Application Programming Interface of DAIDALUS* (Release V-0.99)

Working Document

Abstract

This document describes the Application Programming Interface (API) of DAIDALUS, a reference implementation of a detect and avoid (DAA) concept for Unmanned Aircraft Systems. DAIDALUS is developed by the Formal Methods Group at NASA Langley Research Center and the code is available under NASA's Open Source Agreement (NOSA).

*The title graphic was designed by NASA Engineer Mahyar R. Malekpour.

Contents

1 Introduction

DAIDALUS (**D**etect & **A**vo**id** **A**lerting **L**ogic for **U**nmanned **S**ystems) is a reference implementation of a detect and avoid (DAA) concept for Unmanned Aircraft Systems.[1] At the core of this DAA concept, there is a mathematical definition of the well-clear concept. Two aircraft are considered to be *well clear* of each other if appropriate distance and time variables determined by the relative aircraft states remain outside a set of predefined threshold values. These distance and time variables are closely related to variables used in the Resolution Advisory (RA) logic of the Traffic Alert and Collision Avoidance System Version II (TCAS II). A particular set of threshold values determines a corresponding volume around the aircraft, called the violation volume. For some choice of threshold values, the TCAS II RA volume is strictly contained within the well-clear violation volume. That is, aircraft are declared to be in well-clear violation before an RA is issued. The well-clear definition also satisfies the property of *symmetry*, i.e., in a pair-wise scenario, both aircraft make the same determination of being well-clear or not, and the property of *local-convexity*, i.e., in a non-maneuvering pair-wise scenario, there is at most one time interval in which the aircraft are not well clear.

DAIDALUS includes algorithms for determining the current well-clear status between two aircraft and for predicting a well-clear violation within a lookahead time, assuming non-maneuvering trajectories. In the case of a predicted well-clear violation, DAIDALUS also provides an algorithm that computes the time interval of well-clear violation. Furthermore, DAIDALUS implements algorithms for computing prevention bands, assuming a simple kinematic trajectory model. Prevention bands are ranges of track, ground speed, and vertical speed maneuvers that are predicted to be in well-clear violation within a given lookahead time. These bands provide awareness information to remote pilots and assist them in avoiding certain areas in the airspace. When aircraft are not well clear, or when a well-clear violation is unavoidable, the DAIDALUS prevention bands algorithms compute well-clear recovery bands. Recovery bands are ranges of horizontal and vertical maneuvers that assist pilots in regaining well-clear status within the minimum possible time. Recovery bands are designed so that they do not conflict with resolution advisory maneuvers generated by systems such as TCAS II.

Finally, DAIDALUS implements two alternative alerting schemas. One schema is based on the prediction of well-clear violations for different sets of increasingly conservative threshold values. The second schema is based on the types of bands, which can be either preventive or corrective, computed for a single set of threshold values. A band is preventive if it does not include the current trajectory. Otherwise, it is corrective. Recovery bands, by definition, are always corrective. In general, both schemas yield alert levels that increase

[1] The acronym DAIDALUS is a reference to the craftsman Daedalus of Greek mythology. Daedalus made wings for himself and for Icarus and warned Icarus not to fly too high, because the heat of the sun would melt the wax, nor too low, because the sea foam would soak the feathers.

Class/Interface	Package
Interval	Util
LatLonAlt	Util
Plan	Util
Position	Util
Units	Util
Velocity	Util
BandsRegion	ACCoRD
CD3DTable	ACCoRD
CDCylinder	ACCoRD
ConflictData	ACCoRD
Daidalus	ACCoRD
DaidalusParameters	ACCoRD
DefaultDaidalusParameters	ACCoRD
Detection3D	ACCoRD
KinematicBands	ACCoRD
TCASTable	ACCoRD
TCAS3D	ACCoRD
WCVTable	ACCoRD
WCV_TAUMOD	ACCoRD

Table 1: Classes in DAIDALUS Interface

in severity as a potential pair-wise conflict scenario evolves.

All algorithms included in DAIDALUS are implemented in C++ and Java and the code is available under NASA's Open Source Agreement. The implementations are modular and highly configurable via the DAIDALUS application programming interface (API). The DAIDALUS algorithms have been formally specified in a mathematical notation and verified for correctness in a theorem prover. The software implementations have been validated against the formal models using randomly generated unit test cases. DAIDALUS is under consideration for inclusion in the appendix of RTCA Special Committee 228 Minimum Operational Performance Standards (MOPS) for Unmanned Aircraft Systems.

