IRENE: AE9/AP9/SPM Radiation Environment Model

Application Programming Interface

Version 1.50.001

The IRENE (International Radiation Environment Near Earth): (AE9/AP9/SPM) model was developed by the Air Force Research Laboratory in partnership with MIT Lincoln Laboratory, Aerospace Corporation, Atmospheric and Environmental Research, Incorporated, Los Alamos National Laboratory and Boston College Institute for Scientific Research.

IRENE (AE9/AP9/SPM) development team: Wm. Robert Johnston[1] (PI), T. Paul O'Brien[2] (PI), Gregory Ginet[3] (PI), Stuart Huston[4], Tim Guild[2], Christopher Roth[4], Yi-Jiun Su[1], Rick Quinn[4], Michael Starks[1], Paul Whelan[4], Reiner Friedel[5], Chad Lindstrom[1], Steve Morley[5], and Dan Madden[6].

To contact the IRENE (AE9/AP9/SPM) development team, email ae9ap9@vdl.afrl.af.mil .

The IRENE (AE9/AP9/SPM) model and related information can be obtained from AFRL's Virtual Distributed Laboratory (VDL) website: https://www.vdl.afrl.af.mil/programs/ae9ap9

V1.00.002 release: 05 September 2012

V1.03.001 release: 26 September 2012

V1.04.001 release: 20 March 2013

V1.04.002 release: 20 June 2013

V1.05.001 release: 06 September 2013

V1.20.001 release: 31 July 2014

V1.20.002 release: 13 March 2015

V1.20.003 release: 15 April 2015

V1.20.004 release: 28 September 2015

V1.30.001 release: 25 January 2016

V1.35.001 release: 03 January 2017

V1.50.001 release: 01 December 2017

[1] Air Force Research Laboratory, Space Vehicles Directorate
[2] Aerospace Corporation
[3] MIT Lincoln Laboratory
[4] Atmospheric and Environmental Research, Incorporated
[5] Los Alamos National Laboratory
[6] Boston College Institute for Scientific Research

IRENE: AE9/AP9/SPM Model Application Programming Interface
Version 1.50.001

Table of Contents

Overview

The IRENE (AE9/AP9/SPM) radiation environment model is distributed with a GUI client application and a command-line driven utility application that can be used to run the model either interactively or through batch-driven processes. For situations in which it is more appropriate to integrate the AE9, AP9 and/or SPM model calls and data usage directly into a new or existing application, an Application Programming Interface (API) is also available.

The IRENE model package supports programmatic access through a suite of APIs accessible from the C++ programming language. The model is written in C++ and direct access to all classes and methods of the model is available using the source distribution of the model, available upon request.

Installation and Setup

The IRENE model package is distributed as a zip file that comes with a pre-built Windows 64- and 32-bit binary distribution of the model and (upon request) a complete set of source code files for building the executables on Linux systems, and for the development of user applications employing the API libraries.

The source distribution build process for the package uses CMake to generate an "out-of-source" build. That means that output executable and library files are written to a separate directory from the source files. This is done to facilitate builds for multiple platforms, as well as debug and release modes. Please refer to the "Build Instructions" document included in this distribution for the configuration settings used in the CMake build process, and the use of third-party libraries: Boost® (for linear algebra functions and data structures) and HDF5® (for internal databases). A development license for the Intel MPI Library commercial product will be required for building of multi-threaded 64-bit Windows applications. A free, *30-day trial* version is available at https://software.intel.com/en-us/intel-mpi-library .

C++ API

The IRENE (AE9/AP9/SPM) model C++ source code and header files are provided with the source distribution (available upon request). This API may be accessed at two different levels: the 'Application' level provides a programmatic interface for performing most of the model processing options available via the 'CmdLineAe9Ap9' application (or its GUI), including parallelization; the 'Model' level provides programmatic access directly to the APIs of each of the underlying models that comprise the AE9/AP9/SPM model package. This gives the client application developer a great deal of freedom in determining at what level and granularity to integrate with the model. The methods of the 'Application' and each of the individual model classes are described in detail in the remainder of this document.

The specification of a directory path and/or filename may include a computer system environment variable, ie '`$AX9DATA/igrfDB.h5`' [Linux style] or '`%AX9DATA%/igrfDB.h5`' [Windows style], provided the environment variable ('`AX9DATA`' in this example) has been previously defined.

Two C++-based sample programs using the API are provided within the distribution files, DemoApp and DemoModel; these demonstrate the API usage at both levels. The source code and associated CMake configuration files are located in the 'demoApp' and 'demoModel' directories, respectively.

[In a future release, APIs in both C and Python API will be added.]

C++ Application-Level API Reference

Application Class

Header file: CApplication.h

This class is the main entry point that provides programmatic access to the model results available from the CmdLineAe9Ap9 application. This API does not directly access the models, but instead relies on the same set of supporting programs used by the CmdLineAe9Ap9 application, and then queries the results following the completion of the model calculation. Client applications requiring direct or synchronous access to the model results should use the 'model-level' API methods instead.
Computer system environment variables may be used when specifying a directory path and/or filename. Please note that all time values, both input and output, are in Modified Julian Date (MJD) form. Utility methods for conversions to and from MJD times are included here. Position coordinates are always used in sets of three values, in the coordinate system and units that are specified. Please consult the User's Guide document, "Supported Coordinate Systems" for more details; in particular, note the order of the coordinate values for non-Cartesian coordinate systems.

General:

Application

Usage: Default constructor
Parameters: -none-
Return values: -none-

~Application

Usage: Destructor
Parameters: -none-
Return values: -none-

The following methods are used for adjusting the internal behavior of the model runs.
The use of the *setExecDir*() method is required; all others in this section are optional.

int setExecDir

(const std::string& strExecDir)
Usage: (***Required***) Specifies the directory path for supporting programs needed for model execution.
Parameters:
strExecDir - path to installation executables, such as CmdLineAe9Ap9(.exe)
Return value: int – 0 = success, otherwise error

int setWorkDir

(const std::string& strWorkDir)
Usage: Specifies the directory path in which a temporary directory, used as a repository for the intermediate binary files generated during model execution, is created. When not specified, this defaults to the current working directory of the client application. Use of an alternate directory may improve model performance. If the work directory is located on a RAID-5 disk unit, use of the *setNumFileIo*() method may further improve performance.

Parameters:

strWorkDir - path to location for temporary directory creation; this working directory will be created if it does not exist.

Return value: int – 0 = success, otherwise error

int setBinDirName

(const std::string& strBinDirName)

Usage: Specifies a non-default name (no path) for the temporary directory containing the intermediate binary files generated during model execution. This may be desired when retaining these files for use with external applications. When this directory name is not specified, a unique name is automatically generated by the 'Application' class.

Parameters:

strBinDirName - name of temporary directory to be created

Return value: int – 0 = success, otherwise error

void setDelBinDir

(bool bVerdict)

Usage: Specifies the disposition of the temporary directory and its intermediate binary files when the 'Application' class object is destroyed or a call is made to the *resetModelRun*() method. When not specified, the default setting is *true*.

Parameters:

bVerdict – set to *true* to remove the directory and files, or *false* to retain them.

Return value: -none-

int setNumProc

(const int& iNumProc)

Usage: Specifies the total number of processors to use for the execution of the model calculations. This number *includes* one processor for the 'master' node. *Must be 3 or greater for parallel processing*. A system call is used to query the number of actual processors, and use this number as a limit. When this query fails or returns an incorrect number (such as on a cluster system), specify the number as a *negative* value to bypass the query. When not specified, model calculations will be performed using a single processor.

Parameters:

iNumProc – number of processors

Return value: int – 0 = success, otherwise error

int setNumFileIo

(const int& iNumFileIo)

Usage: Specifies the number of threads to use for the file I/O steps that are performed as part of the normal model calculations. This specification should only be called when using a 'work' directory located on RAID-5 disk unit. RAID-5 disk units are able to efficiently handle concurrent file I/O requests, while 'typical' disk drives cannot. Internally, the number specified is capped at |NumProc|-1. When not specified, these file I/O steps will be performed using a single thread.

Parameters:

iNumFileIo – number of processors

Return value: int – 0 = success, otherwise error

int setWindowsMpiMode

(const std::string& strMode)

Usage: Specifies the MPI communication mode on 64-bit Windows platforms for multi-threaded model execution. This mode determines the additional argument to be supplied to the internal usage of the Intel MPI Library process launcher utility 'mpiexec'. When not specified, the 'Local' mode is used.

Parameters:

strMode – MPI communication mode string ("Local"(default), "SSH", or "Hydra")

Local: for use on the local Windows machine with multiple processors

SSH: for use on a Windows cluster, using 'SSH' for MPI communication

Hydra: for use on a Windows cluster, relies on external 'hydra_service' for MPI communication.

The 'hydra_service' utility program is included in the Ae9Ap9/bin/win64 directory.

See https://software.intel.com/en-us/node/528873 for more information about this utility.

Return value: int – 0 = success, otherwise error

int setChunkSize

(const int& iChunkSize)

Usage: Performance tuning parameter; this specifies the number of ephemeris input entries to be processed during each call to the internal model calculation routines. When not specified, the chunk size is set to 240 by default. On systems with limited available memory resources, a sizing of 120 is recommended for improving model processing performance. Larger sizing may be used on systems with ample memory resources; the sizing for optimal performance will vary according to model inputs and parameter settings.

This specification also governs the size of data 'chunks' that are returned from each call to the various *flyin[]*() and *get[]*() data access methods in the "Model Execution and Results" section of this class API. A change in the chunk size causes an implied 'reset' of these data access methods; subsequent calls to them will restart the data access from the beginning of the ephemeris input time and positions.

Parameters:

iChunkSize – number of entries in processing chunk; values lower than 60 are not recommended

Return value: int – 0 = success, otherwise error

int getChunkSize

Usage: Returns the current value of the 'chunk' size, as described in previous method.

Parameters: -*none*-

Return value: number of entries in processing chunk

int setAsciiOut

(const std::string& strAsciiCode)

Usage: Specifies the types of model calculation results that will be produced in the form of ASCII-formatted files. For API usage, the default mode is 'None'. This method may be called multiple times for specifying more than one type.

Parameters:

strAsciiCode – data category; 'All' or 'None', or any of 'Ephem', 'Flux', 'Fluence', 'DoseRate', 'DoseAccum' and/or 'Aggreg'

Return value: int – 0 = success, otherwise error

bool isAsciiOut

(const std::string& strAsciiCode)

Usage: Queries if the specified is present or not in the current 'AsciiOut' settings. Note that if the current setting is 'All', then any of the inputs (except for 'None') will return *true*.

Parameters:

strAsciiCode – data category to be queried; 'All', 'None', 'Ephem', 'Flux', 'Fluence', 'DoseRate', 'DoseAccum' or 'Aggreg'

Return value: bool – *true* or *false*

External Ephemeris Specification:

These methods are for explicitly specifying the ephemeris (time and position coordinates) from an external source. For ephemeris generation, see the following 'Ephemeris Parameter Inputs' section.

int setInCoordSys

(const std::string& strCoordSys,
const std::string& strCoordUnits)

Usage: Specifies the coordinate system and units for the position values that are specified by the *setEphemeris*() method. When not specified, these settings default to 'GEI' and 'Re'. "Re" = radius of the Earth, defined as 6371.2 km.

Parameters:

strCoordSys – coordinate system identifier: 'GEI','GEO','GDZ','GSM','GSE','SM','MAG','SPH' or 'RLL';
Please consult the User's Guide document, "Supported Coordinate Systems" for more details.

strCoordUnits – coordinate units, 'km' or 'Re'; 'GDZ' is set to always use 'km' for the altitude value

Return value: int – 0 = success, otherwise error

int setEphemeris

(const dvector& vdTimes,
const dvector& vdCoords1,
const dvector& vdCoords2,
const dvector& vdCoords3,
bool bAppend = *false*)

Usage: Specifies the ephemeris time and positions, such as the orbit of a satellite or a grid of positions, to be used for the model calculations.

Parameters:

vdTimes – vector of time values, in Modified Julian Date form. May be identical times (for defining a *grid*) or times in chronological order, associated with position coordinates.

vdCoords1, *vdCoords2*, *vdCoords3* - vectors of position coordinate values associated with times vector. These position values are assumed to be in the coordinate system and units specified by *setInCoordSys*().

Please consult the User's Guide document, "Supported Coordinate Systems" for more details; in particular, note the expected 'standard' order of the coordinate values for non-Cartesian coordinate systems.

bAppend – optional flag for appending this ephemeris information to any ephemeris information specified in previous call(s) to *setEphemeris*(). If *false* (or if parameter is omitted), this ephemeris information will replace any existing ephemeris information.

Return value: int – 0 = success, otherwise error

void clearEphemeris

Usage: Clears any existing ephemeris information that was specified in previous calls to *setEphemeris*().

Parameters: -none-

Return value: -none-

Ephemeris Parameter Inputs:

Ephemeris generation requires the selection of an orbit propagator (and its options), a time range and time step size, and the definition of the orbit characteristics. These orbit characteristics may be defined by either a Two-Line Element (TLE) file, or a set of orbital element values. See the User's Guide '*Orbit Propagation Inputs*' section for details about each of the available settings.

int setPropagator

(const std::string& strPropSpec)

Usage: Specifies the orbit propagator algorithm to use for the ephemeris generation.

Parameters:

strPropSpec – propagator model specification; valid values: 'SatEph', 'SGP4' or 'Kepler'.

Return value: int – 0 = success, otherwise error

int setSGP4Param

(const std::string& strMode,
const std::string& strWGS)

Usage: Specifies the mode and WGS parameter for the SGP4 orbit propagator, if being used.

Parameters:

strMode – SGP4 propagation mode; valid values: 'Standard' or 'Improved'.

strWGS – World Geodetic System version; valid values: '72old', '72' or '84'.

Return value: int – 0 = success, otherwise error

void setKeplerUseJ2

(bool bUseJ2)

Usage: Specifies the use of the 'J2' perturbation for the Kepler orbit propagator, if being used.

Parameters:

bUseJ2 – flag for use of the J2 perturbation feature; *true* or *false*

Return value: -none-

int setTimes

(const double& dStartTime,
const double& dEndTime,
const double& dTimeStepSecs)

Usage: Specifies the start and stop times (*inclusive*), and time step, of the ephemeris information to be generated by the orbit propagator from the specified orbital element values or TLE file.

