms that are part of Chorus. The combination of several features of the Chorus program distinguishes this software from other approaches to conflict resolution [4]. First, the program stores a history of state information over time which enables it to handle transient communication dropouts, and produce solutions that are sensitive to past inputs. Second, the underlying algorithms attempt to resolve both primary and secondary conflicts with all traffic aircraft. *Primary* conflicts are those conflicts that exist on the current trajectory of the aircraft. *Secondary* conflicts are those conflicts which arise when an aircraft is maneuvering to resolve a primary conflict. Single resolutions to solve conflicts for multiple aircraft are known in the literature as a "1 to N" solutions. Third, if the program is run on multiple aircraft, then the conflict resolutions between any two aircraft are implicitly coordinated. This implicit coordination property is established by ensuring that resolutions produced by Chorus will comply with a mathematically-defined criteria whose correctness has been formally verified [1, 2, 6, 8]. Finally, the program was developed using a criteria theory that has undergone rigorous mathematical verification [1, 2, 6, 8]. That is, the net effect of both aircraft maneuvering at the same time is beneficial and will avoid a loss of separation. Fourth, the program produces both instantaneous solutions and kinematic solutions, which are based on simple acceleration models. Finally, the program provides resolutions for recovery from loss of separation (LoS) in addition to conflicts.

Chorus provides several different resolution algorithms that are useful for different applications. All of the Chorus algorithms use iterative search to find resolutions. The algorithms search over a range of potential solutions and methodically step through them to determine if they meet the desired properties. This is a change from our earlier work where we relied exclusively on analytical solutions [5]. A key advantage of the analytical solutions is that they are known to satisfy the desired properties based on an a-priori mathematical analysis and are often computationally more efficient. The advantage of the iterative approach used in Chorus is that the desired properties of the resolution can be much more complicated, but this comes at the cost of higher execution times. Although all of the Chorus resolution algorithms are iterative, in certain cases where an iterative solution is not found, an analytical solution from the ACCoRD `CRSS`[1] resolver is returned. The `CRSS` conflict resolution algorithm has been formally verified using the PVS theorem prover [9].

Resolution algorithms also differ in their design assumptions about how the aircraft will change from its current state (position and velocity) to the future resolution state. Chorus provides algorithms based on two different approaches: instantaneous and kinematic change. An instantaneous maneuver assumes the aircraft can instantly change from its current state to the target state. A kinematic solution

[1] See [5] and other references at `http://shemesh.larc.nasa.gov/fm/fm-atm-cdr.html`

provides a simple model that uses a specified constant acceleration to achieve the target resolution state.

Another important aspect of a conflict resolution algorithm is the goal of the algorithm, in particular, the safety property that the resolution algorithm is seeking to maintain. Chorus provides algorithms for two different goals: conflict avoidance and loss of separation recovery. Conflict avoidance is a safety goal that ensures that the aircraft will not be closer than given horizontal and vertical minimum distances, typically 5 nautical miles horizontally and 1000 feet vertically. The goal of a loss of separation algorithm is to find a resolution that quickly separates the aircraft and prevents a collision. The Chorus loss of separation algorithms produce resolutions that increase the distance at the closest point of approach and avoid nearly parallel final states. Note that this is a weaker property than conflict avoidance, but it can be applied when separation has already been lost.

Sections in this paper describing the resolution algorithms are summarized in table 1.

	instantaneous	kinematic
conflict	Section 5	Section 6
loss of separation (LoS)	Section 7	Section 7

Table 1. Resolution Algorithms in Chorus

The Chorus software has been released as open-source software and is available upon request. See `http://shemesh.larc.nasa.gov/fm/fm-atm-codes.html` for help in obtaining this software.

2 Assumptions

For the Chorus software to operate correctly, certain contextual assumptions must be satisfied. The most significant assumption is one of consistent shared knowledge. This is important for both conflict detection and for consistent criteria calculations. Specifically it is assumed that state information is accurate, within an error bound, and that any two aircraft in a conflict situation are using the same values. This can be achieved by having each aircraft send its state data according to a schedule based on a global clock. This ensures all aircraft use the same set of broadcasts as a basis for their calculations. In addition, the state data used for an aircraft's internal calculations must be the same as the values that it broadcasts to other aircraft. This data is likely to be a discretized version of the native sensor data. For example, values could be read and timestamped on each second and broadcast before the next second. Calculations would then be based on data with timestamps that are at least a full second earlier than the current time. The time latency in the data can be compensated using standard linear projection. Fortunately, the mathematical criteria that serves as a basis of these algorithms calculates direction parameters that are insensitive to linear time projection.

Additionally, there is an assumption of a localized global awareness of traffic. That is, if aircraft A is considered to be the most urgent traffic by aircraft B, it is essential that A consider B to be its most urgent aircraft in order for there to be a guarantee of implicit coordination. There are subtle, multiple aircraft encounters where this property may not hold. For example, suppose that aircraft A is in conflict with aircraft B, but it is not in conflict with aircraft C. Further, suppose that aircraft B is in conflict with both C and A, but it loses separation with C before it loses separation with A. In this case, A views B as most urgent, while B views C as most urgent. Thus, aircraft A and aircraft B hold incompatible views about their most urgent aircraft. One subject of future work is to develop approaches to achieving compatible results for the most urgent aircraft even in complex scenarios such as these. It is important that this work take into consideration broadcast distances, which makes this problem even more difficult.

The positions and velocities used in these algorithms are specified as 3-dimensional inertial Euclidean coordinates. Measurements in other coordinate frames, such as air-relative or geodetic frames, need to be projected into an Euclidean space as appropriate. The Chorus software automatically performs these projections if the input data is geodetic although we do not present the details of those projections in this paper. The East-North-Down projections used in Chorus are believed to be valid given the short distances and short time horizon of the Chorus algorithms.

3 Notation

The Chorus algorithms are captured in two versions written in an object-oriented style in both the Java and C++ programming languages, respectively. The Chorus algorithms are documented in this paper in a pseudocode that is neither Java nor C++. Instead, this pseudocode is a hybrid language incorporating features of both functional languages and object-oriented languages. The purpose of the pseudocode is to capture the major logic of the algorithms without the distracting details of a normal programming language.

Common operations on sets or lists are assumed, so a list size function exists, and elements can be accessed by index, as indicated by array subscript notation `list[x]`. Similarly, standard mathematical vectors are available, `Vect2` for a 2-dimensional vector and `Vect3` for a 3-dimensional vector. In addition, standard arithmetic on vectors is assumed, so for vectors `u`, `v`, and `w` and double `a`, `v = u+w*a` represents the assignments:

```
v.x = u.x + w.x * a
v.y = u.y + w.y * a
v.z = u.z + w.z * a
```

The pseudocode assumes the existence of an n-ary tuple data type, denoted by `(x1,..,xn)`, which allow piecewise assignment,

```
(x1,...,xn) = (y1,...,yn)
```

thus assigning `x1` the value `y1`, etc. In the Java and C++ source code, these operations are often implemented as separate instance variables using appropriate accessor functions, and sometimes may be set as side effects as opposed to being returned explicitly in a function.

Additionally, in order to simplify the presentation, pass-through parameters to longer function calls may be omitted. For example,

```
criteria(s,vo,vi,vo',minRelVs)
```

will generally be represented as

```
criteria(...,vo',...)
```

Certain data that is expressed here as parameters may be implicitly passed as instance variables in the Java and C++ source code implementation.

The definitions of the operators (`+`, `-`, `*`, `/`, `==`, `!=`, etc.) are the definitions from Java or C++, without relying on exotic behavior such as overflow. We use `AND` and `OR` for the boolean connectives rather than `&&` and `||` used in Java and C++. The operator `~=` is defined to mean "almost equals," which means that the values are compared and if they are within a fixed floating point precision of each other, then they are considered to be equal.