2 Preliminaries

DAIDALUS consists of two packages in the hierarchy `gov.nasa.larcfm`: `Units` and `ACCoRD`. This document will refer to classes in these packages through unqualified names. Table 2 lists the packages for the classes and interfaces used in this document.

In DAIDALUS's API, every method that sets or gets a value provides a String argument, where the units are explicitly specified. Strings representing standard symbols for SI units are accepted. As a special case, time is always specified in seconds. Furthermore, northern latitudes and eastern longitudes are positive. Angles representing aircraft track or heading are specified in true

Units	String Symbols
Time	
seconds	`"s"`
Distance	
nautical miles	`"nmi"`
meters	`"m"`
kilometers	`"km"`
feet	`"ft"`
Speed	
knots	`"knot"`, `"kn"`
meters per second	`"m/s"`
feet/minute	`"fpm"`
Acceleration	
meters per second^2	`"m/s^2"`
1G = 9.80665$\frac{m}{s^2}$	`"G"`
Angular Distance	
degrees	`"deg"`
radians	`"rad"`
Angular Speed	
degrees per second	`"deg/s"`
radians per second	`"rad/s"`

Table 2: Typical Units Used in DAIDALUS Interface

north clockwise convention. Table 2 provides a list of symbols typically used in air traffic management.

3 The Class `Daidalus`

The main functional features in DAIDALUS are provided through the class `Daidalus`. The invokation

```
Daidalus daa = new Daidalus();
```

assigns to the variable `daa` a new object of the class `Daidalus` that is initialized to default values, including a well-clear concept based on TAUMOD with the following threshold values DMOD=HMD=4000 ft, TAUMOD=35 s, ZTHR=450 ft. This set of values can be dynamically changed. Furthermore, as explained later, all functional features provided by the class `Daidalus` are available for a larget set of alternative self-separation concepts and different sets of threshold values.

DAIDALUS supports a large set of configurable parameters. These parameters govern the behavior of the kinematic trajectory model used by the DAIDALUS algorithms. These parameters described in Section 8.

3.1 Positions and Velocities

Several methods in DAIDALUS involve 3-D positions and velocities. The classes `Position` and `Velocity` provide interfaces for creating these types of objects. If `p` is a variable of type `Position`, an object of that type can be created by using the factory method invokation

```
p = Position.makeLatLonAlt(lat,lat_u,lon,lon_u,alt,alt_u);
```

where `lat`, `lon`, and `alt` are the latitude, longitude, and altitude given in `lat_u`, `lon_u`, and `alt_u` units, respectively. If `v` is a variable of type `Velocity`, an object of that class can be created using the factory method invokation

```
v = Velocity.makeTrkGsVs(trk,trk_u,gs,gs_u,vs,vs_u);
```

where `trk`, `gs`, and `vs` are the direction of the velocity vector, in true north clockwise convention, the norm of the horizontal speed, and the vertical speed given in `trk_u`, `gs_`, and `vs_u` units, respectively.

3.2 Aircraft State Information

An object `daa` of type `Daidalus` maintains a list of aircraft state information. The ownship state has to be set before any traffic aircraft state. This can be done through the invocation

$$\texttt{daa.setOwnshipState(id}_0\texttt{,s}_0\texttt{,v}_0\texttt{,t}_0\texttt{);}$$

where $\texttt{id}_0$ is the ownship identifier (string), $\texttt{t}_0$ is the current time, and $\texttt{s}_0$, $\texttt{v}_0$ are the ownship's position and ground velocity at time $\texttt{t}_0$.

Traffic information can be added through the invokation

$$\texttt{daa.addTrafficState(id}_i\texttt{,s}_i\texttt{,v}_i\texttt{,t}_i\texttt{);}$$

where $\texttt{id}_i$ is the i-th traffic's identifier, $\texttt{t}_i$ is a time, and $\texttt{s}_i$ and $\texttt{v}_i$ are the position and ground velocity of the i-th traffic aircraft at time $\texttt{t}_i$. The time parameter $\texttt{t}_i$ is optional. If it is not provided the value $\texttt{t}_0$, i.e., the current time, is used by default. If it is provided, but its value is different from $\texttt{t}_0$, the position of the intruder is linearly projected to $\texttt{t}_0$ so that the ownship's and intruder's states are synchronized in time.