Parameters:

dStartTime, *dEndTime* - start and stop time values, in Modified Julian Date form

dTimeStepSecs - time step size, in seconds

Return value: int – 0 = success, otherwise error

TLE files are required to be in the standard NORAD format (see User's Guide, Appendix F). The use of the 'Kepler' propagator requires that the TLE file contain only *one* entry. For the other propagators, the TLE may contain multiple entries (for the same satellite), which must be in chronological order.

int setTLEFile

(const std::string& strTLEFile)

Usage: Specifies the name of the Two-Line Element (TLE) file (including path) to use with the selected orbit propagator; this parameter is not needed if a set of orbital element values are being used instead.
Parameters:
strTLEFile – path and filename of TLE file
Return value: int – 0 = success, otherwise error

void clearTLEFile

Usage: Unsets the specification of the TLE file.
Parameters: -none-
Return value: -none-

The orbital element values to be specified depend on the type of orbit and/or available orbit definition references. Their use also requires an associated element time to be specified. See the User's Guide document '*Orbiter Propagation Inputs*' section for more details.

int setElementTime

(const double& dElementTime)
Usage: Specifies the 'epoch' time associated with the set of orbital element values.
Parameters:
dElementTime – element 'epoch' time, in Modified Julian Date form
Return value: int – 0 = success, otherwise error

int setInclination

(const double& dInclination)
Usage: Specifies the orbital element 'Inclination' value.
Parameters:
dInclination – orbit inclination angle, in degrees (0-180)
Return value: int – 0 = success, otherwise error

int setRightAscension

(const double& dRtAscOfAscNode)
Usage: Specifies the orbital element 'Right Ascension of the Ascending Node' value.
Parameters:
dRtAscOfAscNode – orbit ascending node position, in degrees (0-360)
Return value: int – 0 = success, otherwise error

int setEccentricity

(const double& dEccentricity)
Usage: Specifies the orbital element 'Eccentricity' value.
Parameters:
dEccentricity – orbit eccentricity value, unitless (0-1)
Return value: int – 0 = success, otherwise error

int setArgOfPerigee

(const double& dArgOfPerigee)
Usage: Specifies the orbital element 'Argument of Perigee' value.

Parameters:
dArgOfPerigee – orbit perigee position, in degrees (0-360)
Return value: int – 0 = success, otherwise error

int setMeanAnomaly

(const double& dMeanAnomaly)
Usage: Specifies the orbital element 'Mean Anomaly' value.
Parameters:
dMeanAnomaly – orbit mean anomaly value, in degrees (0-360)
Return value: int – 0 = success, otherwise error

int setMeanMotion

(const double& dMeanMotion)
Usage: Specifies the orbital element 'Mean Motion' value.
Parameters:
dMeanMotion – orbit mean motion value, in units of *revolutions per day* (must be >0)
Return value: int – 0 = success, otherwise error

int setMeanMotion1stDeriv

(const double& dMeanMotion1stDeriv)
Usage: Specifies the orbital element 'First Time Derivative of the Mean Motion' value (this should NOT be divided by 2, as when specified in a TLE); this value is only used by the SatEph propagator.
Parameters:
dMeanMotion1stDeriv – first derivative of mean motion, in units of revs per day^2
Return value: int – 0 = success, otherwise error

int setMeanMotion2ndDeriv

(const double& dMeanMotion2ndDeriv)
Usage: Specifies the orbital element 'Second Time Derivative of the Mean Motion' value (this should NOT be divided by 6, as when specified in a TLE); this value is only used by the SatEph propagator.
Parameters:
dMeanMotion2ndDeriv – second derivative of mean motion, in units of revs per day^3
Return value: int – 0 = success, otherwise error

int setBStar

(const double& dBStar)
Usage: Specifies the orbital element 'B*' value, for modelling satellite drag effects; this value is only used by the SGP4 propagator.
Parameters:
dBStar – ballistic coefficient value
Return value: int – 0 = success, otherwise error

int setAltitudeOfApogee

(const double& dAltApogee)
Usage: Specifies the orbital element 'Apogee Altitude' value (furthest distance).
Parameters:

dAltApogee – altitude (in km) above the Earth's surface at the orbit's apogee
Return value: int – 0 = success, otherwise error

int setAltitudeOfPerigee

(const double& dAltPerigee)
Usage: Specifies the orbital element 'Perigee Altitude' value (closest distance).
Parameters:
dAltPerigee – altitude (in km) above the Earth's surface at the orbit's perigee
Return value: int – 0 = success, otherwise error

int setLocalTimeOfApogee

(const double& dLocTimeApogee)
Usage: Specifies the local time of the orbit's apogee.
Parameters:
dLocTimeApogee – local time, in hours (0-24)
Return value: int – 0 = success, otherwise error

int setLocalTimeMaxInclination

(const double& dLocTimeMaxIncl)
Usage: Specifies the local time of the orbit's maximum inclination (ie max latitude).
Parameters:
dLocTimeMaxIncl – local time, in hours (0-24)
Return value: int – 0 = success, otherwise error

int setTimeOfPerigee

(const double& dTimeOfPerigee)
Usage: Specifies the time of the orbit's perigee, as an alternative to the Mean Anomaly specification. *Any Mean Anomaly value also specified will be overridden by this value*.
Parameters:
dTimeOfPerigee – time, in Modified Julian Date form, for orbit perigee
Return value: int – 0 = success, otherwise error

int setSemiMajorAxis

(const double& dSemiMajorAxis)
Usage: Specifies the orbit's semi-major axis length.
Parameters:
dSemiMajorAxis – semi-major axis length, in units of Re (radius of Earth = 6371.2 km)
Return value: int – 0 = success, otherwise error

int setGeosynchLon

(const double& dGeosynchLon)
Usage: Specifies the geographic longitude of satellite in a geosynchronous orbit
Parameters:
dGeosynchLon – longitude, in degrees (-180 – 360)
Return value: int – 0 = success, otherwise error

int setPositionGEI

(const double& dX,
const double& dY,
const double& dZ)

Usage: Specifies the satellite's position in the GEI coordinate system at the element's 'epoch' time. This must be used in conjunction with the *setVelocityGEI*() method.

Parameters:

dX, dY, dZ – GEI coordinate system satellite position values, in km

Return value: int – 0 = success, otherwise error

int setVelocityGEI

(const double& dXdot,
const double& dYdot,
const double& dZdot)

Usage: Specifies the satellite's velocity in GEI coordinate system at the element's 'epoch' time. This must be used in conjunction with the *setPositionGEI*() method.

Parameters:

dXdot, dYdot, dZdot – GEI coordinate system satellite velocity values, in km/sec

Return value: int – 0 = success, otherwise error

int setCoordSys

(const std::string& strCoordSys,
const std::string& strCoordUnits)

Usage: Specifies the coordinate system and units for the position values that will be generated by the propagator specified by *setPropagator*(). These default to 'GEI' and 'Re' when not specified; if the *setInCoordSys*() method was called, the coordinate system and units will set to match those specifications. "Re" = radius of the Earth, defined as 6371.2 km.

Parameters:

strCoordSys – coordinate system identifier: 'GEI','GEO','GDZ','GSM','GSE','SM','MAG','SPH' or 'RLL';
Please consult the User's Guide document, "Supported Coordinate Systems" for more details.

strCoordUnits – 'km' or 'Re'; 'GDZ' is set to always use 'km' for the altitude value.

Return value: int – 0 = success, otherwise error

Model Parameter Inputs:

Methods for specifying the various options and settings for the radiation belt model flux calculations. Only a subset of these methods may be used with the available 'Legacy' models. See the User's Guide, Appendices A and B for more information.

int setModel

(const std::string& strModel)

Usage: Specifies the name of the flux model to be used in the calculations. See the following 'Legacy Model Parameter Inputs' section for Legacy model-specific options.

Parameters:

strModel – model name: 'AE9', 'AP9' or 'PLASMA'
Legacy models: 'AE8', 'AP8', 'CRRESELE', 'CRRESPRO' or 'CAMMICE'

Return value: int – 0 = success, otherwise error

int setModelDBFile

(const std::string& strDataSource)

Usage: Specifies the name of the database file (including path) for flux model calculations.
Please consult the User's Guide for the exact database filename associated with each model.

Parameters:

strDataSource – model database filename, including path

Return value: int – 0 = success, otherwise error

int setKPhiNNetDBFile

(const std::string& strDataSource)

Usage: Specifies the name of the file (including path) for the K/Phi neural network database.
This database name is in the form of '<path>/fastPhi_net.mat'.

Parameters:

strDataSource – neural network database filename, including path

Return value: int – 0 = success, otherwise error

int setKHMinNNetDBFile

(const std::string& strDataSource)

Usage: Specifies the name of the file (including path) for the K/Hmin neural network database.
This database name is in the form of '<path>/fast_hmin_net.mat'.

Parameters:

strDataSource – neural network database filename, including path

Return value: int – 0 = success, otherwise error

int setMagfieldDBFile

(const std::string& strDataSource)

Usage: Specifies the name of the file (including path) for the magnetic field model database.
This database name is in the form of '<path>/igrfDB.h5'.

Parameters:

strDataSource – magnetic field model database filename, including path

Return value: int – 0 = success, otherwise error

int setDoseModelDBFile

(const std::string& strDataSource)

Usage: Specifies the name of the file (including path) for the dose calculation model database.

This database name is in the form of '<path>/sd2DB.h5'. The use of this method is *optional*. If not specified, the file is assumed to be located in the same directory as the magnetic field database file, 'igrfDB.h5'.

Parameters:

strDataSource – dose calculation model database filename, including path

Return value: int – 0 = success, otherwise error

void setAdiabatic

(bool bVerdict = *true*)

Usage: Specifies whether or not the full adiabatic invariant values are to be calculated and be available during model run. Not applicable for Legacy model runs.

Parameters:

bVerdict – optional flag for activating(default) or deactivating the adiabatic invariant calculation.

Return value: int – 0 = success, otherwise error

int setFluxType

(const std::string& strFluxType)

Usage: Specifies the type of flux values to be calculated by the model. Note that use of '2PtDiff' type requires that both *setEnergies*() and *setEnergies2*() to be specified.

Parameters:

strFluxType – flux type identifier: '1PtDiff', '2PtDiff' or 'Integral'

Return value: int – 0 = success, otherwise error

int setFluxEnergies

(const dvector& vdEnergies);

Usage: Specifies the set energies at which the flux values are calculated by the selected model.

Please consult User's Guide for valid ranges/values, which depend on the model selected.

Parameters:

vdEnergies – vector of energy values; if '2PtDiff' flux type, these specify lower bounds of energy bins.

Return value: int – 0 = success, otherwise error

int setFluxEnergies2

(const dvector& vdEnergies2);

Usage: Specifies the upper bounds of the energy bins, corresponding with the energies (as lower bounds) specified using the *setFluxEnergy*() method; these upper bound values only required for '2PtDiff' flux type.

Please consult User's Guide for valid ranges/values, which depend on the model selected.

Parameters:

vdEnergies2 – vector of energy values, specifying upper bounds of energy bins for '2PtDiff' flux type.

Return value: int – 0 = success, otherwise error

int setPitchAngle

(const double& dPitchAngle)

Usage: Specifies a uni-directional pitch angle for the model calculations (not valid for Legacy models). Multiple calls to this method add to the list of pitch angles. *Not compatible with Dose calculations.*

Parameters:

dPitchAngle – pitch angle, in degrees (0-180)

Return value: int – 0 = success, otherwise error

int setPitchAngle

Usage: Clears the pitch angle list defined by calls to *setPitchAngle*() or *setPitchAngles*() methods.

Parameters: -none-

Return value: int – 0 = success, otherwise error

int setPitchAngles

(const dvector& vdPitchAngles)

Usage: Specifies a list of uni-directional pitch angles for the model calculations (not valid for Legacy models). This replaces any previously-defined pitch angle list values. *Not compatible with Dose calculations.*

Parameters:

vdPitchAngles – vector of pitch angles, in degrees (0-180)

Return value: int – 0 = success, otherwise error

void setFluxMean

(bool bVerdict = *true*)

Usage: Specifies the calculation of the 'mean' flux values by the selected model. All flux values calculated by the Legacy models are considered 'mean' fluxes.

Parameters:

bVerdict – optional flag for activating(default) or deactivating the calculation of mean flux values.

Return value: -none-

int setFluxPercentile

(const int& iPercent)

Usage: Specifies the calculation of 'percentile' flux values by the selected model (not valid for Legacy models). This method may be called multiple times for the specification of more than one percentile.

Parameters:

iPercent – percentile flux value(1-99) to be calculated by the model

Return value: int – 0 = success, otherwise error

int setFluxPercentile

(const ivector& viPercent)

Usage: Specifies the calculation of one or more 'percentile' flux values by the selected model (not valid for Legacy models). The list of percentile values supersedes any prior calls for defining percentile values.

Parameters:

viPercent – vector of percentile flux values (1-99) to be calculated by the model

Return value: int – 0 = success, otherwise error

void clearFluxPercentile

Usage: Clears the current list of defined flux percentile values.
Parameters: -none-
Return value: -none-

int setFluxPerturbedScen

(const int& iScenario)
Usage: Specifies the calculation of the particular scenario number of 'perturbed mean' flux values by the selected model (not valid for Legacy models). This method may be called multiple times for the specification of more than one scenario number.
Parameters:
iScenario – scenario number (1-999) of perturbed mean flux values to be calculated by the model
Return value: int – 0 = success, otherwise error

int setFluxPerturbedScen

(const ivector& viScenario)
Usage: Specifies the calculation of one or more scenario number of 'perturbed mean' flux values by the selected model (not valid for Legacy models). The list of scenario numbers supersedes any prior calls for defining the scenario numbers for perturbed mean calculations.
Parameters:
viScenario – vector of scenario numbers (1-999) of perturbed mean flux values to be calculated by the model
Return value: int – 0 = success, otherwise error

int setFluxPerturbedScenRange

(const int& iScenarioMin,
const int& iScenarioMax)
Usage: Specifies the calculation of several scenario numbers, defined by an inclusive range, of 'perturbed mean' flux values by the selected model (not valid for Legacy models). The resulting list of scenario numbers supersedes any prior calls for defining the scenario numbers for perturbed mean calculations.
Parameters:
iScenarioMin – first of the range of scenario numbers (1-999) of perturbed mean flux values to be calculated by the model
iScenarioMax – last of the range of scenario numbers (1-999) of perturbed mean flux values to be calculated by the model
Return value: int – 0 = success, otherwise error

void clearFluxPerturbedScen

Usage: Clears the current list of defined perturbed mean scenario numbers.
Parameters: -none-
Return value: -none-

int setFluxMonteCarloScen

(const int& iScenario)

Usage: Specifies the calculation of the particular scenario number of 'Monte Carlo' flux values by the selected model (not valid for PLASMA or Legacy models). This method may be called multiple times for the specification of more than one scenario number.