The loop statement syntax is taken from Java and C++ and, like these languages, it has the semantics of a for loop.

```
for (initialization; test; increment) { ... }
```

Our notation for function definitions departs from Java/C++ syntax in that the return type of the function follows the parameters. For example, a function that computes the sine function would be declared as follows:

```
sine(x): double  { ... }
```

Note that the types of the function parameters are not listed. We also allow an N-tuple return type:

```
func(x,y,z):  (a,b)  { ... }
```

Here the function `func` has three parameters and returns two values with types: `a` and `b`.

4 Common Variables and Functions

The following variables are commonly used in the Chorus algorithms:

`so`	initial ownship position ($\mathbf{s_o}$)
`vo`	initial ownship velocity ($\mathbf{v_o}$)
`si`	initial traffic position ($\mathbf{s_i}$)
`vi`	initial traffic velocity ($\mathbf{v_i}$)
`s`	initial relative position ($\mathbf{s}$)
`v`	initial relative velocity ($\mathbf{v}$)

The algorithms use a vector library which provides the ability to algebraically manipulate vectors. The following are common functions:

`v.x`	x component of vector $\mathbf{v}$
`v.y`	y component of vector $\mathbf{v}$
`v.z`	z component of vector $\mathbf{v}$
`vo.trk()`	returns track component of velocity $\mathbf{v_o}$
`vo.gs()`	returns ground speed component (2-D vector magnitude) of velocity $\mathbf{v_o}$
`vo.vs()`	returns vertical speed component of velocity $\mathbf{v_o}$
`vo.mkTrk(trk)`	creates a 3-dimensional velocity vector from $\mathbf{v_o}$, where the track component is changed to `trk`
`vo.mkGs(gs)`	creates a 3-dimensional velocity vector from $\mathbf{v_o}$, where the ground speed component is changed to `gs`
`vo.mkVs(vs)`	creates a 3-dimensional velocity vector from $\mathbf{v_o}$, where the vertical speed component is changed to `vs`

Additionally, the following function is used to define loss of separation between two aircraft:

`LoS(s,D,H)`	returns `TRUE` if the relative position `s` is in loss of separation with respect to a protection zone with horizontal separation D and vertical separation H. This is defined as: `s.x*s.x + s.y*s.y < D*D AND |s.z| < H`

The following are parameters that are assumed to be globally accessible and therefore these variables are not explicitly passed in the pseudocode:

`D`	minimum horizontal separation
`H`	minimum vertical separation
`Tres`	lookahead time for resolution
`aircraftList`	list of all traffic aircraft. The ownship aircraft is not included in this list. The information for each aircraft in this list can be used to return its position and velocity (possibly extrapolated) at a given time.
`minGs`	minimum ground speed allowed for ownship
`maxGs`	maximum ground speed allowed for ownship
`minVs`	minimum vertical speed allowed for ownship
`maxVs`	maximum vertical speed allowed for ownship
`checkSecondary`	a boolean flag, where `TRUE` indicates aircraft other than the most urgent are considered

5 Instantaneous Conflict Algorithms

Chorus' instantaneous conflict resolution algorithm attempts to find a resolution that is free of conflicts with all traffic aircraft. However, if a resolution cannot be

found that satisfies this property, then a resolution that is free of conflicts with only the *most urgent* aircraft is returned. How Chorus determines the aircraft that is *most urgent* is described in section 5.1. The algorithm proceeds in two steps. The first step finds a starting point for iteration (track angle, ground speed, and vertical speed) using the analytical solutions from the `CRSS` algorithm. The `CRSS` class implements the ACCoRD algorithms that have been formally verified[2]. The first step of the instantaneous conflict algorithm is as follows:

```
resolution_i = CRSS.resolution(s,vo,vi,epsh,epsv)
double trk0 = 0
double gs0 = 0
double vs0 = 0
if (CRSS.hasTrkOnly()) trk0 = CRSS.trkOnly()
if (CRSS.hasGsOnly())  gs0 =  CRSS.gsOnly()
if (CRSS.hasVsOnly())  vs0 =  CRSS.vsOnly()
```

The variable `resolution_i` is a status flag and it is used to indicate the nature of the resolution, for instance, "no conflict" or "loss of separation."

The second step of the algorithm seeks to avoid secondary conflicts by iteratively searching starting from the computed values `trk0`, `gs0`, and `vs0`. In all of the algorithms, the search direction (`dir`) is determined by the mathematical criteria. Before we discuss the instantaneous iterative resolution algorithms, we introduce three key functions: `mostUrgent`, `nDetector` and `criteria` in the next three sections.

5.1 Most Urgent Aircraft

An aircraft may have multiple conflicts with other aircraft on its current trajectory or be in loss of separation with more than one aircraft. In the conflict case, Chorus determines which aircraft has the *most urgent* conflict, that is, the conflict with the minimum time into loss of separation. The solution to this conflict serves as a starting point for solving any remaining primary or secondary conflicts. In the loss of separation case, the determination of most urgent is more subtle. The function that determines the most urgent conflict uses the following ranking (highest to lowest):

1. Aircraft that are in loss of separation with the ownship that have convergent velocities, with the smallest cylindrical distance at time of closest approach [7] ranked first.

2. Aircraft that are in loss of separation with the ownship having divergent velocities are ranked by the current horizontal distance.

3. Aircraft that are in conflict with the ownship, with those that are closest (in time) to a loss of separation ranked higher.

4. All other aircraft are ranked by current horizontal distance.

[2]See `http://shemesh.larc.nasa.gov/fm/fm-atm-cdr.html`

Resolving conflicts with multiple aircraft is quite challenging. There are cases where the traffic density is so high that the ownship cannot find a suitable maneuver. Although resolution of all conflicts is desirable, it is not always possible and a compromise must be made. In these cases, Chorus provides a resolution that solves the most urgent conflict and a flag is set to indicate unresolved conflicts. Furthermore, implicit coordination is only guaranteed with respect to the most urgent aircraft.

5.2 Multiple Aircraft Conflict Probe: `nDetector`

The `nDetector` function performs a conflict probe between the position and velocity of one aircraft (usually the ownship) and all other traffic aircraft. The iterative resolution algorithms described in this paper use the `nDetector` function to check potential resolutions, not just the current state of the ownship. Thus, the data sent to the `nDetector` function are possible ownship resolutions, not the actual ownship state data. This function requires the following parameters:

`so`	position of the ownship
`vo`	velocity vector of the ownship
`tOwn`	time when `so` and `vo` were collected
`mu`	index of the most urgent aircraft
`ignore`	index of aircraft that should be ignored. Use -1 to accept all aircraft

The function is defined as follows:

```
nDetector(so,vo,tOwn,mu,ignore) : boolean {
   for (j = 0; j < aircraftList.size(); j++) {
     T = Tres
     if (j == mu) T = MAXDOUBLE
     if (j != ignore) {
        (si,vi) = predLinear(aircraftList[j],tOwn)
        Vect3 s = so - si
        if (CDSS.conflict(s,vo,vi,D,H,T)) return TRUE
     }
  }
  return FALSE
}
```

The function `predLinear(ac, tOwn)` estimates the position and velocity vectors for the traffic aircraft `ac` at time `tOwn`, using a linear extrapolation from the aircraft's last position and velocity. The `CDSS.conflict` function performs the conflict detection and it is defined in Appendix B. If there is a conflict with any aircraft the `nDetector` function returns `TRUE`, otherwise it returns `FALSE`. The time parameter, `T`, indicates how far in the future conflicts should be checked. Setting `T` to `MAXDOUBLE` ensures resolutions will be sufficient to fully resolve the immediate conflict and any secondary conflicts with the most urgent aircraft.

The `ignore` value represents an aircraft to be ignored by the conflict detection. It is set to -1 (an invalid index) for primary detection (i.e. include all aircraft) and the index of the most urgent traffic for secondary conflict detection.

5.3 Coordinated Maneuver Criteria: `criteria`

Two resolution algorithms are said to be coordinated if they produce resolutions that solve the conflict when either one aircraft maneuvers or both aircraft manuever simultaneously. Coordination criteria is a boolean function that takes as inputs the state information for two aircraft. Criteria serves as a mathematical requirement for resolution algorithms such that if any two algorithms satisfy the criteria, then resolutions from those algorithms are coordinated. An elaborate mathematical theory of coordination criteria is presented in [2, 6, 8]. This theory provides important system-wide safety properties, which have been mathematically proven and proofs are contained in the references.