The number of aircraft in the list of aircraft can be obtained with the method call `daa.numberOfAircraft()`. Methods in `Daidalus` often refer to aircraft by their index in this list. The method `daa.addTrafficState` return the index of the aircraft after it has been added to the list. Furthermore, the method `aircraftIndex(id)` of the class `Daidalus` returns the index of the aircraft identified by the string `id`. It returns a negative value if the list of aircraft does not include an aircraft identified by `id`.

3.3 Wind Field

A wind vector can be indicated to DAIDALUS as follows

```
daa.setWindField(wind);
```

where `wind` is an object of type `Velocity`, whose vertical component is assumed to be 0. This wind vector is assumed to be the same for all aircraft.

3.4 Resetting Aircraft Information

The invokation

```
daa.rest();
```

removes all aircraft from the `Daidalus` object `daa` and clear the wind vector.

4 State-Based Detectors

DAIDALUS provides implementations of different types of state-based detection algorithms. Each one of these types is referred to as a *(state-based) detector*. A detector class implements the interface `Detection3D`. This interface provides methods for checking loss of separation, predicting conflicts within a time lookahead interval, and computing several time and distance variables. All functionality provided by DAIDALUS, such as conflict detection, alerting, and preventions bands, is available for any detector.

The following detectors are implemented in DAIDALUS.

- `WCV_TAUMOD`: Violation is defined as loss of well-clear with respect to the TCAS II time variable called modified tau.
- `WCV_TCPA`: Violation is defined as loss of well-clear with respect to time to closest point of approach.
- `WCV_TEP`: Violation is defined as loss of well-clear with respect to time to entry point.
- `CDCylinder`: Violation is defined as loss of horizontal and vertical separation.
- `TCAS3D`: Violation is defined according to the core alerting logic of TCAS II.

When an object `daa` of type `Daidalus` is created, that object is set to use by default a detector of type `WCV_TAUMOD`. If `detector` is an object of type `Detection3D`, the invokation `daa.setDetector(detector)` assigns the detector `detector` to the object `daa`.

Different detector classes depend on different distance and time thresholds. The values of these threshold are configurable. The rest of this sections illustrates how a detector object can be created and configured.

4.1 Well-Clear Detectors

The classes `WCV_TAUMOD`, `WCV_TCPA`, and `WCV_TEP` implement detectors for well-clear concepts based on modified tau, time to closest point of approach, and time to entry point, respectively. These detectors depend on the following distance and time thresholds.

- DTHR: Horizontal distance threshold. Default value is 4000 ft.
- ZTHR: Vertical distance threshold. Default value is 450 ft.
- TTHR: Horizontal time threshold. Default value is 35 s.
- TCOA: Vertical time threshold. Default value is 0 s.

The constructors `WCV_TAUMOD()`, `WCV_TCPA()`, and `WCV_TEP()` initializes the thresholds DTHR, ZTHR, TTHR, and TCOA to their default values. The threshold values of an object of one of the types `WCV_TAUMOD`, `WCV_TCPA`, `WCV_TEP` can be modified and read using the following methods.

- `set_DTHR(val,u)`, `get_DTHR(u)`: Modify the value of DTHR to `val` in specified units `u` and read value of DTHR in specified units `u`, respectively.
- `set_ZTHR(val,u)`, `get_ZTHR(u)`: Modify the value of ZTHR to `val` in specified units `u` and read value of ZTHR in specified units `u`, respectively.
- `set_TTHR(val)`, `get_TTHR()`: Modify the value of TTHR to `val` in seconds and read value of TTHR in seconds, respectively.
- `set_TCOA(val)`, `get_TCOA()`: Modify the value of TCOA to `val` in seconds and read value of TCOA in seconds, respectively.

The thresholds DTHR, ZTHR, TTHR, and TCOA used by the well-clear family of detectors are stored in a data structure represented by the class `WCVTable`. The constructor `WCVTable()` initializes these thresholds to their default values. The threshold values stored in an object of type `WCVTable` can be modified and read using setters and getters methods similar to the ones provided by the classes `WCV_TAUMOD`, `WCV_TCPA`, and `WCV_TEP`. Tables with the values of the experimental well-clear concepts developed by NASA and MITLL can be created using the factory methods `WCVTable.NASA()` and `WCVTable.MITLL()`, respectively.