Parameters:

iScenario – scenario number (1-999) of Monte Carlo flux values to be calculated by the model

Return value: int – 0 = success, otherwise error

int setFluxMonteCarloScen

(const ivector& viScenario)

Usage: Specifies the calculation of one or more scenario number of 'Monte Carlo' flux values by the selected model (not valid for PLASMA or Legacy models). The list of scenario numbers supersedes any prior calls for defining the scenario numbers for Monte Carlo calculations.

Parameters:

viScenario – vector of scenario numbers (1-999) of Monte Carlo flux values to be calculated by the model

Return value: int – 0 = success, otherwise error

int setFluxMonteCarloScenRange

(const int& iScenarioMin,
const int& iScenarioMax)

Usage: Specifies the calculation of several scenario numbers, defined by an inclusive range, of 'Monte Carlo' flux values by the selected model (not valid for PLASMA or Legacy models). The resulting list of scenario numbers supersedes any prior calls for defining the scenario numbers for Monte Carlo calculations.

Parameters:

iScenarioMin – first of the range of scenario numbers (1-999) of Monte Carlo flux values to be calculated by the model

iScenarioMax – last of the range of scenario numbers (1-999) of Monte Carlo flux values to be calculated by the model

Return value: int – 0 = success, otherwise error

void clearFluxMonteCarloScen

Usage: Clears the current list of defined Monte Carlo scenario numbers.

Parameters: -none-

Return value: -none-

int setMonteCarloEpochTime

(const double &dEpochTime)

Usage: Specifies the reference time used in the time progression of the Monte Carlo flux calculations.

Parameters:

dEpochTime – Monte Carlo reference time, in Modified Julian Date form

Return value: int – 0 = success, otherwise error

void setMonteCarloFluxPerturb

(bool bVerdict)

Usage: Enables (*true*) or disables (*false*) the flux perturbations in Monte Carlo calculations.values by the selected model. The default perturbation mode is enabled. Disabling these perturbations is generally only useful for validation or where perturbations dwaft physical features of interest..

Parameters:

bVerdict – *true* or *false* for the use of flux perturbations in the Monte Carlo calculations.

Return value: -none-

The following methods specify the further processing to be performed using the calculated flux values.

These 'Accumulation' settings affect the results of the calculated fluence, dose rate, accumulated dose, and some forms of the processed flux values.

int setAccumMode

(const std::string& strAccumMode)

Usage: Specifies the accumulation mode to be used for the processing of the model flux results. When not specified, the accumulation mode defaults to 'Interval', with a length of 1 day, unless otherwise specified via the *setAccumInterval[Sec]*() method.

Please consult the User's Guide for more details about these accumulation modes.

Parameters:

strAccumMode – accumulation mode identifier: 'Cumul'|'Cumulative', 'Intv'|'Interval', 'Full', 'Boxcar' or 'Expon'|'Exponential' [the 'Boxcar' and 'Exponential' modes are currently considered "experimental"]

Return value: int – 0 = success, otherwise error

int setAccumInterval

(const double& dVal)

Usage: Specifies the time duration of the accumulation of flux data for use in the calculation of the fluence and/or dose results for the 'Interval', 'Boxcar' and/or 'Exponential' modes. When not specified, this duration defaults to 1.0 days (86400 seconds).

Parameters:

dVal – time duration, in units of days+fraction.

Return value: int – 0 = success, otherwise error

int setAccumIntervalSec

(const double& dVal)

Usage: Same as *setAccumInterval*(), except that the duration is specified in seconds. When not specified, this duration defaults to 86400 seconds (1.0 days).

Parameters:

dVal – time duration, in units of seconds.

Return value: int – 0 = success, otherwise error

int setAccumIncrementSec

(const double& dVal)

Usage: Specifies the time delta for the shift of the 'Boxcar' accumulation mode time windows.

Parameters:

dVal – time delta, in seconds, for the increment of time between the start of adjacent Boxcar time windows. Must be greater than 0 and less than the specified interval duration.

Return value: int – 0 = success, otherwise error

int setAccumIncrementFrac

(const double& dVal)

Usage: Specifies the time delta for the shift of the Boxcar accumulation time windows, expressed as a fraction of the accumulation time window duration (specified in *setAccumInterval[Sec]*() calls) .

Parameters:

dVal – fraction, between 0.0 and 1.0 (exclusive of the ends).

Return value: int – 0 = success, otherwise error

void setFluence

(bool bVerdict = *true*)

Usage: Specifies the calculation of fluence values from the flux results of the model run. Use of the *setAccumMode*() and *setAccumInterval[Sec]*() methods will affect the frequency and value of these fluence results; when unspecified, these default to the accumulated fluence being reported at 1 day Intervals.

Parameters:

bVerdict – optional flag for activating(default) or deactivating the calculation of fluence values.

Return value: -none-

Dose Calculations require the use of omni-directional (ie, *pitch angle specifications are NOT permitted*), '1PtDiff'-type differential flux values as their input. A minimum of three shielding depth values are also required. Dose calculations are available for all models except 'PLASMA' and 'CAMMICE'.

void setDoseRate

(bool bVerdict = *true*)

Usage: Specifies the calculation of dose rate values from the flux results of the model run. Use of the *setAccumMode*() and *setAccumInterval[Sec]*() methods will affect the frequency at which these dose rate results are available; when unspecified, these default to 'Interval' dose rate averages over 1 day periods.

Parameters:

bVerdict – optional flag for activating(default) or deactivating the calculation of dose rate values.

Return value: -none-

void setDoseAccum

(bool bVerdict = *true*)

Usage: Specifies the calculation of cumulative or accumulated dose values from the flux results of the model run. Use of the *setAccumMode*() and *setAccumInterval[Sec]*() methods will affect the frequency at which these accumulated dose results are available; when unspecified, these default to the accumulated dose being reported at 1 day Intervals.

Parameters:

bVerdict – optional flag for activating(default) or deactivating the calculation of dose accum values.

Return value: -none-

int setDoseDepthValues

(const dvector& vdDepths)

Usage: Specifies the list of aluminum shielding thickness depths, in the units specified by *setDoseDepthUnits*() method. A minimum of three depth values are required for performing a model run. Please consult User's Guide for nominal range of values, dependent upon the units selected.

Parameters:

vdDepths – vector of depth values

Return value: int – 0 = success, otherwise error

int setDoseDepthUnits

(const std::string& strDepthUnits)

Usage: Specifies the measurement units associated with the depth values specified using the *setDoseDepthValues*() method.

Parameters:

strDepthUnits – unit specification: 'millimeters'|'mm', 'mils' or 'gpercm2'

Return value: int – 0 = success, otherwise error

int setDoseDepths

(const dvector& vdDepths,
const std::string& strDepthUnits)

Usage: Specifies both the list of aluminum shielding thickness depths, and their associated units. A minimum of three depth values are required for performing a model run. Input depth values are expected to be in increasing order, with no duplicates; they will be sorted automatically.

Please consult User's Guide for nominal range of values, dependent upon the units selected.

Parameters:

vdDepths – vector of depth values

strDepthUnits – unit specification: 'millimeters'|'mm', 'mils' or 'gpercm2'

Return value: int – 0 = success, otherwise error

int setDoseDetector

(const std::string& strDetector)

Usage: Specifies the dose detector material type that lies behind the aluminum shielding.

Parameters:

strDetector – material name: 'Aluminum'|'Al', 'Graphite', 'Silicon'|'Si', 'Air', 'Bone', 'Tissue', 'Calcium'|'Ca', 'Gallium'|'Ga', 'Lithium'|'Li', 'Glass'|'SiO2', 'Water'|'H2O'

Return value: int – 0 = success, otherwise error

int setDoseGeometry

(const std::string& strGeometry)

Usage: Specifies the geometry of the aluminum shielding in front of (or around) the detector target.

Parameters:

strGeometry – configuration name: 'Spherical', 'FiniteSlab' or 'SemiInfiniteSlab'

Return value: int – 0 = success, otherwise error

int setDoseNuclearAttenMode

(const std::string& strNucAttenMode)

Usage: Specifies the 'Nuclear Attenuation' mode used during the ShieldDose2 model calculations.

Parameters:

strNucAttenMode – attenuation mode: 'None', 'NuclearInteractions' or 'NuclearAndNeutrons'

Return value: int – 0 = success, otherwise error

These 'Aggregation' settings are used for the calculation of 'confidence levels' from the 'Perturbed Mean' and/or 'Monte Carlo' scenario results. These confidence levels are determined using the percentile calculation method recommended by the National Institute of Standards and Technology (NIST). The endpoints of 0 and100 percent levels are excluded, as well as additional neighboring levels when fewer than 100 scenarios are used in the aggregation (see the User's Guide for more information). The 0 percent level returns the lowest scenario value; the 100 percent level returns the highest scenario value. These results are statistically meaningful only when at least ten scenarios are used.

int setAggregMean

Usage: Adds the 'Mean' to the list for aggregation calculations*. This is NOT a confidence level. The results of this calculation are of indeterminate meaning. Use of this method is discouraged.*
Parameters: -none-
Return value: int – 0 = success, otherwise error

int setAggregMedian

Usage: Adds the 50% value to the list for aggregation confidence level calculations.
Parameters: -none-
Return value: int – 0 = success, otherwise error

int setAggregConfLevel

(const int& iPercent)
Usage: Adds the specified percent value to the list for aggregation confidence level calculations.
Parameters:
iPercent – percent value to add to list for aggregation confidence level calculations (0-100 valid).
Return value: int – 0 = success, otherwise error

void clearAggregConfLevel

Usage: Clears the accumulated list of percent values for aggregation confidence level calculations.
Parameters: -none-
Return value: -none-

Legacy Model Parameter Inputs:

These methods are used for specifying the model parameters applicable only to the 'Legacy' models. The flux values calculated by these models are all 'mean', omni-directional flux values.

int setActivityLevel

(const std::string& strActivityLevel)

Usage: Specifies the geomagnetic activity level parameter for the CRRESPRO, AE8 or AP8 Legacy model run.

Parameters:

strActivityLevel – geomagnetic activity level specification:
for CRRESPRO model, 'active' or 'quiet';
for AE8 or AP8 model, 'min' or 'max'.

Return value: int – 0 = success, otherwise error

int setActivityRange

(const std::string& strActivityRange)

Usage: Specifies the geomagnetic activity level parameter for the CRRESELE Legacy model run. Only one of the *setActivityRange*() or *set15DayAvgAp*() methods may be used, otherwise an error is flagged.

Parameters:

strActivityRange – geomagnetic activity level specification, in terms of Ap values:
'5-7.5', '7.5-10', '10-15', '15-20', '20-25', '>25', 'avg', 'max', or 'all'.

Return value: int – 0 = success, otherwise error

int setActivityLevel

(const std::string& strActivityLevel)

Usage: Specifies the geomagnetic activity level parameter for the CRRESPRO, AE8 or AP8 Legacy model run.

Parameters:

strActivityLevel – geomagnetic activity level specification:
for CRRESPRO model, 'active' or 'quiet';
for AE8 or AP8 model, 'min' or 'max'.

Return value: int – 0 = success, otherwise error

int set15DayAvgAp

(const double& d15DayAvgAp)

Usage: Specifies the 15-day average Ap value for the CRRESELE Legacy model run. Only one of the *setActivityRange*() or *set15DayAvgAp*() methods may be used, otherwise an error is flagged.

Parameters:

d15DayAvgAp – 15-day average Ap value; valid values are in the range of 0 – 400.

Return value: int – 0 = success, otherwise error

void setFixedEpoch

(bool bFixedEpoch)

Usage: Specifies the use of the model-specific fixed epoch (year) for the magnetic field model in the flux calculations. It is *highly recommended* to set this to 'true'. Unphysical results may be produced (especially at low altitudes) if set to 'false'.

Parameters:
bFixedEpoch – *true* or *false*; when *false*, the ephemeris year is used for the magnetic field model.
Return value: -none-

void setShiftSAA

(bool bShiftSAA)
Usage: Shifts the SAA from its fixed-epoch location to the location for the current year of the ephemeris. This setting is ignored if the ***setFixedEpoch*** method is set to 'false'.
Parameters:
bShiftSAA – *true* or *false*.
Return value: -none-

The following methods are applicable only to the CAMMICE/MICS Legacy model. This model is set to produce flux values for twelve pre-defined energy bins (see Appendix B of the User's Guide).

int setMagfieldModel

(const std::string& strMFModel)
Usage: Specifies the magnetic field option for the CAMMICE Legacy model run. 'igrf' uses the IGRF model without an external field model. 'igrfop' adds Olson-Pfitzer/Quiet as the external field model.
Parameters:
strMFModel – magnetic field model specification: 'igrf' or 'igrfop'.
Return value: int – 0 = success, otherwise error

int setDataFilter

(const std::string& strDataFilter)
Usage: Specifies the data filter option for the CAMMICE Legacy model run. 'Filtered' excludes data collected during periods when the DST index was below -100.
Parameters:
strDataFilter – data filter specification: 'all' or 'filtered' .
Return value: int – 0 = success, otherwise error

int setPitchAngleBin

(const std::string& strPitchAngleBin)
Usage: Specifies the pitch angle bin for the CAMMICE Legacy model run.
Parameters:
strPitchAngleBin – pitch angle bin identification: '0-10','10-20','20-30','30-40','40-50','50-60','60-70','70-80','80-90', '100-110','110-120','120-130','130-140','140-150','150-160','160-170','170-180' or 'omni'.
Return value: int – 0 = success, otherwise error

int setSpecies

(const std::string& strSpecies)
Usage: Specifies the (single) particle species for the CAMMICE Legacy model run.
Parameters:
strSpecies – species identification: 'H+', 'He+', 'He+2', 'O+', 'H', 'He', 'O', or 'Ions'.
Return value: int – 0 = success, otherwise error

Model Execution and Results:

These methods are to be used after all desired input parameters have been specified.

Following a call to the *runModel*() method, the results from the requested calculations are accessible via the various *flyin[]*() and *get[]*() methods.
Important: Please note that these *flyin[]*() and *get[]*() methods return the requested data in 'chunks', defaulting to 240 entries at each method call. Therefore, multiple calls may be required to access the full set of generated model results. Following the call to the *runModel*() method, the sizing of these data access segments may be adjusted using the *setChunkSize*() method. A call to the *resetModelData*() method will 'reset' these data access methods, as will a call to change the chunk size. Subsequent calls to the data access methods will restart them from the beginning of the ephemeris input time and positions.