In the Chorus resolution algorithms, criteria are used to set the search direction. Since the initial resolution (from the `CRSS` algorithm) is already coordinated, setting the iterative search direction in the coordinated direction ensures that an iterated solution is also coordinated.

The function `criteria` requires the following parameters:

`s`	position of the ownship relative to the traffic aircraft
`vo`	velocity of the ownship aircraft
`vi`	velocity of the traffic aircraft
`vo'`	velocity vector of the ownship to be checked against the criteria
`minRelVs`	desired minimum relative exit vertical speed (used in LoS only)

5.4 Track Algorithm

The track algorithm depends upon both the `nDetector` conflict detector and the `criteria` function. The algorithm uses iteration on the variable `trkDelta` in the direction `dir` $= \pm 1$ and with an increment size `step`. At each iteration step a new track is computed: `trk0+dir*trkDelta` where `trk0` is the solution obtained from `CRSS`. The iteration is continued until the `nDetector` function indicates there is no conflict with any traffic aircraft and `criteria` indicates that the new vector meets the implicit coordination criteria, or until 180° have been searched. If a track solution cannot be found that solves the conflict with the most urgent aircraft (primary conflict) and also avoids secondary conflicts, then the resolution that only resolves the conflict with the most urgent aircraft, which is supplied by the `CRSS` function, is returned.

```
inst_track(so,vo,tOwn,mu,trk0,dir,step) : (boolean,double) {
  hasTrk = FALSE
  for (trkDelta = 0; trkDelta < PI; trkDelta = trkDelta + step) {
```

```
    trk = trk0 + dir * trkDelta
    vo' = vo.mkTrk(trk)
    if ( NOT nDetector(so,vo',tOwn,mu,-1) AND criteria(..., vo',...)) {
      (hasTrk,trkOnly) = (TRUE,trk)
      break
    }
  }
  if ( NOT hasTrk) {
    (hasTrk,trkOnly) = (CRSS.hasTrkOnly(),CRSS.trkOnly())
  }
  return (hasTrk,trkOnly)
}
```

5.5 Ground Speed Algorithm

The ground speed algorithm depends upon the `nDetector` conflict detector and the `criteria` function. The algorithm uses iteration on the variable `gsDelta` in the direction `dir` $= \pm 1$ and with an increment size `step`. At each iteration step a new ground speed is computed: `gs0+dir*gsDelta`, where `gs0` is the ground speed solution obtained from `CRSS`. The iteration is continued until the `nDetector` function indicates there is no conflict with any traffic aircraft and `criteria` indicates that the new vector meets the implicit coordination criteria, or until the ground speed exceeds the preset `minGs, maxGs` bounds. If no ground speed solution can be found that resolves the conflict with the most urgent aircraft (primary conflict) and that also avoids secondary conflicts, then a resolution that only resolves the conflict with the most urgent aircraft is returned. This resolution is computed by the `CRSS` function,

```
inst_gs(so,vo,tOwn,mu,gs0,dir,step) : (boolean,double) {
   for (gsDelta = 0; ; gsDelta = gsDelta + step) {
      nvGs = gs0 + dir * gsDelta
      if (nvGs < minGs OR nvGs > maxGs ) break
      vo' = vo.mkGs(nvGs)
      if ( NOT nDetector(so,vo',tOwn,mu,-1) AND criteria(..., vo',...)) {
         (hasGs,gsOnly) = (TRUE,nvGs)
         break
      }
   }
   if ( NOT hasGs) {
     (hasGs,gsOnly) = (CRSS.hasGsOnly(),CRSS.gsOnly())
   }
 return (hasGs,gsOnly)
}
```

5.6 Vertical Speed Algorithm

The vertical speed algorithm depends upon the `nDetector` conflict detector and the `criteria` function. The algorithm uses iteration on the variable `vsDelta` in the direction `dir` $= \pm 1$ and with an increment size `step`. At each iteration step a new track is computed: `vs0+dir*vsDelta` where `vs0` is the vertical speed solution obtained from `CRSS`. The iteration continues until the `nDetector` function indicates there is no conflict with any traffic aircraft and `criteria` indicates that the new vector meets the implicit coordination criteria, or until the vertical speed exceeds the preset `minVs, maxVs` bounds. If no vertical speed solution can be found that solves the primary conflict (with the most urgent aircraft) and also avoids secondary conflicts, then the resolution that only resolves the conflict with the most urgent aircraft (which is computed by the `CRSS` function) is returned.

```
inst_vs(so,vo,tOwn,mu,vs0,dir,step) : (boolean,double) {
  for (vsDelta = 0; ; vsDelta = vsDelta + step) {
    nVs = vs0 + dir * vsDelta
    if ((nVs > maxVs) OR (nVs < minVs)) break
    vo' = vo.mkVs(nVs)
    if ( NOT nDetector(so,vo',tOwn,mu,-1) AND criteria(..., vo',...)) {
       (hasVs,vsOnly) = (TRUE,nVs)
       break
    }
  }
  if ( NOT hasVs) {
     (hasVs,vsOnly)  = (CRSS.hasVsOnly(),CRSS.vsOnly())
  }
return (hasVs,vsOnly)
}
```

6 Kinematic Conflict Algorithms

The kinematic algorithms are more complex than the instantaneous algorithms, but provide better solutions because they take into consideration the fact that aircraft must maneuver over time to achieve the resolutions. Since the aircraft converge over time, the kinematic algorithms produce larger resolutions than the instantaneous algorithms. The difference between these resolutions will grow depending on how close the two aircraft are to each other, as shown in figure 1. In this figure, the original velocity is shown in red, the instantaneous solution is the straight resolution vector and the kinematic solution is the curved resolution vector. The kinematic solvers use models of aircraft dynamics with an assumed constant acceleration to project the positions of the ownship and traffic as the iteration as time progresses. The kinematic models used by these algorithms are defined in Appendix A. The determination of the *most urgent* conflict is described in section 5.1.

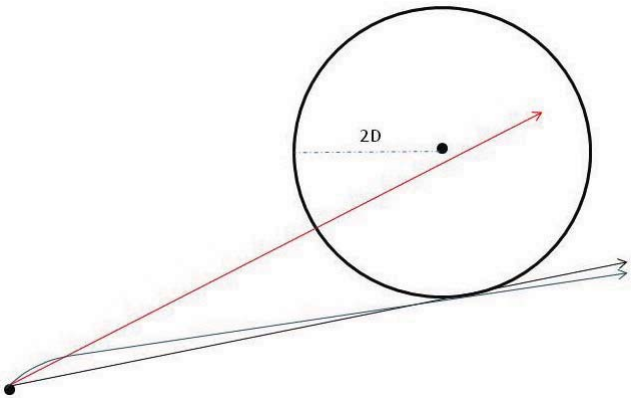

Figure 1. Kinematic and Instantaneous Trajectories for Conflict Resolution

6.1 Future Conflict Detection: `nDetectorConfFut`

The function `nDetectorConfFut` uses a kinematic projection of the future position of the aircraft and performs a conflict probe at the future point.

This function performs conflict detection with respect to all aircraft on a future state of the ownship. The ownship future state is passed as a parameter. The traffic states are projected linearly using a relative time (`delTm`). It uses a list of traffic state data called `aircraftList`. A function `predLinear` is used to interpolate the traffic data linearly to match the times of the ownship. It is expected that most of the time the interpolation will be unnecessary because the time stamps should match (except during communications dropouts).