Instances of the classes `WCV_TAUMOD`, `WCV_TCPA`, and `WCV_TEP` initialized to the threshold values stored in an object `wcvtable` of type `WCVTable` can be created as follows.

```
Detection3D wcv_taumod = new WCV_TAUMOD(wcvtable);
Detection3D wcv_tcpa = new WCV_TCPA(wcvtable);
Detection3D wcv_tep = new WCV_TEP(wcvtable);
```

If wcv is an object of one of the classes WCV_TAUMOD, WCV_TCPA, and WCV_TEP, the method call wcv.setWCVTable(wcvtable) copies the threshold values stored in wcvtable into the table used by the detector wcv. Furthermore, the method call wcv.getWCVTable() returns a reference to the thresholds table object of wcv.

4.2 Distance-Based Detector

The detector CDCylinder implements a detection logic for a cylindrical protected area. This detector depends on a horizontal distance threshold, whose default value is 5 nmi, and a vertical distance threshold, whose default value is 1000 ft. These default values are used when an instance of the class CDCylinder is constructed without parameters. An object of type CDCylinder for detecting violations to a minimum horizontal separation D, in u units, and a minimum vertical separation H, in v units, can be created as follows.

```
Detection3D cd3d = new CDCylinder(D,u,H,v);
```

The distance thresholds of an object of type CDCylinder can be modified and read using the following methods.

- setHorizontalSeparation(val,u), getHorizontalSeparation(u): Modify the value of the minimum horizontal separation to val in specified units u and read value of minimum horizontal separation in specified units u, respectively.
- setVerticalSeparation(val,u), getVerticalSeparation(u): Modify the value of the minimum vertical separation to val in specified units u and read value of minimum vertical separation in specified units u, respectively.

The distance thresholds used by the class CDCylinder are stored in a table of type CD3DTable. The constructor CD3DTable() creates an object where these thresholds are initialized to default values. The threshold values stored in an object of type CD3DTable can be modified and read using methods similar to the ones provided by the class CDCylinder.

An instance of class CDCylinder initialized to the threshold values stored in an object cd3dtable of type CD3DTable can be created as follows.

```
Detection3D cd3d = new CDCylinder(cd3dtable);
```

The method call cd3d.setCD3DTable(wcvtable) copies the threshold values stored in cd3dtable into the table used by the detector cd3d. Furthermore, the method call cd3d.getCD3DTable() returns a reference to the thresholds table object of cd3d.

4.3 TCAS II Detectors

Detector for the TCAS II Resolution Advisory (RA) and Traffic Alert (TA) logic can be created by using the code

```
Detection3D tcas3d = new TCAS3D(tcastable);
```

where `tcastable` is an instance of the class `TCASTable`. This class stores the TCAS II thresholds TAU (time), DMOD (horizontal distance), ZTHR (vertical distance), and HMD (horizontal distance).

An object `ra` of type `TCASTable` that contains the threshold values of the TCAS II Resolution Advisory (RA) logic can be created through

```
TCASTable ra = new TCASTable();
```

or, alternatively,

```
TCASTable ra = new TCASTable(true);
```

An object `ta` of type `TCASTable` that contains the threshold values of the TCAS II Traffic Alert (TA) logic can be created as follows.

```
TCASTable ta = new TCASTable(false);
```

The TCAS II table is indexed by sensitivity levels according to the altitude of the aircraft. Sensitivity levels are integers between 2 and 8. The static method `TCASTable.getSensitivityLevel(val,u)` returns the sensitivity level for altitude `val`, given in u units.

Methods are provided for reading and setting the threshold values stored for any sensitivity level stored in an object of type `TCASTable`. In particular, the method invokations `tcastable.getTAU(sl)` and `tcastable.setTAU(sl,val)` read and set the threshold value of TAU, where the units are specified in seconds. The methods `tcastable.getDMOD(sl,u)`, `tcastable.getZTHR(sl,u)`, and `tcastable.getHMD(sl,u)` return, respectively, the threshold values DMOD, ZTHR, and HMD, in u units, stored in `tcastable` for a given sensitivity level `sl`. Furthermore, the method invokations `tcastable.setDMOD(sl,val,u)`, `tcastable.setZTHR(sl,val,u)`, and `tcastable.setHMD(sl,val,u)` set, respectively, the threshold values DMOD, ZTHR, and HMDF of `tcastable` to `val`, given in u units.