The *flyin[]*() and *get[]Data*() methods include an optional argument '*bFluxAccumAvg*'. This flag is used to distinguish which *flux* data is to be returned when an accumulation is active; in such cases, the available data includes the flux values calculated at the specific ephemeris input times and positions, and the flux *average* values over the defined time interval periods. When this flag is omitted, it defaults to *false*, and so the first type of data is returned; when this flag is *true*, the second type of data is returned. An error occurs when the flag is *true* but the accumulation mode is 'Cumulative' (which has no interval).

int runModel

Usage: Invokes the execution of the model calculations based on the specified parameter inputs. When errors in the inputs/settings are detected, informative messages are shown in the console output.
Parameters: -none-
Return value: int – 0 = success, otherwise error

int getEphemeris

(dvector& vdTimes,
dvector& vdCoord1,
dvector& vdCoord2,
dvector& vdCoord3)

Usage: Returns the ephemeris information, either generated or specified.
Parameters:
vdTimes – returned vector of ephemeris time values, in Modified Julian Date form
vdCoord1, vdCoord2, vdCoord3 – returned vectors of ephemeris position values, in the coordinate system and units previously specified (also accessible from *getCoordSys*() and *getCoordUnit*() methods). Please consult the User's Guide document, "Supported Coordinate Systems" for more details; in particular, note the 'standard' ordering of the returned coordinate values for non-Cartesian coordinate systems.
Return value: int – Number of ephemeris records returned (≥0 = success, <0 = error)

void getCoordSys

(std::string& strCoordSys)

Usage: Returns the coordinate system of the ephemeris information returned with the model results.
Parameters:

strCoordSys – returned coordinate system identifier; see *set[In]CoordSys*() methods for list of values, or consult the User's Guide document, "Supported Coordinate Systems" for more details.
Return value: -none-

void getCoordUnits

(std::string& strCoordUnits)
Usage: Returns the coordinate system units of the ephemeris information returned with the model results.
Parameters:
strCoordUnits – returned coordinate system unit value; 'km' or 'Re' (radius of Earth = 6731.2 km)
Return value: -none-

int flyinMean

(vdvector& vvdFluxData,
bool bFluxAccumAvg = *false*)
Usage: Returns the 'Mean' model flux, for omni-directional model runs. This method may also be used for accessing flux results from the 'Legacy' models. A previous call to the *setFluxMean(<true>)* method is required for these results to be available for the Ae9/Ap9/SPM models.
Parameters:
vvvdFluxData – returned 2-dimensional vector of the 'mean' flux values. [time, energy]
bFluxAccumAvg – optional flag for returning flux accumulation average values, if *true*. When specified as *false,* or omitted, returns flux values at the ephemeris input times and positions.
Return value: int – 0 = success, otherwise error

int flyinMean

(vvdvector& vvvdFluxData,
bool bFluxAccumAvg = *false*)
Usage: Returns the 'Mean' model flux. This method can also be used for accessing flux results from the 'Legacy' models. A previous call to the *setFluxMean(<true>)* method is required for these results to be available for the Ae9/Ap9/SPM models.
Parameters:
vvvdFluxData – returned 3-dimensional vector of the 'mean' flux values. [time,energy,direction]
bFluxAccumAvg – optional flag for returning flux accumulation average values, if *true*. When specified as *false,* or omitted, returns flux values at the ephemeris input times and positions.
Return value: int – 0 = success, otherwise error

int flyinMean

(dvector& vdTimes
dvector& vdCoord1,
dvector& vdCoord2,
dvector& vdCoord3,
vdvector& vvdPitchAngles,
vvdvector& vvvdFluxData,
bool bFluxAccumAvg = *false*)
Usage: Returns the 'Mean' model flux, along with the associated ephemeris values and pitch angles. This method can also be used for accessing flux results from the 'Legacy' models. A previous call to the *setFluxMean(<true>)* method is required for these results to be available for the Ae9/Ap9/SPM models.

Parameters:

vdTimes – returned vector of ephemeris time values, in Modified Julian Date form

vdCoord1, vdCoord2, vdCoord3 – returned vectors of ephemeris position values, in the coordinate system and units previously specified (also accessible from *getCoordSys*() and *getCoordUnit*() methods). Please consult the User's Guide document, "Supported Coordinate Systems" for more details; in particular, note the 'standard' ordering of the returned coordinate values for non-Cartesian coordinate systems.

vvdPitchAngles – returned 2-dimensional vector of associated pitch angles; will be empty if omni-directional.[time,direction]

vvvdFluxData – returned 3-dimensional vector of the 'mean' flux values. [time,energy,direction]

bFluxAccumAvg – optional flag for returning flux accumulation average values, if *true*. When specified as *false,* or omitted, returns flux values at the ephemeris input times and positions.

Return value: int – 0 = success, otherwise error

int flyinPercentile

(const int& iPercentile,
vvdvector& vvvdFluxData,
bool bFluxAccumAvg = *false*)

Usage: Returns the model flux results for the specified model Percentile number. The percentile number specified must be one of those included in previous calls to the *setFluxPercentile()* methods.

Parameters:

iPercentile – percentile number of the flux values to be returned.

vvvdFluxData – returned 3-dimensional vector of the flux values for the specified percentile. [time,energy,direction]

bFluxAccumAvg – optional flag for returning flux accumulation average values, if *true*. When specified as *false,* or omitted, returns flux values at the ephemeris input times and positions.

Return value: int – 0 = success, otherwise error

int flyinPercentile

(const int& iPercentile,
dvector& vdTimes,
dvector& vdCoord1,
dvector& vdCoord2,
dvector& vdCoord3,
vdvector& vvdPitchAngles,
vvdvector& vvvdFluxData,
bool bFluxAccumAvg = *false*)

Usage: Returns the model flux results for the specified model Percentile number, along with the associated ephemeris values and pitch angles. The percentile number specified must be one of those included in previous calls to the *setFluxPercentile()* methods.

Parameters:

iPercentile – percentile number of the flux values to be returned.

vdTimes – returned vector of ephemeris time values, in Modified Julian Date form

vdCoord1, vdCoord2, vdCoord3 – returned vectors of ephemeris position values, in the coordinate system and units previously specified (also accessible from *getCoordSys*() and *getCoordUnit*() methods).

Please consult the User's Guide document, "Supported Coordinate Systems" for more details; in particular, note the 'standard' ordering of the returned coordinate values for non-Cartesian coordinate systems.

vvdPitchAngles – returned 2-dimensional vector of associated pitch angles; will be empty if omni-directional.[time,direction]

vvvdFluxData – returned 3-dimensional vector of the flux values for the specified percentile. [time,energy,direction]

bFluxAccumAvg – optional flag for returning flux accumulation average values, if *true*. When specified as *false,* or omitted, returns flux values at the ephemeris input times and positions.

Return value: int – 0 = success, otherwise error

int flyinPerturbedMean

(const int& iScenario,
vvdvector& vvvdFluxData,
bool bFluxAccumAvg = *false*)

Usage: Returns the model flux results for the specified Perturbed Mean scenario number. The scenario number specified must be one of those included in previous calls to the *setFluxPerturbedScen [Range]()* methods.

Parameters:

iScenario – scenario number of the Perturbed Mean flux values to be returned.

vvvdData – returned 3-dimensional vector of the flux values for the specified scenario number. [time,energy,direction]

bFluxAccumAvg – optional flag for returning flux accumulation average values, if *true*. When specified as *false,* or omitted, returns flux values at the ephemeris input times and positions.

Return value: int – 0 = success, otherwise error

int flyinPerturbedMean

(const int& iScenario,
dvector& vdTimes,
dvector& vdCoord1,
dvector& vdCoord2,
dvector& vdCoord3,
vdvector& vvdPitchAngles,
vvdvector& vvvdFluxData,
bool bFluxAccumAvg = *false*)

Usage: Returns the model flux results for the specified Perturbed Mean scenario number, along with the associated ephemeris values and pitch angles. The scenario number specified must be one of those included in previous calls to the *setFluxPerturbedScen[Range]()* methods.

Parameters:

iScenario – scenario number of the Perturbed Mean flux values to be returned.

vdTimes – returned vector of ephemeris time values, in Modified Julian Date form

vdCoord1, vdCoord2, vdCoord3 – returned vectors of ephemeris position values, in the coordinate system and units previously specified (also accessible from *getCoordSys*() and *getCoordUnit*() methods). Please consult the User's Guide document, "Supported Coordinate Systems" for more details; in particular, note the 'standard' ordering of the returned coordinate values for non-Cartesian coordinate systems.

vvdPitchAngles – returned 2-dimensional vector of associated pitch angles; will be empty if omni-directional.[time,direction]

vvvdFluxData – returned 3-dimensional vector of the flux values for the specified scenario number. [time,energy,direction]

bFluxAccumAvg – optional flag for returning flux accumulation average values, if *true*. When specified as *false,* or omitted, returns flux values at the ephemeris input times and positions.

Return value: int – 0 = success, otherwise error

int flyinMonteCarlo

(const int& iScenario,
vvdvector& vvvdFluxData,
bool bFluxAccumAvg = *false*)

Usage: Returns the model flux results for the specified Monte Carlo scenario number. The scenario number specified must be one of those included in previous calls to the *setFluxMonteCarloScen[Range]()* methods. This method supersedes the '*flyinScenario*' from previous Ae9Ap9 API versions.

Parameters:

iScenario – scenario number of the Monte Carlo flux values to be returned.

vvvdFluxData – returned 3-dimensional vector of the flux values for the specified scenario number. [time,energy,direction]

bFluxAccumAvg – optional flag for returning flux accumulation average values, if *true*. When specified as *false,* or omitted, returns flux values at the ephemeris input times and positions.

Return value: int – 0 = success, otherwise error

int flyinMonteCarlo

(const int& iScenario,
dvector& vdTimes,
dvector& vdCoord1,
dvector& vdCoord2,
dvector& vdCoord3,
vdvector& vvdPitchAngles,
vvdvector& vvvdFluxData,
bool bFluxAccumAvg = *false*)

Usage: Returns the model flux results for the specified Monte Carlo scenario number, along with the associated ephemeris values and pitch angles. The scenario number specified must be one of those included in previous calls to the *setFluxMonteCarloScen[Range]()* methods.

Parameters:

iScenario – scenario number of the Monte Carlo flux values to be returned.

vdTimes – returned vector of ephemeris time values, in Modified Julian Date form

vdCoord1, vdCoord2, vdCoord3 – returned vectors of ephemeris position values, in the coordinate system and units previously specified (also accessible from *getCoordSys*() and *getCoordUnit*() methods). Please consult the User's Guide document, "Supported Coordinate Systems" for more details; in particular, note the 'standard' ordering of the returned coordinate values for non-Cartesian coordinate systems.

vvdPitchAngles – returned 2-dimensional vector of associated pitch angles; will be empty if omni-directional.[time,direction]

vvvdFluxData – returned 3-dimensional vector of the flux values for the specified scenario number. [time,energy,direction]

bFluxAccumAvg – optional flag for returning flux accumulation average values, if *true*. When specified as *false,* or omitted, returns flux values at the ephemeris input times and positions.

Return value: int – 0 = success, otherwise error

int getAdiabaticCoords

(vdvector& vvdAlpha,
vdvector& vvdLm,
vdvector& vvdK,
vdvector& vvdPhi,
vdvector& vvdHmin,
vdvector& vvdLstar,
dvector& vdBmin,
dvector& vdBlocal,
dvector& vdMagLT)

Usage: Returns the adiabatic invariant values associated with the previously-defined or generated ephemeris and pitch angles. If omnidirectional, values returned are for pitch angle of 90.

Parameters:

vvdAlpha – returned 2-dimensional vector of equatorial pitch angles ('alpha') associated with the pitch angles at the ephemeris locations. [time,direction]

vvdLm – returned 2-dimensional vector of McIllwain L-shell value associated with the pitch angles at the ephemeris locations. [time,direction]

vvdK – returned 2-dimensional vector of adiabatic invariant 'K' value associated with the pitch angles at the ephemeris locations. [time,direction]

vvdPhi – returned 2-dimensional vector of adiabatic invariant 'Phi' associated with the pitch angles at the ephemeris locations. [time,direction]

vvdHmin – returned 2-dimensional vector of adiabatic invariant 'Hmin' associated with the pitch angles at the ephemeris locations. [time,direction]

vvdLstar – returned 2-dimensional vector of adiabatic invariant 'L*' associated with the pitch angles at the ephemeris locations. [time,direction]

vdBmin – returned 1-dimensional vector of minimum IGRF model magnetic field strength value (nanoTeslas) along field line containing the ephemeris location. [time].

vdBlocal – returned 1-dimensional vector of IGRF model magnetic field strength value (nanoTeslas) at the ephemeris location. [time]

vdMagLT – returned 1-dimensional vector of dipole model-based magnetic local time (hours) at the ephemeris locations. [time]

Return value: int – 0 = success, otherwise error

int getAdiabaticCoords

(dvector& vdTimes,
dvector& vdCoord1,
dvector& vdCoord2,
dvector& vdCoord3,
vdvector& vvdPitchAngles,
vdvector& vvdAlpha,
vdvector& vvdLm,
vdvector& vvdK,
vdvector& vvdPhi,

vdvector& vvdHmin,
vdvector& vvdLstar,
dvector& vdBmin,
dvector& vdBlocal,
dvector& vdMagLT)

Usage: Returns the ephemeris values, pitch angles and corresponding adiabatic invariant values.

Parameters:

vdTimes – returned vector of ephemeris time values, in Modified Julian Date form

vdCoord1, vdCoord2, vdCoord3 – returned vectors of ephemeris position values, in the coordinate system and units previously specified (also accessible from *getCoordSys*() and *getCoordUnit*() methods). Please consult the User's Guide document, "Supported Coordinate Systems" for more details; in particular, note the 'standard' ordering of the returned coordinate values for non-Cartesian coordinate systems.

vvdPitchAngles – returned 2-dimensional vector of associated pitch angles; will contain single angle of 90 for all times if omni-directional.[time,direction]

vvdAlpha – returned 2-dimensional vector of equatorial pitch angles ('alpha') associated with the pitch angles at the ephemeris locations. [time,direction]

vvdLm – returned 2-dimensional vector of McIllwain L-shell value associated with the pitch angles at the ephemeris locations. [time,direction]

vvdK – returned 2-dimensional vector of adiabatic invariant 'K' value associated with the pitch angles at the ephemeris locations. [time,direction]

vvdPhi – returned 2-dimensional vector of adiabatic invariant 'Phi' associated with the pitch angles at the ephemeris locations. [time,direction]

vvdHmin – returned 2-dimensional vector of adiabatic invariant 'Hmin' associated with the pitch angles at the ephemeris locations. [time,direction]

vvdLstar – returned 2-dimensional vector of adiabatic invariant 'L*' associated with the pitch angles at the ephemeris locations. [time,direction]

vdBmin – returned 1-dimensional vector of minimum IGRF model magnetic field strength value (nanoTeslas) along field line containing the ephemeris location. [time].

vdBlocal – returned 1-dimensional vector of IGRF model magnetic field strength value (nanoTeslas) at the ephemeris location. [time]

vdMagLT – returned 1-dimensional vector of dipole model-based magnetic local time (hours) at the ephemeris locations. [time]

Return value: int – 0 = success, otherwise error

int getModelData

(const std::string& strDataType,
const std::string& strFluxMode,
const int& iCalcVal,
dvector& vdTimes,
dvector& vdCoord1,
dvector& vdCoord2,
dvector& vdCoord3,
vdvector& vvdPitchAngles,
vvdvector& vvvdData,
bool bFluxAccumAvg = *false*)

Usage: Returns the model results from for the specified data type, flux mode and, if applicable, the percentile or scenario identifier. See the following routine for accessing *aggregation* results. The associated ephemeris and possibly pitch angles are also returned.