The function `nDetectorConfFut` returns a status per table 2 and has the following parameters:

`soFut`	the future position of the ownship
`v'`	the velocity vector to be checked for conflicts
`delTm`	the amount of time that the ownship is projected into the future (i.e., the time after `tOwn` when `soFut` is valid)
`mu`	the most urgent aircraft index
`tOwn`	timestamp for the current position and velocity (i.e., $\mathbf{s_o}$ and $\mathbf{v_o}$)

Pseudocode for the function is:

```
nDetectorConfFut(soFut,v',tOwn,delTm,mu) : int {
  int rtn = -1
  for (i = 0; i < aircraftList.size(); i++) {
    (si,vi) = predLinear(aircraftList[i],tOwn)
    Vect3 s = soFut - si
    if (LoS(s,D,H)) {
       if (i == mu) return 3   // LoS with primary
```

-1	no conflict
0	secondary conflicts, no LoS, but is conflict free with most urgent aircraft
1	primary conflict
2	LoS with an aircraft that is not the most urgent one
3	LoS with most urgent aircraft

Table 2. Values returned from `nDetectorConfFut`

```
      else rtn = 2
    }
    Tresolve = Tres
    if (i == mu) Tresolve = MAX_VALUE
    boolean conf = CDSS.conflict(s,v',vi,D,H,Tres)
    if (conf AND rtn < 0) rtn = 0
    if (i == mu AND conf AND rtn < 1) rtn = 1
  }
  return rtn
}
```

6.2 Common Variables

In addition to `D`, `H`, `Tres`, and `aircraftList` described in section 4, the following variables are assumed to exist in the kinematic calculations:

`maxBank`	maximum bank angle allowed for ownship
`gsAccel`	ground speed acceleration allowed for ownship
`vsAccel`	vertical speed acceleration allowed for ownship
`minRelGs`	desired minimum relative speed for horizontal exit (used in LoS only)
`minRelVs`	desired minimum relative speed for vertical exit (used in LoS only)

6.3 Kinematic Track Algorithm

The kinematic track algorithm iteratively searches for a solution that satisfies the criteria and is also free of secondary conflicts. It has the following parameters:

`so`	position of the ownship
`vo`	velocity of the ownship
`tOwn`	timestamp for the so,vo data
`mu`	index of the most urgent aircraft
`step`	the iteration step size

Pseudocode for the function is:

```
kinematicTrack(so,vo,tOwn,mu,step) : (boolean,double) {
  omega = turnRate(vo.gs(),dir*maxBankAngle)
```

```
  nDetKin = 0
  crit = TRUE
  nvTrk = 0
  Vect3 firstFound = ZERO
  hasTrk = FALSE
  for (trkDelta = 0; trkDelta < PI; trkDelta = trkDelta + step) {
    tm = dir * trkDelta / omega
    nvTrk = vo.trk() + dir * trkDelta
    (soAtTm,vo') = turnOmega(so,vo,tm,omega)
    nDetKin = nDetectorConfFut(soAtTm,vo',tOwn,tm,mu)
    crit = criteria(..., vo',...)
    if (nDetKin == 0 AND crit AND firstFound == ZERO) firstFound = vo'
    if (nDetKin >= 2) break  // check for LoS
    if (nDetKin < 0 AND crit) {
       (hasTrk,trkOnly) = (TRUE,nvTrk)
       break
    }
  }
  if ( NOT hasTrk AND nDetKin < 2 AND firstFound != ZERO) {
    (hasTrk,trkOnly) = (TRUE,firstFound.trk())
  }
  if ( NOT hasTrk AND nDetKin >= 2) {
    if (infeasibleUseFutureLos) {
      if (nDetKin == 3) {  // entered LoS while iterating
        (hasTrk,trkOnly) = projectAndUseKinTrkLoS(tm,mu,step)
      } else {
        (hasTrk,trkOnly) = (TRUE,nvTrk)
      }
    } else {
      if (nDetKin == 3) {
        (hasTrk,trkOnly) = (CRSS.hasTrkOnly(),CRSS.trkOnly())
      }
    }
  }
  return (hasTrk,trkOnly)
}
```

A variable `firstFound` holds the first vector that is conflict-free with respect to the most urgent conflict. If a solution free of secondary conflicts is not found, the algorithm reverts to this solution. If the returned `hasTrk` value is `FALSE`, the returned `trkOnly` value is undefined.

This algorithm contains a configuration flag, `infeasibleUseFutureLos`, which describes what should be done with the iteratively discovered solution that still has at least one loss of separation. If `infeasibleUseFutureLos` is `FALSE` then the solution from `CRSS` is returned. If `infeasibleUseFutureLos` is `TRUE`, then one of two possible solutions is returned. If the only solutions found continue to have

conflicts, then the algorithm projects the aircraft states into the future to exactly one half second after entry into LoS and computes a LoS resolution, using the function `projectAndUseKinTrkLoS` that is described below. Otherwise it returns a solution that is free of conflicts with the most urgent aircraft, but still has losses of separation with other aircraft.

The projection into the future is accomplished by the function `projectAndUseKinTrkLoS` which is defined as follows:

```
projectAndUseKinTrkLoS(timeIntoLoS,mu,step) : (boolean,double) {
     si = aircraftList.get(mu).position(timeIntoLoS + 0.5)
     vi = aircraftList.get(mu).velocity(timeIntoLoS + 0.5)
     soFut = so.linear(vo,timeIntoLoS + 0.5)
     sFut = soFut - si
     if (sFut*(vo-vi) > 0 OR sFut.vect2().norm() < D/2.0) {
       (hasTrk,trkOnly) = (TRUE,vo.trk()+dir*pi/4.0)
     } else {
       (hasTrk,trkOnly) = trackLoS(soFut,si,vo,vi,mu,TRUE,step)
     }
}
```

The variable `timeIntoLos` is the computed relative time into LoS from the current position. The first if-branch is used for cases where the loss of separation entry point is the top or bottom of the protection zone, that is, there is already a horizontal loss of separation. The function `trackLoS` is defined in section 7.3.

6.4 Kinematic Ground Speed Algorithm

The kinematic ground speed algorithm iteratively searches for a solution that satisfies the criteria and is also free of secondary conflicts. It has the following parameters:

`so`	position of the ownship
`vo`	velocity of the ownship
`tOwn`	timestamp for the so,vo data
`mu`	index of the most urgent aircraft
`step`	step size for search

```
kinematicGroundSpeed(so,vo,tOwn,mu,step) : (boolean,double) {
  int nDetKin = 0
  Vect3 firstFound = ZERO
  hasGs = FALSE
  for (gsDelta = 0; TRUE; gsDelta = gsDelta + step) {
    double tm = gsDelta / gsAccel
    double nvoGs = vo.gs() + gsDir * gsDelta
    if ((nvoGs > maxGs) OR (nvoGs < 0)) break
    (soAtTm,vo') = gsAccel(so,vo,tm,gsDir * gsAccel)
    nDetKin = nDetectorConfFut(soAtTm,vo',tOwn,tm,mu)
    boolean crit = criteria(s,vo,vi,vo',...)
```

```
    if (nDetKin == 0 AND crit AND firstFound == ZERO) firstFound = vo'
    if (nDetKin >= 2) break     // check for LoS
    if (nDetKin < 0 AND crit) {
      (hasGs,gsOnly) = (TRUE,vo')
      break
    }
  }
  if ( NOT hasGs AND nDetKin < 2 AND firstFound != ZERO) {
    (hasGs,gsOnly) = (TRUE,firstFound.gs())
  }
  if (nDetKin >= 2) {    // some LoS or still in conflict
    if (infeasibleUseFutureLos) {
      si = aircraftList.get(mu).position(timeIntoLoS + 0.5)
      vi = aircraftList.get(mu).velocity(timeIntoLoS + 0.5)
      soFut = so.linear(vo,timeIntoLoS + 0.5)
      siFut = si.linear(vi,timeIntoLoS + 0.5)
      (hasGs,gsOnly) = gsLoS(soFut,vo,siFut,vi,TRUE,step)
    } else {
      (hasGs,gsOnly) = (CRSS.hasGsOnly(),CRSS.gsOnly())
    }
  }
  return (hasGs,gsOnly)
}
```

The variable `timeIntoLos` is the computed relative time into LoS from the current position. Just as in section 6.3, the `infeasibleUseFutureLos` flag is a configuration parameter that chooses between two different algorithms when the iterative solver cannot find a solution free of LoS for all aircraft. If the `infeasibleUseFutureLos` flag is `FALSE` then it reverts back to the `CRSS` solution. If the `infeasibleUseFutureLos` flag is `TRUE`, then the algorithm projects the aircraft states into the future to exactly one half second after entry into LoS and computes a LoS resolution. The function `gsLoS` is defined in section 7.4.