The TCAS II RA logic includes a miss-distance filter, which uses the distance threshold HMD. The filter can be set OFF/ON for a particular table `tcastable` by using the method invokation `tcastable.setHMDFilter(b)`, where `b` is a `boolean` value representing the values ON (`true`) and OFF (`false`). The call `tcastable.getHMDFilter()` returns the status of the HMD filter in the object `tcastable`.

At any moment, the threshold values stored in an object `tcastable` can be reset to the default values of the TCAS II detection logic by using the method invokation `tcastable.setDefaultRAThresholds(b)`, where `b` is a `boolean` value. If `b` is `true`, the threshold values in `tcastable` are set to the RA logic. Otherwise, they are set to the values of the TA logic.

If no arguments are given to the constructor `TCAS3D()`, the detector will be initiated with the default threshold values of the TCAS II RA logic.

5 Conflict Detection Logic

Time to violation between the ownship and a traffic aircraft at index `ac`, with respect to the core detector object in the `Daidalus` object `daa`, can be computed as follows.

```
double t2v = daa.timeToViolation(ac);
```

If `t2v` is negative, the aircraft are not in conflict within the interval [t_0,t_0+T], where t_0 is the current time and T is the lookahead time. If `t2v` is zero, the aircraft are in violation at current time. If `t2v` is positive, its value represents a time, in seconds, indicating that the aircraft are in conflict within lookahead time T, where first time to violation will occur in `t2v` seconds.

It is possible to compute time to violation relative to an arbitrary time t. The invokation

```
double t2v = daa.timeToViolationAt(ac,t);
```

linearly projects the aircraft states to time `t` and computes the time to violation relative to the time interval [`t`,`t`+T],

The method `daa.timeIntervalOfViolation(ac)` computes an object of type `ConflictData` that represents the time interval of violation between the ownship and the aircraft at index `ac`. If `det` is an object of type `ConflictData`, the methods call `det.getTimeIn()` and `det.getTimeOut()` return times, in seconds, that represent the interval of violation. The call `det.conflict()` returns true if this interval is non-empty.

The method `daa.timeIntervalOfViolationAt(ac,t)` projects the ownship and aircraft at index `ac` to time `t` and computes the time interval of violation relative to `t`.

6 Alerting Logic

DAIDALUS supports two different alerting logics. The first type is referred to as "thresholds-based" and corresponds to the alerting schema used in NASA's TP5 experiment. The second type is referred to as "bands-based" and corresponds to the alerting schema to be used in NASA's CASSAT experiment. By default the alerting schema used by DAIDALUS is bands-based. The method call `daa.setAlertingLogic(b)`, where `b` is a Boolean value, sets a particular alerting logic to the Daidalus object `daa`. If `b` is `true` the alerting logic is set to the bands-based schema; otherwise, the alerting logic is set to the thresholds-based schema.

The interface to both alerting logics are the same. Given an object `daa` of type `Daidalus`, the alert level between the ownship and the traffic aircraft at index `ac` can be computed as follows

```
int alert_level = daa.alerting(ac);
```

If `alert_level` is negative, `ac` is not a valide index in `daa`. If `alert_level` is zero, no alert is issued for aircraft `ac` at time `t`. Otherwise, `alert_level` is a

positive numerical value. The higher the number the higher the severity of the alert.

In general, the numerical values provided by both alerting logics correspond to proximate alert (number 1), preventive alert (number 2), corrective alert (number 3), and caution/warning (number 4).

Both logics are configurable in terms of distance and time thresholds (see Section 8).

The call `daa.alertingAt(ac,t)` projects the ownship and traffic at index `ac` to time `t` and computes the alerting level at this time.

7 Prevention Bands

The following code creates an object `bands` of type `KinematicBands` for the aircraft in the object `daa`.

```
KinematicBands bands = daa.getKinematicBands();
```

For efficiency reasons, bands are computed in a lazy way, i.e., the method `daa.getKinematicBands` does not compute the bands, but just creates an object of type `KinematicBands`. Bands are actually computed when information about bands is requested. Furthermore, when a wind field is set, the output of the prevention bands algorithm is heading and air speed instead of track and ground speed.