Parameters:

strDataType – data type identifier: "flux"|"fluence"|"doserate"|"doseaccum"

strFluxMode – flux mode identifier: "mean"|"percent"|"perturbed"|"montecarlo"|"adiabat"(only w/ 'flux')

iCalcVal – additional model data qualifier: ignored for 'mean' and 'adiabat' flux modes; model percentile (1-99) for 'percent'; model scenario number (1-999) for 'perturbed' or 'montecarlo'. The number specified must be one of those included in previous calls to the appropriate *setFluxPercentile(), setFluxPerturbedScen[Range]()* or *setFluxMonteCarloScen[Range]()* methods.

vdTimes – returned vector of ephemeris time values, in Modified Julian Date form; when using an accumulation mode other than 'None', this time specifies the *ending* time of the accumulation interval.

vdCoord1, vdCoord2, vdCoord3 – returned vectors of ephemeris position values, in the coordinate system and units previously specified (also accessible from *getCoordSys*() and *getCoordUnit*() methods). Please consult the User's Guide document, "Supported Coordinate Systems" for more details; in particular, note the 'standard' ordering of the returned coordinate values for non-Cartesian coordinate systems. When using an accumulation mode other than 'None', *all* coordinate values will be zero.

vvdPitchAngles – returned 2-dimensional vector of associated pitch angles; will be empty if omni-directional.[time,direction]

vvvdData – returned 3-dimensional vector of the specified data values.[time,energy|depth,direction]

bFluxAccumAvg – optional flag (ignored when Data Type is other than 'flux') for returning *flux* accumulation average values, if *true*. When specified as *false,* or omitted, returns *flux* values at the ephemeris input times and positions.

Return value: int – 0 = success, otherwise error

int getAggregData

(const std::string& strDataType,
const std::string& strFluxMode,
const int& iPercent,
dvector& vdTimes,
dvector& vdCoord1,
dvector& vdCoord2,
dvector& vdCoord3,
vdvector& vvdPitchAngles,
vvdvector& vvvdData,
bool bFluxAccumAvg = *false*)

Usage: Returns the confidence level results from multiple scenarios of data input, for the specified data type, flux mode and confidence level percent. The associated ephemeris and possibly pitch angles are also returned.

Parameters:

strDataType – data type identifier: "flux"|"fluence"|"doserate"|"doseaccum"

strFluxMode – flux mode identifier: "perturbed"|"montecarlo"

iPercent – aggregation confidence level percent (0-100 *or -1 for mean*). The number specified must be one of those included in previous calls to the *setAggregConfLevel()* and related methods.

vdTimes – returned vector of ephemeris time values, in Modified Julian Date form; when using an accumulation mode other than 'None', this time specifies the *ending* time of the accumulation interval.

vdCoord1, vdCoord2, vdCoord3 – returned vectors of ephemeris position values, in the coordinate system and units previously specified (also accessible from *getCoordSys*() and *getCoordUnit*() methods). Please consult the User's Guide document, "Supported Coordinate Systems" for more details; in particular, note the ordering of the returned coordinate values for non-Cartesian coordinate systems. When using an accumulation mode other than 'None', *all* coordinate values will be zero.

vvdPitchAngles – returned vector of associated pitch angles; will be empty if omni-directional.

vvvdData – returned 3-dimensional vector of the specified data values.[time,energy|depth,direction]

bFluxAccumAvg – optional flag (ignored when Data Type is other than 'flux') for returning *flux* accumulation average values, if *true*. When specified as *false,* or omitted, returns *flux* values at the ephemeris input times and positions.

Return value: int – 0 = success, otherwise error

Utility methods for your convenience

int reduceDataDimension

(const vvdvector& vvvdData,
vdvector& vvdData)

Usage: Utility method for changing 3-dimensional return vectors to 2-dimension; this is useful only for simplifying the model results from omni-direction flux calculations (or uni-directional flux calculations where only a *single* pitch angle has been specified).

Parameters:

vvvdData – 3-dimensional model data vector.[time,energy|depth,direction]

vvdData – returned 2-dimension model data vector.[time,energy|depth]

Return value: int – 0 = success, otherwise error

void resetModelData

Usage: Resets the data access to the model output files generated during the most recent model run. Subsequent calls to the various *flyin[]*() and *get[]*() data access methods will return data starting at the beginning of the ephemeris input times and positions.

Parameters: -none-

Return value: -none-

int resetModelRun

(bool bDelBinDir = *true*,
bool dResetParam = *false*)

Usage: Performs cleanup of the most recent model run. This method is needed only if further model run calculations are to be performed again *using the same object*, with new or revised input parameter settings.

Parameters:

bDelBinDir – optional flag for the deletion the temporary binary directory containing the most recent model run output files. If *true* (or omitted), the directory is removed. This argument *must* be specified if the second argument is also specified.

bResetParam – optional flag for the reset of all previously-specified model run input parameters. If specified as *true*, the input parameters are reset to their initial default values. No change if omitted.

Return value: int – 0 = success, otherwise error

int validateParameters

Usage: Performs a validation of all input parameters, verifying that there are no conflicts between the various settings and that all required values have been specified. *Use of this method is optional*, as it is called internally by the *runModel*() method.

Parameters: -none-

Return value: int – 0 = success; >0: number of errors detected

void resetParameters

Usage: Resets *all* model run input parameters to their initial default values.

Parameters: -none-

Return value: -none-

Time Conversion Utilities:

These are utility methods for the conversion between Modified Julian Date values and other date and time format values.

double getGmtSeconds

(const int& iHours,
  const int& iMinutes,
  const double& dSeconds)

Usage: Determines the GMT seconds of day for the input hours, minutes and seconds.
Parameters:
iHours – hours of day (0-23)
iMinutes – minutes of hour (0-59)
dSeconds – seconds of minute (0-59.999)
Return value: double – GMT seconds of day

int getDayOfYear

(const int& iYear,
  const int& iMonth,
  const int& iDay)

Usage: Determines the day number of year for the input year, month and day number.
Parameters:
iYear – year (1950-2049)
iMonth – month (1-12)
iDay – day of month (1-28|29|30|31)
Return value: int – day number of year

double getModifiedJulianDate

(const int& iYear,
  const int& iDdd,
  const double& dGmtsec)

Usage: Determines the Modified Julian Date for the input year, day of year and GMT seconds.
Parameters:
iYear – year (1950-2049)
iDdd – day of year (1-365|366)
dGmtsec – GMT seconds of day (0-86399.999)
Return value: double – Modified Julian Date (33282.0 - 69806.999)

double getModifiedJulianDate

(const int& iUnixTime)

Usage: Determines the Modified Julian Date for the input UNIX time value.
(*due to limitations of Unix time, this will be valid only between 01 Jan 1970 – 19 Jan 2038*).
Parameters:
iUnixTime – Unix Time, in seconds from 01 Jan 1970, 0000 GMT; (0 – MaxInt)
Return value: double – Modified Julian Date (40587.0 - 65442.134)

int getDateTime

(const double& dModJulDate,
int& iYear,
int& iDdd,
double& dGmtsec)

Usage: Determines the year, day of year and GMT seconds for the input Modified Julian Date.
Parameters:
dModJulDate – Modified Julian Date (33282.0 - 69806.999)
iYear – returned year (1950-2049)
iDdd – returned day of year (1-365|366)
dGmtsec – returned GMT seconds of day (0-86399.999)
Return value: int – 0 = success, otherwise error

int getDateTime

(const double& dModJulDate,
int* piYear,
int* piDdd,
double* pdGmtsec)

Usage: Determines the year, day of year and GMT seconds for the input Modified Julian Date.
Parameters:
dModJulDate – Modified Julian Date value (33282.0 - 69806.999)
piYear – *pointer* to returned year (1950-2049)
piDdd – *pointer* to returned day of year (1-365|366)
pdGmtsec – *pointer* to returned GMT seconds of day (0-86399.999)
Return value: int – 0 = success, otherwise error

int getHoursMinSec

(const double& dGmtsec,
int& iHours,
int& iMinutes,
double& dSeconds)

Usage: Determines the hours, minutes and seconds for the input GMT seconds.
Parameters:
dGmtsec – GMT seconds of day (0-86399.999)
iHours – returned hours of day (0-23)
iMinutes – returned minutes of hour (0-59)
dSeconds – returned seconds of minute (0-59.999)
Return value: int – 0 = success, otherwise error

int getHoursMinSec

(const double& dGmtsec,
int* piHours,
int* piMinutes,
double* pdSeconds)

Usage: Determines the hours, minutes and seconds for the input GMT seconds.
Parameters:
dGmtsec – GMT seconds of day (0-86399.999)

piHours – *pointer* to returned hours of day (0-23)
piMinutes – *pointer* to returned minutes of hour (0-59)
pdSeconds – *pointer* to returned seconds of minute (0-59.999)
Return value: int – 0 = success, otherwise error

int getMonthDay

(const int& iYear,
const int& iDdd,
int& iMonth,
int& iDay)
Usage: Determines the month and day number for the input year and day of year.
Parameters:
iYear – year (1950-2049)
iDdd – day of year (1-365|366)
iMonth – returned month (1-12)
iDay – returned day of month (1-28|29|30|31)
Return value: int – 0 = success, otherwise error

int getMonthDay

(const int& iYear,
const int& iDdd,
int* piMonth,
int* piDay)
Usage: Determines the month and day number for the input year and day of year.
Parameters:
iYear – year (1950-2049)
iDdd – day of year (1-365|366)
piMonth – *pointer* to returned month (1-12)
piDay – *pointer* to returned day of month (1-28|29|30|31)
Return value: int – 0 = success, otherwise error

C++ Model-Level API Reference

These classes provide direct programmatic access to each of the model/processing components that comprise the CmdLineAe9Ap9 application. There is no parallelized processing available at this level. Error-checking is limited to within each class (no error-checking between classes), and so is unable to detect incompatible processing operations (ie uni-directional integral flux input to dose calculations). Computer system environment variables may be used when specifying database filenames.

EphemModel Class

Header file: CEphemModel.h

This class is the entry point that provides direct programmatic access to the ephemeris generation model.
Please note that all time values, both input and output, are in Modified Julian Date (MJD) form. Conversions to and from MJD times are available from the DateTime class, described elsewhere in this Model-Level API section. Position coordinates are always used in sets of three values, in the coordinate system and units that are specified. Please consult the User's Guide document, "Supported Coordinate Systems" for more details; in particular, note the order of the coordinate values for non-Cartesian coordinate systems.
Ephemeris generation requires the selection of an orbit propagator (and its options), a time range and time step size, and the definition of the orbit characteristics. These orbit characteristics may be defined by either a Two-Line Element (TLE) file, or a set of orbital element values. See the User's Guide '*Orbit Propagation Inputs*' section for details about each of the available settings.
The number of ephemeris entries produced by each call the *computeEphemeris*() method may be controlled by the sizing specified using the *setChunkSize*() method. When not specified, the default behavior is to produce ephemeris for the entire period as defined in the *setTimes*() method. For large sets of times, this could cause the ephemeris data to potentially occupy a sizeable amount of system memory, and potentially hinder subsequent processing.

General:

EphemModel

Usage: Default constructor
Parameters: -none-
Return values: -none-

~EphemModel

Usage: Destructor
Parameters: -none-
Return values: -none-

int setChunkSize

(const int& iChunkSize)

Usage: Specifies the number of time and position entries that are returned from each call to the *computeEphemeris*() methods. This is useful for breaking up the ephemeris data into manageable segments for its use in other calculations. Recommended sizing is 240, or 120 on systems with limited

available memory resources.
When a sizing is *not* specified, it defaults to 0, meaning the *entire set* of times specified in the *setTimes*() methods will be calculated and returned in a *single* call to the *computeEphemeris*() methods. For large sets of times, this could cause this data to potentially occupy a sizeable amount of system memory, and potentially hinder subsequent processing.

Parameters:

iChunkSize – number of entries in processing chunk; values lower than 60 are not recommended

Return value: int – 0 = success, otherwise error

int getChunkSize

Usage: Returns the current value of the 'chunk' size, as described in previous method.

Parameters: -*none*-

Return value: number of entries in processing chunk

Model Parameter Inputs:

int setMagfieldDBFile

(const std::string& strDataSource)

Usage: Specifies the name of the file (including path) for the magnetic field model database. This database name is in the form of '<path>/igrfDB.h5'.

Parameters:

strDataSource – magnetic field model database filename, including path

Return value: int – 0 = success, otherwise error

int setPropagator

(const std::string& strPropSpec)

Usage: Specifies the orbit propagator algorithm to use for the ephemeris generation.

Parameters:

strPropSpec - valid values: 'SatEph', 'SGP4' or 'Kepler'.

Return value: int – 0 = success, otherwise error

int setSGP4Param

(const std::string& strMode,
const std::string& strWGS)

Usage: Specifies the mode and WGS parameter for the SGP4 orbit propagator, if being used.

Parameters:

strMode – SGP4 propagation mode; valid values: 'Standard' or 'Improved'.

strWGS – World Geodetic System version; valid values: '72old', '72' or '84'.