6.5 Kinematic Vertical Speed Algorithm

The kinematic vertical speed algorithm iteratively searches for a solution that satisfies the criteria and is also free of secondary conflicts. It has the following parameters

`so`	position of the ownship
`vo`	velocity of the ownship
`tOwn`	timestamp for the so,vo data
`mu`	index of the most urgent aircraft
`step`	step size for search

```
kinematicVerticalSpeed(so,vo,tOwn,mu) : (boolean,double) {
  int nDetKin = 0
```

```
  Vect3 firstFound = ZERO
  hasVs = FALSE
  for (vsDelta = 0; TRUE; vsDelta = vsDelta + step) {
    nVs = vo.vs() + dir * vsDelta
    if ((nVs > maxVs) OR (nVs < minVs)) break
    tm = |nVs - vo.vs()| / vsAccel
    (soAtTm,vo') = vsAccel(so,vo,tm,vsDir*vsAccel)
    nDetKin = nDetectorConfFut(soAtTm,vo',tOwn,tm,mu)
    boolean crit = criteria(s,vo,vi,vo',...)
    if (nDetKin == 0 AND crit AND firstFound == ZERO) firstFound = vo'
    if (nDetKin >= 2) break
    if (nDetKin < 0 AND crit) {
      (hasVs,vsOnly) = (TRUE,nVs)
      break
    }
  }
  if ( NOT hasVs AND nDetKin < 2 AND firstFound != ZERO) {
    (hasVs,vsOnly) = (TRUE,firstFound.vs())
  }
  if (nDetKin >= 1) { // LoS with primary or secondary or still conflict
    if (infeasibleUseFutureLos) {
      si = aircraftList.get(mu).position(timeIntoLoS + 0.5)
      vi = aircraftList.get(mu).velocity(timeIntoLoS + 0.5)
      soFut = so.linear(vo,timeIntoLoS + 0.5)
      siFut = si.linear(vi,timeIntoLoS + 0.5)
      (hasVs,vsOnly) = vsLoS(soFut,vo,si,vi,TRUE,step,epsv)
    } else {
      (hasVs,vsOnly) = (CRSS.hasVsOnly(),CRSS.vsOnly())
    }
  }
  return (hasVs,vsOnly)
}
```

The variable `timeIntoLos` is the computed relative time into LoS from the current position. Just as in section 6.3, the `infeasibleUseFutureLos` flag is a configuration parameter that chooses between two different algorithms when the iterative solver cannot find a solution free of LoS for all aircraft. If the `infeasibleUseFutureLos` flag is `TRUE`, the algorithm projects the aircraft states into the future to exactly one half second after entry into LoS and computes a LoS resolution. If the `infeasibleUseFutureLos` flag is `FALSE`, then it uses the `CRSS` solution. The function `vsLoS` is defined in section 7.5.

-1	no conflict
1	primary conflict
2	LoS with at least one aircraft that is not the most urgent aircraft
3	LoS with most urgent aircraft

Table 3. Values returned from `nDetectorLoSFut`

7 Loss of Separation Algorithms

The criteria [8] provides a notion of correctness for loss of separation (LoS) algorithms that is based on the concept of repulsion. A solution is *repulsive* if the distance at the time of closest approach between the two aircraft is greater than the current distance. This is a fairly weak property, but the algorithms obtain good results by seeking maximally repulsive solutions. That is, the iterations continue until the maximum vector that is repulsive is found. In the loss of separation case, the key factor is that the resolutions are implicitly coordinated. Therefore, the direction of the maneuver is far more important than the magnitude of the maneuvers. The LoS algorithms also seek to achieve a minimum relative exiting speed to avoid solutions that place the aircraft in nearly parallel trajectories at the end of the maneuvers.

Each of the resolution algorithms presented below can produce either a instantaneous solution or a kinematic solution, controlled by the parameter `kinematic`. Recall that an instantaneous solution is one where the aircraft is assumed to immediately achieve the maneuver whereas a kinematic solution is one where the aircraft must accelerate from its current position to achieve the resolution.

7.1 Detect Future LoS: `nDetectorLoSFut`

This function `nDetectorLoSFut` performs detection with respect to all aircraft (except a designated most urgent aircraft, which is ignored) on a future state. The ownship future state is passed as a parameter. The traffic states are projected linearly using a relative time (`delTm`). The return values are listed in table 3. The parameters are:

`soFut`	future position of the ownship
`nv`	velocity vector to be checked for conflicts
`tOwn`	timestamp for so, vo data
`delTm`	the amount of time that the ownship is projected into the future (i.e., the time after `tOwn` when `soFut` is valid)
`mu`	index of the most urgent aircraft

The `nDetectorLoSFut` function is used in the LoS case and hence it does not check for conflicts with the LoS aircraft (i.e. `mu`).

```
nDetectorLoSFut(soFut,nv,tOwn,delTm,mu) : int {
  int rtn = -1
```

```
  for (j = 0; j < aircraftList.size(); j++) {
    if (j != mu ) {
      (si,vi) = predLinear(aircraftList[j],tOwn)
      si = si.linear(vi,delTm)
      Vect3 s = soFut - si
      if (LoS(s,D,H)) {
        rtn = 2
        break
      }
      boolean conf = CDSS.conflict(s,nv,vi,D,H,Tres)
      if (conf AND rtn == -1) rtn = 1
    }
  }
  return rtn
}
```

The function `predLinear(tOwn)` retrieves the position and velocity vectors for the traffic aircraft for time `tOwn`. The `CDSS.conflict` function performs the conflict detection (See appendix B). If there is a conflict with any aircraft the `nDetectorLoSFut` function returns `TRUE`, otherwise it returns `FALSE`.

7.2 Determine Divergence: `divergentHorizGt`

A pair are aircraft are said to be *divergent* if the separation at all future positions is greater than their current separation. The function `divergentHorizGt` determines if the current state between the two aircraft is divergent and the relative speed is greater than a specified minimum relative speed:

```
divergentHorizGt(s,vo,vi) : boolean {
  Vect2 v = vo - vi
  return s.dot(v) > 0 AND v.norm() > minRelGs
}
```

7.3 Track LoS Algorithm

The following pseudocode describes the iterative LoS track algorithm. It can produce an instantaneous resolution or a kinematic one. The following parameters are used:

`so`	ownship's position
`vo`	ownship's velocity
`si`	traffic aircraft's position
`vi`	traffic aircraft's velocity
`mu`	index of the aircraft to be resolved against, usually the most urgent aircraft
`kinematic`	flag which determines whether a kinematic or an instantaneous solution is desired
`step`	step size for search