The method call `daa.getKinematicBandsAt(t)` projects the list of aircraft states to time `t` and creates a `KinematicBands` object for those projected states.

7.1 Track Bands

The computation of track bands require a turn rate, which can be set by calling the method `bands.setTurnRate(val,u)`, where `val` is a non-negative angular velocity specified in u units. By default the turn rate is zero and, in this case, the maximum bank angle is assumed.

```
for (int i = 0; i < bands.trackLength(); i++ ) {
  Interval iv = bands.track(i,"deg"); //i-th band region
  double lower_trk = iv.low; //[deg]
  double upper_trk = iv.up;  //[deg]
  BandsRegion regionType = bands.trackRegion(i);
  ...
}
```

7.2 Ground Speed Bands

```
for (int i = 0; i < bands.groundSpeedLength(); i++ ) {
  Interval iv = bands.groundSpeed(i,"knot"); //i-th band region
  double lower_gs = iv.low; //[knot]
```

```
  double upper_gs = iv.up;  //[knot]
  BandsRegion regionType = bands.groundSpeedRegion(i);
  ...
}
```

7.3 Vertical Speed Bands

```
for (int i = 0; i < bands.verticalSpeedLength(); i++ ) {
  Interval iv = bands.verticalSpeed(i,"fpm"); //i-th band region
  double lower_vs = iv.low; //[fpm]
  double upper_vs = iv.up;  //[fpm]
  BandsRegion regionType = bands.verticalSpeedRegion(i);
  ...
}
```

7.4 Altitude Bands

Altitude bands require a climb/descend rate to compute a time for a target altitude. This time can be set by the method `bands.setVerticalRate(val,u)`, where `val` is a vertical non-negative climb/descend rate specified in `u` units. By default the vertical rate is zero and, in this case, instantaneous climb/descend and level off to a target altitude is assumed.

```
for (int i = 0; i < bands.altitudeLength(); i++ ) {
  Interval iv = bands.altitude(i,"ft"); //i-th band region
  double lower_alt = iv.low; //[ft]
  double upper_alt = iv.up;  //[ft]
  BandsRegion regionType = bands.altitudeRegion(i);
  ...
}
```

7.5 Bands Region

The class `BandsRegion` is an enumerated type that identifies the type of a particular band region. The enumeration consists of `NONE`, i.e., "green" bands, `NEAR`, i.e., "red" bands, and `RECOVERY`, i.e., "dashed" bands.

8 Parameters

DAIDALUS supports a large set of parameters. These parameters can be set once for all through the class `DefaultDaidalusParameters` or for each particular `Daidalus` object. The following setter methods, and their corresponding getter methods, are supported by both classes, e.g., `DefaultDaidalusParameters` and `Daidalus`. In the case of `DefaultDaidalusParameters`, the methods are static.

- `setLookaheadTime(val)`: Set lookahead time to `val` in seconds.

- `setAlertingTime(val)`: Set alerting time to `val` in seconds. By default, `val` is set to the same value as lookahead time. The value `val` should be less or equal than lookahead time.
- `setMaxGroundSpeed(val,u)`: Set the maximum horizontal speed to `val` specified in `u` units for ground speed bands.
- `setMaxVerticalSpeed(val u)`: Set the maximum vertical speed to `val` specified in `u` units for vertical speed bands.
- `setMinAltitude(val,u)`: Set the minimum altitude to `val` in `u` units for altitude bands.
- `setMaxAltitude(val,u)`: Set the maximum altitude to `val` in `u` units for altitude bands.
- `setHorizontalAcceleration(val,u)`: Set the maximum horizontal acceleration to `val` specified in `u` units for ground speed bands.
- `setVerticaAcceleration(val,u)`: Set the maximum vertical acceleration to `val` specified in `u` units for vertical speed bands.
- `setBankAngle(val,u)`: Set the maximum bank angle to `val` in `u` units for track bands.
- `setTurnRate(val,u)`: Set the turn rate to `val` in `u` units for track bands.
- `setVerticalRate(val,u)`: Set the vertical rate to `val` in `u` units for altitude bands.
- `setTrackStep(val,u)`: Set track granularity to the value `val`, given in `u` units for track bands.
- `setGroundSpeedStep(val,u)`: Set ground speed granularity to the value `val`, given in `u` units for ground speed bands.
- `setVerticalSpeedStep(val,u)`: Set vertical speed granularity to the value `val`, given in `u` unit for vertical speed bands.
- `setAltitudeStep(val,u)`: Set altitude granularity to the value `val`, given in `u` units for altitude bands.
- `setAlertingLogic(val)`: Set alerting logic to bands-based schema, when `val` is `true`, or thresholds-based schema, when `val` is false.