Return value: int – 0 = success, otherwise error

void setKeplerUseJ2

(bool bUseJ2)

Usage: Specifies the use of the 'J2' perturbation for the Kepler orbit propagator, if being used.
Parameters:
bUseJ2 – *true* or *false*
Return value: -none-

int setTimes

(const dvector& vdTimes)
Usage: Specifies a specific set of time values for the ephemeris information to be generated by the orbit propagator from the specified orbital element values or TLE file.
Parameters:
vdTimes – vector of chronologically-ordered time values, in Modified Julian Date form
Return value: int – 0 = success, otherwise error

int setTimes

(const double& dStartTime,
const double& dEndTime,
const double& dTimeStepSecs)
Usage: Specifies the start and stop times (*inclusive*), and time step, of the ephemeris information to be generated by the orbit propagator from the specified orbital element values or TLE file.
Parameters:
dStartTime, *dEndTime* - start and stop time values, in Modified Julian Date form
dTimeStepSecs - time step size, in seconds
Return value: int – 0 = success, otherwise error

TLE files are required to be in the standard NORAD format (see User's Guide, Appendix F). Use of the Kepler propagator requires that the TLE file contain only one entry. For the other propagators, the TLE may contain multiple entries (for the same satellite), which must be in chronological order.

int setTLEFile

(const std::string& strTLEFile)
Usage: Specifies the name of the Two-Line Element (TLE) file (including path) to use with the selected orbit propagator; this parameter is not needed if a set of orbital element values are being used instead.
Parameters:
strTLEFile – path and filename of TLE file
Return value: int – 0 = success, otherwise error

The orbital element values to be specified depend on the type of orbit and/or available orbit definition references. Their use requires an associated element time to also be specified. See the User's Guide document "Orbiter Propagation Inputs" section for more details.

int setElementTime

(const double& dElementTime)
Usage: Specifies the 'epoch' time associated with the set of orbital element values.
Parameters:
dElementTime – time, in Modified Julian Date form
Return value: int – 0 = success, otherwise error

int setInclination

(const double& dInclination)
Usage: Specifies the orbital element 'Inclination' value.
Parameters:
dInclination – orbit inclination angle, in degrees (0-180)
Return value: int – 0 = success, otherwise error

int setRightAscension

(const double& dRtAscOfAscNode)
Usage: Specifies the orbital element 'Right Ascension of the Ascending Node' value.
Parameters:
dRtAscOfAscNode – orbit ascending node position, in degrees (0-360)
Return value: int – 0 = success, otherwise error

int setEccentricity

(const double& dEccentricity)
Usage: Specifies the orbital element 'Eccentricity' value.
Parameters:
dEccentricity – orbit eccentricity value, unitless (0-1)
Return value: int – 0 = success, otherwise error

int setArgOfPerigee

(const double& dArgOfPerigee)
Usage: Specifies the orbital element 'Argument of Perigee' value.
Parameters:
dArgOfPerigee – orbit perigee position, in degrees (0-360)
Return value: int – 0 = success, otherwise error

int setMeanAnomaly

(const double& dMeanAnomaly)
Usage: Specifies the orbital element 'Mean Anomaly' value.
Parameters:
dMeanAnomaly – orbit mean anomaly value, in degrees (0-360)
Return value: int – 0 = success, otherwise error

int setMeanMotion

(const double& dMeanMotion)
Usage: Specifies the orbital element 'Mean Motion' value.
Parameters:
dMeanMotion – orbit mean motion value, in units of revolutions per day (>0)
Return value: int – 0 = success, otherwise error

int setMeanMotion1stDeriv

(const double& dMeanMotion1stDeriv)
Usage: Specifies the orbital element 'First Time Derivative of the Mean Motion' value (this should NOT be divided by 2, as when specified in a TLE); this value is only used by the SatEph propagator.

Parameters:
dMeanMotion1stDeriv – first derivative of mean motion, in units of revs per day^2
Return value: int – 0 = success, otherwise error

int setMeanMotion2ndDeriv

(const double& dMeanMotion2ndDeriv)
Usage: Specifies the orbital element 'Second Time Derivative of the Mean Motion' value (this should NOT be divided by 6, as when specified in a TLE); this value is only used by the SatEph propagator.
Parameters:
dMeanMotion2ndDeriv – second derivative of mean motion, in units of revs per day^3
Return value: int – 0 = success, otherwise error

int setBStar

(const double& dBStar)
Usage: Specifies the orbital element 'B*' value, for modelling satellite drag effects; this value is only used by the SGP4 propagator.
Parameters:
dBStar – ballistic coefficient value
Return value: int – 0 = success, otherwise error

int setAltitudeOfApogee

(const double& dAltApogee)
Usage: Specifies the orbital element 'Apogee Altitude' value (furthest distance).
Parameters:
dAltApogee – altitude (in km) above the Earth's surface at the orbit's apogee
Return value: int – 0 = success, otherwise error

int setAltitudeOfPerigee

(const double& dAltPerigee)
Usage: Specifies the orbital element 'Perigee Altitude' value (closest distance).
Parameters:
dAltPerigee – altitude (in km) above the Earth's surface at the orbit's perigee
Return value: int – 0 = success, otherwise error

int setLocalTimeOfApogee

(const double& dLocTimeApogee)
Usage: Specifies the local time of the orbit's apogee.
Parameters:
dLocTimeApogee – local time, in hours (0-24)
Return value: int – 0 = success, otherwise error

int setLocalTimeMaxInclination

(const double& dLocTimeMaxIncl)
Usage: Specifies the local time of the orbit's maximum inclination (ie max latitude).
Parameters:
dLocTimeMaxIncl – local time, in hours (0-24)

Return value: int – 0 = success, otherwise error

int setTimeOfPerigee

(const double& dTimeOfPerigee)

Usage: Specifies the time of the orbit's perigee, as an alternative to the Mean Anomaly specification. *Any Mean Anomaly value also specified will be overridden by this value*.

Parameters:

dTimeOfPerigee – time, in Modified Julian Date form, for orbit perigee

Return value: int – 0 = success, otherwise error

int setSemiMajorAxis

(const double& dSemiMajorAxis)

Usage: Specifies the orbit's semi-major axis length.

Parameters:

dSemiMajorAxis – semi-major axis length, in units of Re (radius of Earth = 6371.2 km)

Return value: int – 0 = success, otherwise error

int setGeosynchLon

(const double& dGeosynchLon)

Usage: Specifies the geographic longitude of satellite in a geosynchronous orbit

Parameters:

dGeosynchLon – longitude, in degrees (-180 – 360)

Return value: int – 0 = success, otherwise error

int setPositionGEI

(const double& dX,
const double& dY,
const double& dZ)

Usage: Specifies the satellite's position in the GEI coordinate system at the element's 'epoch' time. This must be used in conjunction with the *setVelocityGEI*() method.

Parameters:

dX, dY, dZ – GEI coordinate system satellite position values, in km

Return value: int – 0 = success, otherwise error

int setVelocityGEI

(const double& dXdot,
const double& dYdot,
const double& dZdot)

Usage: Specifies the satellite's velocity in GEI coordinate system at the element's 'epoch' time. This must be used in conjunction with the *setPositionGEI*() method.

Parameters:

dXdot, dYdot, dZdot – GEI coordinate system satellite velocity values, in km/sec

Return value: int – 0 = success, otherwise error

void resetOrbitInputs

Usage: Resets all parameters that specify the orbit definition to their default values. These include the various orbital element values, element time, state vectors and TLE specifications.

Parameters: -none-

Return value: -none-

Model Execution and Results:

The ephemeris computation requires that a propagator model be selected, the magnetic field database be specified (used for coordinate conversions), an ephemeris generate time range and time step defined, and the orbit described by either a TLE file or an appropriate set of element values & time.

int computeEphemeris

(dvector& vdTimes,
dvector& vdXGEI,
dvector& vdYGEI,
dvector& vdZGEI,
dvector& vdXDotGEI,
dvector& vdYDotGEI,
dvector& vdZDotGEI)

Usage: Returns the generated ephemeris information in the GEI coordinate system, for the current time segment. The number of entries that are returned from each call to the *computeEphemeris*() method is specified using the *setChunkSize*() method. If this 'chunk' size is not specified, or set to '0', the ephemeris for the entire time period, defined by the *setTimes*() methods, is returned in a single call. If a great number of ephemeris entries are requested in a single call, the amount of memory required (automatically allocated) may become excessive and/or hinder further processing. When a non-zero 'chunk' size is specified, multiple calls to this method may be required for accessing the ephemeris information for the entire time period; however, this provides the ephemeris information in manageable segments, for its use in other subsequent calculation tasks. This segmentation enables the processing performance to be tuned to the available system memory.

Parameters:

vdTimes – returned vector of ephemeris time values, in Modified Julian Date form

vdXGEI, vdYGEI, vdZGEI – returned vectors of ephemeris position values, in the 'GEI' coordinate system and units of 'km'

vdXDotGEI, vdYDotGEI, vdZDotGEI – returned vectors of ephemeris velocity values, in the 'GEI' coordinate system and units of 'km/sec'

Return value: int – Number of ephemeris records returned (≥0 = success, <0 = error)

int computeEphemeris

(const std::string& strCoordSys,
const std::string& strCoordUnits,
dvector& vdTimes,
dvector& vdCoord1,
dvector& vdCoord2,
dvector& vdCoord3)

Usage: Returns the generated ephemeris information in the coordinate system and units specified, for the current time segment. The number of entries that are returned from each call to the *computeEphemeris*() method is specified using the *setChunkSize*() method. If this 'chunk' size is not specified, or set to '0', the ephemeris for the entire time period, defined by the *setTimes*() methods, is returned in a single call. If a great number of ephemeris entries are requested in a single call, the amount of memory required (automatically allocated) may become excessive and/or hinder further processing. When a non-zero 'chunk' size is specified, multiple calls to this method may be required for accessing the ephemeris information for the entire time period; however, this provides the ephemeris information in manageable segments, for its use in other subsequent calculation tasks. This segmentation enables the processing performance to be tuned to the available system memory.

Parameters:

strCoordSys – coordinate system identifier: 'GEI','GEO','GDZ','GSM','GSE','SM','MAG','SPH' or 'RLL'; Please consult the User's Guide document, "Supported Coordinate Systems" for more details.

strCoordUnits - 'km' or 'Re'; 'GDZ' is set to always use 'km' for the altitude value. 1 Re = 6371.2 km.

vdTimes – returned vector of ephemeris time values, in Modified Julian Date form

vdCoord1, vdCoord2, vdCoord3 – returned vectors of ephemeris position values, in the coordinate system and units specified.

Please consult the User's Guide document, "Supported Coordinate Systems" for more details; in particular, note the 'standard' ordering of the returned coordinate values for non-Cartesian coordinate systems.

Return value: int – Number of ephemeris records returned (≥0 = success, <0 = error)

void restartEphemeris

Usage: When the 'chunk' size, specified using the *setChunkSize*() method, is larger than 0, this method explicitly resets the *computeEphemeris*() methods to begin the ephemeris generation at the previously-defined 'start time' at their next call. This reset is done automatically when the chunk size is changed, or any of the orbital element parameters or propagator settings is modified.

Parameters: -none-

Return value: -none-

int convertCoordinates

```
( const std::string& strCoordSys,
  const std::string& strCoordUnits,
  const dvector& vdTimes,
  const dvector& vdCoord1,
  const dvector& vdCoord2,
  const dvector& vdCoord3,
  const std::string& strNewCoordSys,
  const std::string& strNewCoordUnits,
  dvector& vdNewCoord1,
  dvector& vdNewCoord2,
  dvector& vdNewCoord3 )
```

Usage: Converts the set of input times, position coordinates from one coordinate system to another.

Parameters:

strCoordSys – coordinate system identifier: 'GEI','GEO','GDZ','GSM','GSE','SM','MAG','SPH' or 'RLL'; Please consult the User's Guide document, "Supported Coordinate Systems" for more details.

strCoordUnits - 'km' or 'Re'; 'GDZ' is set to always use 'km' for the altitude value.

vdTimes – vector of time values, in Modified Julian Date form

vdCoord1, vdCoord2, vdCoord3 –vectors of position values, in the coordinate system and units specified by the *strCoordSys* and *strCoordUnit* parameter values.

Consult the User's Guide document, "Supported Coordinate Systems" for more details; in particular, note the expected 'standard' ordering of the input and output coordinate values for non-Cartesian coordinate systems.

strNewCoordSys – 'new' coordinate system identifier: 'GEI','GEO','GDZ','GSM','GSE','SM','MAG','SPH' or 'RLL'; Please consult the User's Guide document, "Supported Coordinate Systems" for more details.

strNewCoordUnits – 'new' units: 'km' or 'Re'; 'GDZ' is set to always use 'km' for the altitude value.

vdNewCoord1, vdNewCoord2, vdNewCoord3 –vectors of position values, in the 'new' coordinate system and units specified by the *strNewCoordSys* and *strNewCoordUnit* parameter values.

Return value: int – 0 = success, otherwise error

int convertCoordinates

```
( const std::string& strCoordSys,
  const std::string& strCoordUnits,
  const double& dTime,
  const double& dCoord1,
  const double& dCoord2,
  const double& dCoord3,
  const std::string& strNewCoordSys,
  const std::string& strNewCoordUnits,
  double& dNewCoord1,
  double& dNewCoord2,
  double& dNewCoord3 )
```

Usage: Converts a single input time, position coordinates from one coordinate system to another.

Parameters:

strCoordSys – coordinate system identifier: 'GEI','GEO','GDZ','GSM','GSE','SM','MAG','SPH' or 'RLL'; Please consult the User's Guide document, "Supported Coordinate Systems" for more details.

strCoordUnits - 'km' or 'Re'; 'GDZ' is set to always use 'km' for the altitude value.

dTimes – time value, in Modified Julian Date form

dCoord1, dCoord2, dCoord3 –position values, in the coordinate system and units specified by the *strCoordSys* and *strCoordUnit* parameter values.

Consult the User's Guide document, "Supported Coordinate Systems" for more details; in particular, note the expected 'standard' ordering of the input and output coordinate values for non-Cartesian coordinate systems.

strNewCoordSys – 'new' coordinate system identifier: 'GEI','GEO','GDZ','GSM','GSE','SM','MAG','SPH' or 'RLL'; Please consult the User's Guide document, "Supported Coordinate Systems" for more details.

strNewCoordUnits – 'new' units: 'km' or 'Re'; 'GDZ' is set to always use 'km' for the altitude value.

dNewCoord1, dNewCoord2, dNewCoord3 – position values, in the 'new' coordinate system and units specified by the *strNewCoordSys* and *strNewCoordUnit* parameter values.

Return value: int – 0 = success, otherwise error

Ae9Ap9Model Class

Header file: Ae9Ap9Model.h

This class is the entry point that provides direct programmatic access to the Ae9, Ap9 and SPM model. Please note that all time values, both input and output, are in Modified Julian Date (MJD) form. Conversions to and from MJD times are available from the DateTime class, described elsewhere in this Model-Level API section. Position coordinates are always used in sets of three values, in the coordinate system and units that are specified. Please consult the User's Guide document, "Supported Coordinate Systems" for more details; in particular, note the order of the coordinate values for non-Cartesian coordinate systems.