The `trackLos` function is defined as follows:

```
trackLoS(so,si,vo,vi,mu,kinematic,step) : (boolean,double) {
  vo' = ZERO
  omega = turnRate(vo.gs(),maxBank)
  boolean criteriaEverSatisfied = FALSE
  for (trkDelta = step; trkDelta < PI/2; trkDelta = trkDelta + step) {
    nvoTrk = vo.trk() + trkDir * trkDelta
    tm = 0.0
    Vect3 soAtTm = so
    Vect3 siAtTm = si
    if (kinematic) {
      tm = trkDelta / omega
      soAtTm = turnOmega(so,vo,tm,trkDir * omega).first
      siAtTm = si.linear(vi,tm)
    }
    sAtTm = soAtTm - siAtTm
    prevVo' = vo'
    vo' = vo.mkTrk(nvoTrk)
    boolean divg = divergentHorizGt(sAtTm,vo',vi)
    criteria = criteria(sAtTm,prevNvo,vi,vo',...)
    if (criteria) criteriaEverSatisfied = TRUE
    if (checkSecondary) {
      nDetKin = nDetectorLoSFut(soAtTm,vo',tOwn,tm,mu)
    }
    divgOrNotCrit = (divg OR NOT criteria)
    if (divgOrNotCrit) {
      if ( NOT criteria) lastSolution = prevVo'
        else lastSolution = nvo
      if (firstSolution == ZERO) firstSolution = lastSolution
      if ( NOT criteria OR nDetKin <= 0 OR nDetKin >= 2) break
    }
  }
  if (firstSolution == ZERO) {
     trkSolution = vo'
  } else if (divgOrNotCrit AND nDetKin <= 0) {
     trkSolution = lastSolution
  } else {
     trkSolution = firstSolution
  }
  hasTrk = (NOT (trkSolution = ZERO) AND criteriaEverSatisfied)
  return (hasTrk,trkSolution.trk())
}
```

The repulsive criteria is usually satisfied in one direction up to a particular track value. This algorithm seeks to find the maximally repulsive track, but will stop

earlier if divergence is reached. There are some cases where there are no repulsive tracks available in the search direction. Therefore a flag `criteriaEverSatisfied` informs `hasTrk`. If the non-criteria region is reached, there is no need to search further. When in the divergent region and the solution is free of all secondary conflicts, the search is stopped. Finally, if a secondary LoS occurs while searching, then the search is abandoned at that point. It is not clear whether it is better to return no track solution at this point, or just accept that a second LoS must occur and defer its resolution until later. The algorithm above does the latter. In the instantaneous (i.e., non-kinematic) version, the value of `sAtTm` is set to `s` and remains constant throughout the for loop. The tests at the end cover several different cases. The first branch covers the case where the criteria is satisfied all the way through the loop and divergence is not reached. In this case the maximal search value is used. The second branch covers the case where divergence or not criteria was reached, and so `lastSolution` is used. The last branch occurs when a LoS occurs or no solution free of secondary conflicts is found (`nDetKin > 0`). In this case the `firstSolution` value is used.

The parameter `minRelHoriz` impacts the timeliness of the speed of exit from loss of separation. If this parameter is set to 0, then the algorithm merely seeks to achieve some positive relative speed. But if a larger value is specified, the algorithm seeks to achieve a higher relative exiting speed. This is likely to result in a larger maneuver.

7.4 Ground Speed LoS Algorithm

The following parameters are used by the ground speed LoS algorithm

`so`	initial ownship position
`vo`	initial ownship velocity
`si`	initial traffic position
`vi`	initial traffic velocity
`mu`	the index of the aircraft to be resolved against, usually the most urgent aircraft
`kinematic`	flag which determines whether a kinematic or an instantaneous solution is desired
`step`	step size for search

The following pseudo code describes the iterative ground speed algorithm:

```
gsLoS(so,si,vo,vi,kinematic,step) : (boolean,double) {
  boolean divgOrNotCrit = FALSE
  boolean minSepOk = TRUE
  int nDetKin = 0
  Vect3 sAtTm = ZERO
  Vect3 vo' = vo
  Vect3 prevVo' = ZERO
  Vect3 firstSolution = ZERO
  Vect3 lastSolution = ZERO
```

```
boolean criteriaEverSatisfied = FALSE
for (gsDelta = step; TRUE; gsDelta = gsDelta + step) {
   nvoGs = vo.gs() + gsDir * gsDelta
   if ((nvoGs > maxGs) OR (nvoGs < minGs)) break
   tm = 0.0
   Vect3 soAtTm = so
   Vect3 siAtTm = si
   if (kinematic) {
     tm = gsDelta / gsAccel
     soAtTm = gsAccel(so,vo,tm,gsDir * gsAccel)
     siAtTm = si.linear(vi,tm)
   }
   sAtTm = soAtTm.Sub(siAtTm)
   prevVo' = vo'
   vo' = vo.mkGs(nvoGs)
   distBetw = sAtTm.vect2().norm()
   if (distBetw < minSep) minSep = distBetw
   boolean divg = divergentHorizGt(sAtTm,vo',vi)
   boolean criteria  = criteria(sAtTm,prevNvo,vi,vo',...)
   if (criteria) criteriaEverSatisfied = TRUE
   if (minSep <= gsLosDiscard)) break
   divgOrNotCrit = (divg OR NOT criteria)
   if (divgOrNotCrit) {      // save first solution
     if ( NOT criteria) lastSolution = prevVo'
       else lastSolution = vo'
     if (firstSolution == ZERO) firstSolution = lastSolution
     }
   }
 if (checkSecondary) {
   nDetKin = nDetectorLoSFut(soAtTm,vo',tOwn,tm,mu)
 }
 if (divgOrNotCrit) {
   if ( NOT criteria OR nDetKin <= 0 OR nDetKin >= 2) break
 }
}
if (firstSolution == ZERO) {
  gsSolution = prevVo'
} else if (divgOrNotCrit AND nDetKin <= 0) {
  gsSolution = lastSolution
} else {
  gsSolution = firstSolution
}
boolean divg0 = divergentHorizGt(sAtTm,vo',vi)
double distAtTau = distAtTau(sAtTm,vo',vi,TRUE)
if ( NOT divg0 AND distAtTau <= ChorusConfig.gsLosDiscard)  {
```

```
    hasGs = FALSE
  } else {
    hasGs = (NOT gsSolution.equals(ZERO) AND criteriaEverSatisfied)
  }
  gsOnly = gsSolution.gs()
  return (hasGs,gsOnly)
}
```

In the instantaneous (i.e., non-kinematic) version, the value of `sAtTm` is set to `s` and remains constant throughout the for loop. The algorithm searches for a maximally repulsive solution that satisfies the criteria. If divergence is reached, then the search continues until a solution free of secondary conflicts is found (i.e. `lastSolution`). If the additional search is unfruitful, then the algorithm reverts back to the first solution that was divergent (i.e. `firstSolution`). An additional test is applied that eliminates the ground speed solution when the projected distance at the closest point of approach (tau) is smaller than the parameter `gsLosDiscard`. The search is also short-circuited if a LoS occurs or the minimum separation becomes less than `gsLosDiscard`. There are some cases where there are no repulsive ground speeds available in the search direction. Therefore a flag `criteriaEverSatisfied` informs `hasGs`. The case where the criteria is satisfied over the entire search range and divergence is not reached must also be covered. In this case, `firstSolution` will still be `ZERO`. We use the next-to-last value searched, i.e., `prevVo'` as the solution, because the last value may be outside of the `(minGs,maxGs)` range.

7.5 Vertical Speed LoS Algorithm

The vertical LoS solver uses an inner collision region, a cylinder defined by a radius, `caD`, equal to 1000 ft and a half-height, `caH`, equal to 200 ft. The iterative search varies the vertical speed until there is no longer a conflict with this inner collision region and the relative exit speed is sufficiently large. The following parameters are used by the ground speed LoS algorithm

`so`	initial ownship position
`vo`	initial ownship velocity
`si`	initial traffic position
`vi`	initial traffic velocity
`mu`	index of the aircraft to be resolved against, usually the most urgent aircraft
`kinematic`	flag which determines whether a kinematic or an instantaneous solution is desired
`step`	step size for search
`epsv`	The vertical epsilon value calculated by the criteria