Parameters can be loaded from a file and saved to a file through the methods `loadFromFile(file)` and `saveToFile(file)`, where `file` is a file name. These methods are supported by both classes, e.g., `DefaultDaidalusParameters` and `Daidalus`. In the case of `DefaultDaidalusParameters`, the methods are static. An example of a parameters file containing the complete list of supported parameters and default values is provided below.

```
# WC Thresholds
DTHR = 4000.0000 [ft] # 1219.2000 [internal]
ZTHR = 450.0000 [ft] # 137.1600 [internal]
TTHR = 35.0000 [s] # 35.0000 [internal]
TCOA = 0.0000 [s] # 0.0000 [internal]
# CD3D Thresholds
D = 5.0000 [nmi] # 9260.0000 [internal]
H = 1000.0000 [ft] # 304.8000 [internal]
# Conflict Bands Parameters
alerting_time = 0.0000 [s] # 0.0000 [internal].
# If alerting_time is set to 0, lookahead_time is used instead
lookahead_time = 180.0000 [s] # 180.0000 [internal]
min_gs = 0.0000 [knot] # 0.0000 [internal]
max_gs = 700.0000 [knot] # 360.1111 [internal]
min_vs = -5000.0000 [fpm] # -25.4000 [internal]
max_vs = 5000.0000 [fpm] # 25.4000 [internal]
min_alt = 500.0000 [ft] # 152.4000 [internal]
max_alt = 50000.0000 [ft] # 15240.0000 [internal]
implicit_bands = false
# Kinematic Bands Parameters
trk_step = 1.0000 [deg] # 0.0175 [internal]
gs_step = 1.0000 [knot] # 0.5144 [internal]
vs_step = 10.0000 [fpm] # 0.0508 [internal]
alt_step = 500.0000 [ft] # 152.4000 [internal]
horizontal_accel = 2.0000 [m/s^2] # 2.0000 [internal]
vertical_accel = 2.0000 [m/s^2] # 2.0000 [internal]
turn_rate = 3.0000 [deg/s] # 0.0524 [internal]
bank_angle = 30.0000 [deg] # 0.5236 [internal].
# bank_angle is only used when turn_rate is set to 0
vertical_rate = 0.0000 [fpm] # 0.0000 [internal]
# Recovery Bands Parameters
recovery_stability_time = 2.0000 [s] # 2.0000 [internal]
max_recovery_time = 0.0000 [s] # 0.0000 [internal].
# If set to 0, lookahead time is used instead
min_horizontal_recovery = 0.0000 [nmi] # 0.0000 [internal].
# If recovery  is set to 0, TCAS RA HMD is used instead
min_vertical_recovery = 0.0000 [ft] # 0.0000 [internal].
# If min_vertical_recovery  isset to 0, TCAS RA ZTHR is used instead
conflict_crit = false
recovery_crit = false
recovery_trk = true
recovery_gs = true
recovery_vs = true
# Alerting
bands_alerting = true
trk_alerting = true
```

```
gs_alerting = false
vs_alerting = true
preventive_alt = 700.0000 [ft] # 213.3600 [internal]
preventive_trk = 10.0000 [deg] # 0.1745 [internal].
# If preventive_trk equals 0, no tracks are preventive.
# If preventive_trk less than 0, all tracks are preventive
preventive_gs = 100.0000 [knot] # 51.4444 [internal].
# If preventive_gs equals 0, no grounds speeds are preventive.
# If preventive_gs less than 0, all ground speeds are preventive
preventive_vs = 500.0000 [fpm] # 2.5400 [internal].
# If preventive_vs equals 0, no vertical speeds are preventive.
# If preventive_vs less than 0, all vertical speeds are preventive
time_to_warning = 15.0000 [s] # 15.0000 [internal]
warning_when_recovery = false
# Other Parameters
ca_bands = false
```