General:

Ae9Ap9Model

Usage: Default constructor.
Important: This class uses the 'ae9ap9' namespace. This namespace qualifier is required for its object variable declaration: ie `ae9ap9::Ae9Ap9Model ae9ap9Obj;`
Parameters: -none-
Return values: -none-

~Ae9Ap9Model

Usage: Destructor
Parameters: -none-
Return values: -none-

Model Parameter Inputs:

int setModelDBFile

(const std::string& strModelDBFile)
Usage: Specifies the name of the database file (including path) for flux model calculations.
Please consult the User's Guide for the exact database filename associated with each model.
Once initialized via calls to either *loadModelDB*() or *setFluxEnvironment*() methods, the specified model database *cannot* be changed. For best results, use separate model objects for different model species.
Parameters:
strModelDBFile – model database filename, including path
Return value: int – 0 = success, otherwise error

int setKPhiNNetDBFile

(const std::string& strKPhiNNetDBFile)
Usage: Specifies the name of the file (including path) for the K/Phi neural network database.
This database name is in the form of '<path>/fastPhi_net.mat'.
Parameters:
strDataSource – neural network database filename, including path
Return value: int – 0 = success, otherwise error

int setKHMinNNetDBFile

(const std::string& strKHMinNNetDBFile)

Usage: Specifies the name of the file (including path) for the K/Hmin neural network database. This database name is in the form of '<path>/fast_hmin_net.mat'.

Parameters:

strDataSource – neural network database filename, including path

Return value: int – 0 = success, otherwise error

int setMagfieldDBFile

(const std::string& strMagfieldDBFile)

Usage: Specifies the name of the file (including path) for the magnetic field model database. This database name is in the form of '<path>/igrfDB.h5'.

Parameters:

strDataSource – magnetic field model database filename, including path

Return value: int – 0 = success, otherwise error

int loadModelDB

Usage: Performs the initial loading of information from the model database file specified in the *setModelDBFile*() method. Use of this method is optional, as it is called internally on the initial call to the *setFluxEnvironment*() method. However, it is required if the *getModel[Name|Species]*() methods are called before the initial call to *setFluxEnvironment*().
Once initialized, these model databases specified <u>*cannot*</u> be changed.

Parameters: -none-

Return value: int – 0 = success, otherwise error

std::string getModelName

Usage: Returns the name of the model described by the model database file specified in the previous call to *setModelDBFile*() method. This returned model name is available only after a call to either the *loadDB*() or *setFluxEnvironment*() methods .

Parameters: -none-

Return value: std:string – name of model associated with database ('AE9', 'AP9', 'SPME', 'SPMHE' or 'SPMO')

std::string getModelSpecies

Usage: Returns the name of the particle species of the model database file specified in the previous call to *setModelDBFile*() method. This returned species name is available only after a call to either the *loadDB*() or *setFluxEnvironment*() methods .

Parameters: -none-

Return value: std:string – name of species of the associated with database ('e-', 'H+', 'He+' or 'O+')

Model Execution and Results:

The *setFluxEnvironment*() method is used to specify the ephemeris (time and position) and particle energies for the flux calculation of the Ae9Ap9 model. The multiple versions of this method provide different ways to define the flux particle direction(s) – omnidirection (default), fixed (over time) or variable pitch angles, or explicit direction vectors. The various *flyin[type]* () (or *computeFlux[type]* ()) methods return the respective types of flux values using the most recently defined 'flux environment' specifications.

For best model performance, it is recommended that the amount of ephemeris information being supplied as input to the *setFluxEnvironment*() method be moderated. This method performs numerous calculations on each time, position coordinate and pitch angle(s) combination, retaining these intermediate results in additional internally-allocated memory. To avoid stressing the system memory resources, limit the number of entries in the time and coordinate vectors for each call: a value of 120 is advised for systems with limited memory, 240 for typical systems, and 480 for systems with larger amounts available memory. The subsequent call to the needed *flyin[type]* () method(s) will return fluxes for the current ephemeris segment.

The ephemeris information produced by the *EphemModel::computeEphemeris*() can be segmented through the use of that class's *setChunkSize*() method. The segmentation of ephemeris information from other sources will need to be accomplished by the calling process.

int setFluxEnvironment

```
( const std::string& strFluxType,
  const dvector& vdEnergies,
  const dvector& vdEnergies2,
  const dvector& vdTimes,
  const std::string& strCoordSys,
  const std::string& strCoordUnits,
  const dvector& vdCoords1,
  const dvector& vdCoords2,
  const dvector& vdCoords3 )
```

Usage: Specifies the flux type, energies and ephemeris time and positions to be used for *omni-directional* flux model calculations. Note that use of '2PtDiff' flux type requires that both sets of energy values to be specified. Please consult User's Guide for the valid energy ranges/values, which depend on the model being used.

Parameters:

strFluxType – flux type identifier: '1PtDiff', '2PtDiff' or 'Integral'

vdEnergies – vector of energy values; if '2PtDiff' flux type, these specify lower bounds of energy bins.

vdEnergies2 – ignored unless flux type is '2ptDiff'; vector of energy values, specifying upper bounds of energy bins

vdTimes - vector of time values, in Modified Julian Date form. May be identical times or times in chronological order, associated with position coordinates.

strCoordSys – coordinate system identifier: 'GEI','GEO','GDZ','GSM','GSE','SM','MAG','SPH' or 'RLL'; Please consult the User's Guide document, "Supported Coordinate Systems" for more details.

strCoordUnits - 'km' or 'Re'; 'GDZ' is set to always use 'km' for the altitude value.

vdCoords1, *vdCoords2*, *vdCoords3* - vectors of position coordinate values associated with times vector. These position values are assumed to be in the coordinate system and units specified by the strCoordSys and strCoordUnits parameters. Please consult the User's Guide document, "Supported

Coordinate Systems" for more details; in particular, note the expected 'standard' order of the coordinate values for non-Cartesian coordinate systems.

Return value: int – 0 = success, otherwise error

int setFluxEnvironment

(const std::string& strFluxType,
const dvector& vdEnergies,
const dvector& vdEnergies2,
const dvector& vdTimes,
const std::string& strCoordSys,
const std::string& strCoordUnits,
const dvector& vdCoords1,
const dvector& vdCoords2,
const dvector& vdCoords3,
const dvector& vdPitchAngles)

Usage: Specifies the flux type, energies and ephemeris time and positions, with a *fixed* set of pitch angles, to be used for *uni-directional* flux model calculations. Note that use of '2PtDiff' flux type requires that both sets of energy values to be specified. Please consult User's Guide for the valid energy ranges/values, which depend on the model being used.

Parameters:

strFluxType – flux type identifier: '1PtDiff', '2PtDiff' or 'Integral'

vdEnergies – vector of energy values; if '2PtDiff' flux type, these specify lower bounds of energy bins.

vdEnergies2 – ignored unless flux type is '2ptDiff'; vector of energy values, specifying upper bounds of energy bins

vdTimes - vector of time values, in Modified Julian Date form. May be identical times or times in chronological order, associated with position coordinates.

strCoordSys – coordinate system identifier: 'GEI','GEO','GDZ','GSM','GSE','SM','MAG','SPH' or 'RLL'; Please consult the User's Guide document, "Supported Coordinate Systems" for more details.

strCoordUnits - 'km' or 'Re'; 'GDZ' is set to always use 'km' for the altitude value.

vdCoords1, *vdCoords2*, *vdCoords3* - vectors of position coordinate values associated with times vector. These position values are assumed to be in the coordinate system and units specified by the strCoordSys and strCoordUnits parameters. Please consult the User's Guide document, "Supported Coordinate Systems" for more details; in particular, note the expected 'standard' order of the coordinate values for non-Cartesian coordinate systems.

vdPitchAngles – vector of pitch angles, in degrees (0-180); to be used at each time/position

Return value: int – 0 = success, otherwise error

int setFluxEnvironment

(const std::string& strFluxType,
const dvector& vdEnergies,
const dvector& vdEnergies2,
const dvector& vdTimes,
const std::string& strCoordSys,
const std::string& strCoordUnits,
const dvector& vdCoords1,
const dvector& vdCoords2,
const dvector& vdCoords3,

const vdvector& vvdPitchAngles)

Usage: Specifies the flux type, energies and ephemeris time and positions, with a *varying* set of pitch angles, to be used for *uni-directional* flux model calculations. Note that use of '2PtDiff' flux type requires that both sets of energy values to be specified. Please consult User's Guide for the valid energy ranges/values, which depend on the model being used.

Parameters:

strFluxType – flux type identifier: '1PtDiff', '2PtDiff' or 'Integral'

vdEnergies – vector of energy values; if '2PtDiff' flux type, these specify lower bounds of energy bins.

vdEnergies2 – ignored unless flux type is '2ptDiff'; vector of energy values, specifying upper bounds of energy bins

vdTimes - vector of time values, in Modified Julian Date form. May be identical times or times in chronological order, associated with position coordinates.

strCoordSys – coordinate system identifier: 'GEI','GEO','GDZ','GSM','GSE','SM','MAG','SPH' or 'RLL'; Please consult the User's Guide document, "Supported Coordinate Systems" for more details.

strCoordUnits - 'km' or 'Re'; 'GDZ' is set to always use 'km' for the altitude value.

vdCoords1, *vdCoords2*, *vdCoords3* - vectors of position coordinate values associated with times vector. These position values are assumed to be in the coordinate system and units specified by the strCoordSys and strCoordUnits parameters. Please consult the User's Guide document, "Supported Coordinate Systems" for more details; in particular, note the expected 'standard' order of the coordinate values for non-Cartesian coordinate systems.

v*dPitchAngles* – 2-dimensional vector of pitch angles, in degrees (0-180). [time,pitch angles]

Return value: int – 0 = success, otherwise error

int setFluxEnvironment

(const std::string& strFluxType,
const dvector& vdEnergies,
const dvector& vdEnergies2,
const dvector& vdTimes,
const std::string& strCoordSys,
const std::string& strCoordUnits,
const dvector& vdCoords1,
const dvector& vdCoords2,
const dvector& vdCoords3,
const dvector& vdFluxDir1,
const dvector& vdFluxDir2,
const dvector& vdFluxDir3)

Usage: Specifies the flux type, energies and ephemeris time and positions, at a set of *varying* direction vectors, to be used for *uni-directional* flux model calculations. Note that use of '2PtDiff' flux type requires that both sets of energy values to be specified. Please consult User's Guide for the valid energy ranges/values, which depend on the model being used.

Parameters:

strFluxType – flux type identifier: '1PtDiff', '2PtDiff' or 'Integral'

vdEnergies – vector of energy values; if '2PtDiff' flux type, these specify lower bounds of energy bins.

vdEnergies2 – ignored unless flux type is '2ptDiff'; vector of energy values, specifying upper bounds of energy bins

vdTimes - vector of time values, in Modified Julian Date form. May be identical times or times in chronological order, associated with position coordinates.

strCoordSys – Cartesian coordinate system identifier: 'GEI','GEO','GSM','GSE','SM' or 'MAG';
Please consult the User's Guide document, "Supported Coordinate Systems" for more details.
strCoordUnits - 'km' or 'Re'
vdCoords1, *vdCoords2*, *vdCoords3* - vectors of position coordinate values associated with times vector. These position values are assumed to be in the coordinate system and units specified by the strCoordSys and strCoordUnits parameters. Please consult the User's Guide document, "Supported Coordinate Systems" for more details.
vdFluxDir1, *vdFluxDir2*, *vdFluxDir3* - vectors of direction vector values associated with times and position vectors. These direction values must be in the same Cartesian coordinate system and units as the position vectors.
Return value: int – 0 = success, otherwise error

int getPitchAngles

(vdvector& vvdPitchAngles)
Usage: Returns the set of uni-directional pitch angles used in the model calculations, corresponding to the direction vectors input to the fourth form of the *setFluxEnvironment*() method.
Parameters:
v*vdPitchAngles* – 2-dimensional vector of pitch angles, in degrees (0-180). [time,direction]
Return value: int – 0 = success, otherwise error

int flyinMean or ***computeFluxMean***

(vvdvector& vvvdFluxData)
Usage: Returns the 'Mean' model flux at the times, positions, energies (and possibly directions) specified in the most recent call to the *setFluxEnvironment*() method.
This same method may also be called as *computeFluxMean*(), with same exact arguments.
Parameters:
vvvdFluxData – returned 3-dimensional vector of the 'mean' flux values. [time,energy,direction]
Return value: int – 0 = success, otherwise error

int flyinMean

(vdvector& vvdFluxData)
Usage: Returns the 'Mean' model flux at the times, positions, energies specified in the most recent call to the *setFluxEnvironment*() method. This method may only be used when calculating *omnidirectional* fluxes.
Parameters:
vvvdFluxData – returned 2-dimensional vector of the 'mean' flux values. [time,energy]
Return value: int – 0 = success, otherwise error

int flyinPercentile or ***computeFluxPercentile***

(const int& iPercentile,
vvdvector& vvvdFluxData)
Usage: Returns the model flux results for the specified model Percentile number, at the times, positions, energies (and possibly directions) specified in the most recent call to the *setFluxEnvironment*() method.
This same method may also be called as *computeFluxPercentile*(), with same exact arguments.
Parameters:
iPercentile – percentile number of the flux values to be returned.

vvvdFluxData – returned 3-dimensional vector of the flux values for the specified percentile. [time,energy,direction]

Return value: int – 0 = success, otherwise error

int flyinPerturbedMean or ***computeFluxPerturbedMean***

(const int& iScenario,
vvdvector& vvvdFluxData)

Usage: Returns the model flux results for the specified Perturbed Mean scenario number, at the times, positions, energies (and possibly directions) specified in the most recent call to the *setFluxEnvironment*() method.
This same method may also be called as *computeFluxPerturbedMean*(), with same exact arguments.

Parameters:

iScenario – scenario number of the Perturbed Mean flux values to be returned.

vvvdFluxData – returned 3-dimensional vector of the flux values for the specified scenario number. [time,energy,direction]

Return value: int – 0 = success, otherwise error

int flyinScenario or ***computeFluxScenario***

(const double &dEpochTime,
const int& iScenario,
vvdvector& vvvdFluxData,
bool bPerturbFluxMap = *true*)

Usage: Returns the model flux results for the specified Monte Carlo scenario number, with the specified time progression reference time, at the times, positions, energies (and possibly directions) specified in the most recent call to the *setFluxEnvironment*() method.
This same method may also be called as *computeFluxScenario*(), with same exact arguments.