The vertical LoS algorithm is defined as follows:

```
vsLoS(so,si,vo,vi,kinematic,step,epsv) : (boolean,double) {
  boolean solFound = FALSE
```

```
boolean innerConf = FALSE
vsSolution = ZERO
int nDetKin = 0
Vect3 firstSoln = ZERO
for (vsDelta = step/2; TRUE; vsDelta = vsDelta + step) {
  double nvoVs = vo.vs() + dir * vsDelta
  if ((nvoVs > maxVs) OR (nvoVs < minVs)) break
  double tm = 0.0
  Vect3 soAtTm = so
  Vect3 siAtTm = si
  if (kinematic) {
    tm = |nvoVs-voVs| / vsAccel
    soAtTm = vsAccel(so ,vo,tm,dir * vsAccel)
    siAtTm = si.linear(vi,tm)
  }
  Vect3 sAtTm = soAtTm - siAtTm
  vo' = vo.mkVs(nvoVs)
  double algInnerFactor = 2.0
  boolean innerLos = CD3D.lossOfSep(soAtTm,siAtTm,caD,
                                    algInnerFactor * caH)
  if (innerLos) break
  innerConf = CD3D.cd3d(sAtTm,vo',vi,caD,algInnerFactor*caH)
  solFound = (epsv*(nvoVs - vi.z) >= minRelVs AND NOT innerConf)
  if (checkSecondary) {
    nDetKin = nDetectorFut(soAtTm,vo',tm,mu)
    if (nDetKin == 2) {
      solFound = FALSE
      break
    }
  }
  if (solFound AND firstSoln == ZERO) firstSoln = vo'
  solFound = (solFound AND nDetKin <= 0)
  if (solFound) break
}
if (solFound) {  // solution free of secondary conflicts found
  vsSolution =  vo'
  hasVs = TRUE
} else {
   if (firstSoln != ZERO) {
     vsSolution = firstSoln
     hasVs = TRUE
   } else {
     hasVs = FALSE
   }
}
```

```
  return (hasVs,vsSolution.vs())
}
```

As noted earlier, in the instantaneous (i.e., non-kinematic) version, the value of `sAtTm` is set to `s` and remains constant throughout the for loop. The algorithm searches in the repulsive direction until there is no conflict with the inner collision region and the relative exit speed is sufficiently large. The vertical LoS criteria is simpler in that it is satisfied for all vertical speeds in the appropriate direction so a specific test is not needed (i.e. it is implicitly satisfied). The search continues beyond this point if there are secondary conflicts in hope of reaching a solution free of secondary conflicts. If the additional search is unfruitful, then the algorithm reverts to the first solution that was found (i.e. `firstSoln`). The search is also short-circuited if a loss of separation occurs. The criteria variable `epsv` is used in the test `epsv*(nvoVs - vi.z) >= minRelVs` as a fast absolute value (i.e. `|nvoVs - vi.z| >= minRelVs`).

8 Summary

Nine Chorus resolution algorithms have been defined that provide instantaneous and kinematic resolutions for both conflict and loss of separation situations. These algorithms have been documented using a pseudocode that clarifies the essential features of the algorithms. The rationale and key design decisions used in these algorithms are highlighted and discussed.

References

1. Ricky Butler and César Muñoz. A formal framework for the analysis of algorithms that recover from loss of separation. Technical Memorandum NASA/TM-2008-215356, NASA, Langley Research Center, Hampton VA 23681-2199, USA, October 2008.

2. Ricky Butler and César Muñoz. Formally verified practical algorithms for recovery from loss of separation. Technical Memorandum NASA/TM-2009-215726, NASA, Langley Research Center, Hampton VA 23681-2199, USA, June 2009.

3. María Consiglio, James Chamberlain, César Muñoz, and Keith Hoffler. Concept of integration for UAS operations in the NAS. In *Proceedings of 28th International Congress of the Aeronautical Sciences, ICAS 2012*, Brisbane, Australia, 2012.

4. James Kuchar and Lee Yang. A review of conflict detection and resolution modeling methods. *IEEE Transactions on Intelligent Transportation Systems*, 1(4):179–189, December 2000.

5. Jeffrey Maddalon, Ricky Butler, Alfons Geser, and César Muñoz. Formal verification of a conflict resolution and recovery algorithm. Technical Paper NASA/TP-2004-213015, NASA, April 2004.

6. César Muñoz, Ricky Butler, Anthony Narkawicz, Jeffrey Maddalon, and George Hagen. A criteria standard for conflict resolution: A vision for guaranteeing the safety of self-separation in NextGen. Technical Memorandum NASA/TM-2010-216862, NASA, Langley Research Center, Hampton VA 23681-2199, USA, October 2010.

7. César Muñoz and Anthony Narkawicz. Time of closest approach in three-dimensional airspace. Technical Memorandum NASA/TM-2010-216857, NASA, Langley Research Center, Hampton VA 23681-2199, USA, October 2010.

8. Anthony Narkawicz and César Muñoz. State-based implicit coordination and applications. Technical Publication NASA/TP-2011-217067, NASA, Langley Research Center, Hampton VA 23681-2199, USA, March 2011.

9. 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.

10. David Wing, Thomas Prevot, Timothy Lewis, Lynne Martin, Sally Johnson, Christopher Cabrall, Sean Commo, Jeffrey Homola, Manasi Sheth-Chandra, Joey Mercer, and Susan Morey. Pilot and controller evaluations of separation function allocation in air traffic management. In *Proceedings of the Tenth USA/Europe Air Traffic Management Research and Development Seminar (ATM2013)*, June 2013.

11. David J. Wing and William Cotton. Autonomous flight rules: A concept for self-separation in U.S. domestic airspace. Technical Report NASA/TP-2011-217174, NASA Langley Research Center, 2011.

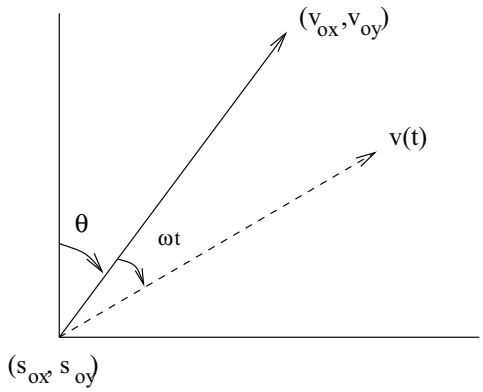

Figure A1. Kinematic Trajectory for Turns

Appendix A

Kinematic Trajectory Generation

We formulate trajectory generation for turns as the computation of the future position and velocity of the aircraft, $\mathbf{s}(t)$ and $\mathbf{v}(t)$, given a constant turn rate (angular velocity), ω, and a time t. The initial position of the aircraft is $\mathbf{s}_o = (s_{ox}, s_{oy})$ and the initial velocity vector is $\mathbf{v}_o = (v_{ox}, v_{oy})$. We let v be the speed of the aircraft, which equals $|\mathbf{v}_o|$, and θ is the track angle of the initial velocity vector. In aircraft navigation, track angles and headings are measured from true north in the clockwise direction, thus $\theta = \text{atan}(v_{ox}, v_{oy})$. Using these definitions, the initial velocity is

$$\begin{aligned} v_{ox} &= v \sin\theta \\ v_{oy} &= v \cos\theta \end{aligned} \tag{A1}$$

By assumption, the aircraft's new target velocity vector will be achieved by a constant angular velocity ω (i.e. $\omega = \frac{d\theta}{dt}$). This will result in time based velocity vector of $\mathbf{v}(t) = (v_x(t), v_y(t))$, where

$$\begin{aligned} v_x(t) &= v \sin(\theta + \omega t) \\ v_y(t) &= v \cos(\theta + \omega t) \end{aligned} \tag{A2}$$

These relationships are shown in figure A1.

Integrating, we obtain the trajectory of positions:

$$\begin{aligned} s_x(t) - s_x(0) &= -\frac{v}{\omega} \cos(\theta + \omega t) \,|_0^t = \frac{v}{\omega}[\cos\theta - \cos(\theta + \omega t)] \\ s_y(t) - s_y(0) &= \frac{v}{\omega} \sin(\theta + \omega t) \,|_0^t = -\frac{v}{\omega}[\sin\theta - \sin(\theta + \omega t)] \end{aligned} \tag{A3}$$

Collecting terms yields:

$$\begin{aligned} s_x(t) &= s_x(0) + \frac{v}{\omega}[\cos\theta - \cos(\theta + \omega t)] \\ s_y(t) &= s_y(0) - \frac{v}{\omega}[\sin\theta - \sin(\theta + \omega t)] \end{aligned} \tag{A4}$$

This is directly coded into the function `turnOmega` which takes the following parameters:

`Vect3 s0`	starting position
`Vect3 v0`	initial velocity
`t`	time of turn
`omega`	rate of change of track, sign indicates direction

The `turnOmega` function is defined:

```
turnOmega(s0,v0,t,omega) : (Vect3, Vect3) {
  if (omega ~= 0) return (s0 + v0 * t,v0)
  v = v0.gs()
  theta = v0.trk()
  xT = s0.x + (v/omega) * (cos(theta) - cos(omega*t+theta))
  yT = s0.y - (v/omega) * (sin(theta) - sin(omega*t+theta))
  zT = s0.z + v0.z*t
  s' = (xT,yT,zT)
  v' = v0.mkTrk(v0.track() + omega * t)
  return (s',v')
}
```

Note, the symbol "`~=`" should be read as "almost equals," which means that the values are compared and if they are within a fixed floating point precision of each other, then they are considered to be equal.

Instead of a turn rate, ω, turns are often specified with a bank angle. Thus, a means is needed to compute ω from a bank angle, usually represented as ϕ. First, consider the equation for the turn radius of aircraft turning in level flight with bank angle ϕ:

$$R = \frac{v^2}{g \tan\phi} \tag{A5}$$

where R is the turn radius, and g is gravitational acceleration. This formula relies on the assumption that the wing provides most of the lift force. For high bank angles, $\phi > 45$, where the body of the aircraft starts to provide lift, this equation is no longer valid. Continuing the development of an equation for ω, we use the relationship $v = |\omega|R$ to obtain

$$\omega = \frac{g \tan\phi}{v} \tag{A6}$$

This equation is captured in the function `turnRate`.

```
turnRate(speed,bankAngle) : double {
  if (bankAngle ~= 0.0) return 0.0
  return g * tan(bankAngle) / speed
}
```

The function `turnTime` computes the time it takes to complete a turn given a ground speed, the magnitude of the track change and the maximum bank angle:

```
turnTime(groundSpeed,deltaTrack,bankAngle) : double {
  omega = turnRate(groundSpeed,bankAngle)
  if (omega == 0.0) return MAXDOUBLE
    return |deltaTrack / omega|
}
```

The functions `gsAccel` and `vsAccel` are used to compute the trajectory for a constant ground speed acceleration and a constant vertical speed acceleration, respectively.

```
 gsAccel(so3,vo3,t,a) : (Vect3, Vect3) {
   Vect2 so = so3.vect2()
   Vect2 vo = vo3.vect2()
   Vect2 sK = so.Add(vo.Hat().Scal(vo.norm() * t + 0.5 * a * t * t))
   double nz = so3.z + vo3.z * t
   Vect3 nso = new Vect3(sK,nz)
   double nvoGs = vo3.gs() + a * t
   Vect3 nvo = vo3.mkGs(nvoGs)
   return (nso,nvo)
 }

vsAccel(so3,vo3,t,a) : (Vect3, Vect3) {
  nvoVs = vo3.vs() + a * t
  Vect3 nvo = vo3.mkVs(nvoVs)
  Vect3 nso = (so3.x + t * vo3.x,
               so3.y + t * vo3.y,
               so3.z + vo3.z * t + 0.5 * a * t * t)
  return (nso,nvo)
}
```

Appendix B

The CDSS Conflict Probe

The `CDSS.conflict` function takes the following parameters:

`s`	the relative position of the aircraft
`vo`	the ownship's velocity
`vi`	the intruder's velocity
`D`	the minimum horizontal distance
`H`	the minimum vertical distance
`B`	the the lower bound of the lookahead time (the value 0 is is implicitly passed in the pseudocode calls)
`T`	the upper bound of the lookahead time ($T < 0$ means infinite lookahead time)

It is defined as follows:

```
conflict(s,vo,vi,D,H,B,T) : boolean {
  if (T >= 0 AND B >= T) return FALSE
  Vect2 s2  = s.vect2()
  Vect2 vo2 = vo.vect2()
  Vect2 vi2 = vi.vect2()
  if (vo.z ~= vi.z) AND |s.z| < H) {
    return CD2D.cd2d(s.vect2(),vo2,vi2,D,B,T)
  }
  vz = vo.z - vi.z
  m1 = max(-H - sign(vz) * s.z,B * |vz|)
  if (T < 0) {
    m2 = H - sign(vz) * s.z
  } else {
    m2 = min(H - sign(vz) * s.z, T * |vz|)
  }
  if ( NOT (vo.z ~= vi.z) AND m1 < m2) {
    return CD2D.cd2d(|vz|s2,vo2,vi2,D * |vz|,m1,m2)
  } else {
    return FALSE
  }
}
```

where `cd2d` is defined as follows

```
cd2d(s,vo,vi,D,B,T) : boolean {
  if (T < 0) {
    v = vo - vi
    return almost_horizontal_los(s,D) OR Delta(s,v,D) > 0 AND s * v < 0
```

```
    }
    if (B >= T) return FALSE
    v = vo - vi
    return almost_horizontal_los(s+Bv,D) OR omega_vv(s,v,D,B,T) < 0
```

and `almost_horizontal_los(s,D)` is defined as

```
  almost_horizontal_los(s,D) : boolean {
    return NOT (s*s ~= D * D) AND s * s < D * D
  }
```

and `omega_vv(s,v,D,B,T)` is defined as follows:

```
   omega_vv(s,v,D,B,T) : double {
     if (s*s ~= D * D) AND B ~= 0) {
       return s*v
     } else {
       tau = min(max(B * v * v,-(s * v)),T * v * v)
     }
     return v*v*s*s + (2*tau)*(s*v) + tau*tau - D*D*v*v
   }
```

and `Delta(s,v,D)` and `det(s,v)` are defined as

```
  Delta(s,v,D) : double {
    return D * D * V * V - det(s,v) * det(s,v)
  }

  det(s,v) : double {
    return s.x * v.y - s.y * v.x
  }
```

In the above functions, the notation `~=` is used for approximately equal: equal within a specified floating point precision.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
01- 08 - 2013	Technical Memorandum	

4. TITLE AND SUBTITLE

The Chorus Conflict and Loss of Separation Resolution Algorithms

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Butler, Ricky W.; Hagen, George E.; Maddalon, Jeffrey M.

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

411931.02.02.07.13.01

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

NASA Langley Research Center
Hampton, VA 23681-2199

8. PERFORMING ORGANIZATION REPORT NUMBER

L-20284

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSOR/MONITOR'S ACRONYM(S)

NASA

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

NASA/TM-2013-218030

12. DISTRIBUTION/AVAILABILITY STATEMENT

Unclassified - Unlimited
Subject Category 63
Availability: NASA CASI (443) 757-5802

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The Chorus software is designed to investigate near-term, tactical conflict and loss of separation detection and resolution concepts for air traffic management. This software is currently being used in two different problem domains: en-route self- separation and sense and avoid for unmanned aircraft systems. This paper describes the core resolution algorithms that are part of Chorus. The combination of several features of the Chorus program distinguish this software from other approaches to conflict and loss of separation resolution. First, the program stores a history of state information over time which enables it to handle communication dropouts and take advantage of previous input data. Second, the underlying conflict algorithms find resolutions that solve the most urgent conflict, but also seek to prevent secondary conflicts with the other aircraft. Third, if the program is run on multiple aircraft, and the two aircraft maneuver at the same time, the result will be implicitly co- ordinated. This implicit coordination property is established by ensuring that a resolution produced by Chorus will comply with a mathematically-defined criteria whose correctness has been formally verified. Fourth, the program produces both instantaneous solutions and kinematic solutions, which are based on simple accel- eration models. Finally, the program provides resolutions for recovery from loss of separation. Different versions of this software are implemented as Java and C++ software programs, respectively.

15. SUBJECT TERMS

Air traffic management; Algorithms; Conflict detection and Resolution; Loss of separation

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	37	(443) 757-5802