Parameters:

dEpochTime – Monte Carlo reference time, in Modified Julian Date form

iScenario – scenario number of the Monte Carlo flux values to be returned.

vvvdFluxData – returned 3-dimensional vector of the flux values for the specified scenario number. [time,energy,direction]

bPerturbFluxMap – optional flag, needed only for developmental testing, for controlling application of flux perturbations within calculations. Defaults to *true* when omitted (recommended).

Return value: int – 0 = success, otherwise error

AccumModel Class

Header file: CAccumModel.h

This class is the entry point that provides direct programmatic access to the accumulation model, which performs calculations on the flux data values over time. For the proper processing of the flux data, the *addToBuffer*() method is used collect data for each 'chunk'; the desired *compute**() methods *must be called each time* for that data to be properly processed for its contribution to the desired accumulation; these 'compute' calls may not necessarily return results at each call. Different types of accumulations of may be active simultaneously.

General:

AccumModel

Usage: Default constructor
Parameters: -none-
Return values: -none-

~AccumModel

Usage: Destructor
Parameters: -none-
Return values: -none-

Model Parameter Inputs:

void setTimeInterval

(const double& dVal)

Usage: Specifies the time duration of the accumulation of time-tagged data for use in the calculation of integrated data results. This defined interval (via this method or *setTimeIntervalSec*()) is used only with the *computeIntvFluence*(), *computeBoxcarFluence*() and *computeExponentialFlux*() methods. If unspecified, the interval duration defaults to 1 day (86400 seconds).

Parameters:
dVal – time duration, in units of days+fraction
Return value: -none-

void setTimeIntervalSec

(const double& dVal)

Usage: Specifies the time duration of the accumulation of time-tagged data for use in the calculation of the integrated data results. This defined interval (via this method or *setTimeInterval*()) is used only with the *computeIntvFluence*(), *computeBoxcarFluence*() and *computeExponentialFlux*() methods. If unspecified, the interval duration defaults to 86400 seconds (1 day).

Parameters:
dVal – time duration, in seconds; a negative value means the entire time range of the model run.
Return value: -none-

int setTimeIncrement

(const double& dVal)

Usage: Specifies the time delta for the shift of the 'Boxcar' accumulation mode time windows. This defined increment is used only with the *computeBoxcarFluence*() method.

Parameters:

dVal – time delta, in seconds, for the increment of time between the start of adjacent Boxcar time windows. Must be greater than zero, and less than the value specified in the *setTimeInterval*() method.

Return value: int – 0 = success, otherwise error

int setBufferSize

(const int& iSize)

Usage: Specifies the size (in data record counts) for the collection of input data. Use of this method is optional, as the size is automatically set when using the first form of the *addToBuffer*() method.

Parameters:

iSize – number of data records to collect in buffer of input data

Return value: int – 0 = success, otherwise error

Model Execution and Results:

int addToBuffer

(const dvector& vdTimes,
const vvdvector& vvvdData)

Usage: Loads the sets of input time-tagged data into the accumulation buffer. If the buffer size was not previously set (using the *setBufferSize*() method), it is sized to match the input data. Buffer sizes smaller than the size of the inputs will lead to data loss.

Parameters:

vdTimes – vector of time values, in Modified Julian Date form

vvvdData – 3-dimensional vector of data values to be loaded into buffer. [time,energy,direction]

Return value: int – 0 = success, otherwise error

int addToBuffer

(const double& dTime,
const vdvector& vvdData)

Usage: Loads a single set of input time-tagged data into the accumulation buffer. Use of this form may require the use of the *setBufferSize*() method.

Parameters:

dTime – time value, in Modified Julian Date form

vvvdData – 2-dimensional vector of data values to be loaded into buffer. [energy,direction]

Return value: int – 0 = success, otherwise error

int computeFluence

(dvector& vdFluenceTimes,
vvdvector& vvvdFluence)

Usage: Processes the current contents of the data buffer, then returns the cumulative time-integrated data results at the time tags of the current buffer contents. *Any specified accumulation time interval has no effect on these calculations.*

Parameters:

vdFluenceTimes – returned vector of times, in Modified Julian Date form

vvvdFluence – returned 3-dimensional vector of the calculated fluence values [time,energy,direction]

Return value: int – 0 or greater = number of data records returned in vectors; otherwise error

int computeIntvFluence

(dvector& vdFluenceTimes,
vvdvector& vvvdFluence,
ivector& viIntIndices,
bool bReturnPartial = *false*)

Usage: Processes the current contents of the data buffer, then returns the time-integrated data results from any *completed* accumulation intervals (whose duration is previously specified in calls to the *setTimeInterval[Sec]*() methods). Linear interpolation of the buffered data values is performed when their associated times do not line up with the start or stop times of the accumulation intervals. The returned times correspond to the end times of these completed accumulation intervals. The last argument should be omitted (or set to *false*) until all data has been processed; the subsequent call with the last argument set to *true* will return any remaining fluence data (for a partial interval period).

Parameters:

vdFluenceTimes – returned vector of times, in Modified Julian Date form, corresponding to the end of the intervals.

vvvdFluence – returned 3-dimensional vector of the calculated fluence values for zero or more completed accumulation intervals. [time,energy,direction]

viIntIndices – returned vector of the (nearest) indices to the current data buffer entries at which the completed accumulation intervals end. -1 means at or off end of buffer. This information is useful for the annotation of supplementary information associated with the buffered data, such as a time-varying pitch angles.

bReturnPartial – optional flag for returning the calculated fluence values for an incomplete accumulation interval, if any. Set to *true* only after *all* data has been processed.

Return value: int – 0 or greater = number of data records returned in vectors; otherwise error

int accumIntvFluence

(const dvector& vdFluenceTimes,
const vvdvector& vvvdFluence,
vvdvector& vvvdFluenceIntvAccum,
bool bAccumReset = *false*)

Usage: Sums the input fluence values over time, returning the cumulative fluence values since the initial call or the last reset. This method is intended to be used in tandem with 'Interval' accumulation fluence results from the *computeIntvFluence*() method.

Parameters:

vdFluenceTimes – vector of times, in Modified Julian Date form

vvvdFluence – 3-dimensional vector of the previously-calculated fluence values. [time,energy,direction]

vvvdFluenceIntvAccum – returned 3-dimensional vector of the corresponding *cumulative* fluence values since the initial call to this routine, or a reset was indicated.

bAccumReset – optional flag for forcing a reset of the internal fluence accumulation data.

Return value: int – 0 or greater = number of data records returned in vectors; otherwise error

int computeFullFluence

(dvector& vdFluenceTimes,
vvdvector& vvvdFluence,

bool bReturnFinal = *false*)

Usage: Processes the current contents of the data buffer. Because this is performing an accumulation over the 'full' time duration, no results are normally returned. After all data has been processed, the single set of time-integrated data results, for the entire time duration, is returned in the subsequent call to this method where the bReturnFinal argument is set to *true*. *Any specified accumulation time interval has no effect on these calculations.*

Parameters:

vdFluenceTimes – returned vector of time, in Modified Julian Date form

vvvdFluence – returned 3-dimensional vector of the calculated fluence values. Due to the nature of this accumulation, this vector will be empty except when bReturnFinal is *true*. [time,energy,direction]

bReturnFinal – optional flag for returning the single set of cumulative fluence values. Set to *true* only after *all* data has been processed.

Return value: int – 0 or 1 = number of data records returned in vectors; otherwise error

int computeBoxcarFluence

(dvector& vdFluenceTimes,
vvdvector& vvvdFluence,
ivector& viIntIndices,
bool bReturnPartial = *false*)

Usage: Processes the current contents of the data buffer, then returns the time-integrated data results from any *completed* boxcar accumulation intervals. The duration of the boxcar windows is previously specified in calls to the *setTimeInterval[Sec]*() methods; the time spacing between the start times of subsequent boxcar windows is specified with the *setTimeIncrement*() method . Linear interpolation of the buffered data values is performed when their associated times do not line up with the start or end times of the boxcar accumulation intervals. The returned times correspond to the end times of these completed boxcar accumulation intervals.

The returned boxcar fluence values may subsequently be used as input to the *computeAverageFlux*() method to calculate the associated boxcar interval average flux. The *applyWorstToDate*() method can be used to further process these results, when the appropriate 'maximum' vector of values is maintained.

Parameters:

vdFluenceTimes – returned vector of times, in Modified Julian Date form

vvvdFluence – returned 3-dimensional vector of the calculated fluence values for zero or more completed boxcar accumulation intervals. [time,energy,direction]

viIntIndices – returned vector of the (nearest) indices to the current data buffer entries at which the completed accumulation intervals end. -1 means at or off end of buffer. This information is useful for the annotation of supplementary information associated with the buffered data, such as a time-varying pitch angles.

bReturnPartial – optional flag for returning the calculated fluence values for an incomplete boxcar accumulation interval, if any. Set to *true* only after *all* data has been processed.

Return value: int – 0 or greater = number of data records returned in vectors; otherwise error

int computeAverageFlux

(const dvector& vdFluenceTimes,
const vvdvector& vvvdFluence,
const double& dIntervalSec,
vvdvector& vvvdFluxAvg)

Usage: Computes the average flux rate over fixed-length intervals, based on the input fluence and interval time.

Parameters:

vdFluenceTimes – vector of times, in Modified Julian Date form

vvvdFluence – 3-dimensional vector of the previously-calculated fluence values, associated with the interval end times specified in vdFluenceTimes. [time,energy,direction]

dIntervalSec – duration of time interval, in seconds, used in the calculation of the fluence values. Specify the value used in the *setTimeInterval[Sec]*() method for 'interval'- and 'boxcar'-type accumulations; specify '0' for use with cumulative fluences (no accumulation); specify '-1' for 'full' accumulation fluence.

vvvdFluxAvg – returned 3-dimensional vector of the calculated flux average values [time,energy,direction]

Return value: int – 0 or greater: number of sets of flux averages returned; otherwise error

int computeExponentialFlux

(dvector& vdExpFluxTimes,
vvdvector& vvvdExpFlux,
ivector& viIntIndices,
bool bReturnFinal = *false*)

Usage: Processes the current contents of the data buffer, then returns the exponential average flux data results at the specified interval. The interval is previously specified in calls to the *setTimeInterval[Sec]*() methods. Linear interpolation of the buffered data values is performed when their associated times do not line up with the interval times. The returned times correspond to the end times of these completed processing intervals.

The *applyWorstToDate*() method can be used to further process these results, when the appropriate 'maximum' vector of values is maintained.

Parameters:

vdExpFluxTimes – returned vector of times, in Modified Julian Date form

vvvdExpFlux – returned 3-dimensional vector of the calculated exponential average flux values for zero or more completed processing intervals. [time,energy,direction]

viIntIndices – returned vector of the (nearest) indices to the current data buffer entries at which the completed accumulation intervals end. -1 means at or off end of buffer. This information is useful for the annotation of supplementary information associated with the buffered data, such as a time-varying pitch angles.

bReturnFinal – optional flag for returning the exponential average flux values at the most recently processed data time. Set to *true* only after *all* data has been processed.

Return value: int – 0 or greater = number of data records returned in vectors; otherwise error

int applyWorstToDate

(const vvdvector& vvvdData,
vdvector& vvdMaxData,
vvdvector& vvvdDataWorst)

Usage: Scans the input data, determining the maximum values over time in each of the other two dimensions. These maximum values are returned, as well as the 'worst to date' for the input data.

Parameters:

vvvdData – 3-dimensional vector of data, of no specific type. [time,energy|depth,direction]

vvdMaxData – input and returned 2-dimensional vector of the (current and previously-)determined maximum data values. [energy|depth,direction]. A "reset" of these maximum values is implied when an empty vvvdMaxData vector is passed as *input*.

vvvdDataWorst – returned 3-dimensional vector of the corresponding 'worst to date' of the input data values. [time,energy|depth,direction]

Return value: int – 0 = success, otherwise error

void resetFluence

Usage: clears the internally-stored timestep-based cumulative fluence accumulation data. Subsequent calls to c*omputeFluence*() method will start a new set of fluence accumulation.

Parameters: -none-

void resetIntvFluence

Usage: clears the internally-stored interval-based cumulative fluence accumulation data. Subsequent calls to c*omputeIntvFluence*() method will start a new set of fluence accumulation.

Parameters: -none-

void resetFullFluence

Usage: clears the internally-stored full fluence accumulation data. Subsequent calls to c*omputeFullFluence*() method will start a new set of fluence accumulation.

Parameters: -none-

double getFluenceStartTime

Usage: Returns the starting time for the cumulative fluence accumulation data.

Parameters: -none-

Return value: double – fluence accumulation data start time, in MJD form.

double getIntvFluenceStartTime

Usage: Returns the starting time for the current interval of fluence accumulation data.

Parameters: -none-

Return value: double – fluence accumulation data start time, in MJD form.

double getFullFluenceStartTime

Usage: Returns the starting time for the full fluence accumulation data.

Parameters: -none-

Return value: double – fluence accumulation data start time, in MJD form.

void getLastLength

(dvector& vdLastLength)

Usage: Returns a vector of the abbreviated time duration(s) of the most recent *compute*Fluence*|*Flux*() method call in which the 'bReturnPartial' input parameter was set to *true*. The returned vector will usually contain only one value. However, after calling the *computeBoxcarFluence*() method, more than one value may be returned, but occurs only if the remaining data was spanning more than one 'Boxcar' type accumulation window.

Parameters:

vdLastLength – returned vector of time duration(s), in seconds.

DoseModel Class

Header file: CDoseModel.h

This class is the entry point that provides direct programmatic access to the ShieldDose2 model for the calculation of radiation dose rates received by a target material behind or inside aluminum shielding. Input flux values *must* be 1pt Differential, Omni-directional fluxes, otherwise dose results are invalid.

General:

DoseModel

Usage: Default constructor
Parameters: -none-
Return values: -none-

~DoseModel

Usage: Destructor
Parameters: -none-
Return values: -none-

Model Parameter Inputs:

int setModelDBFile

(const std::string& strModelDBFile)
Usage: Specifies the name of the file (including path) for the dose calculation model database. This database name is in the form of '<path>/sd2DB.h5'.
Parameters:
strModelDBFile – model database filename, including path
Return value: int – 0 = success, otherwise error

int setSpecies

(const std::string& strSpecies)
Usage: Specifies the particle species for the ShieldDose2 model calculations.
Parameters:
strSpecies – species identification: 'H+'|'protons' or 'e-'|'electrons'.
Return value: int – 0 = success, otherwise error

int setEnergies

(const dvector& vdEnergies,
std::string &strEnergyUnits = "MeV")
Usage: Specifies the set of energies (and units) of the flux values that are to be input to the *computeFluxDose[Rate]*() methods.
Parameters:
vdEnergies – vector of energy values
strEnergyUnits – energy units: 'eV', 'keV', 'MeV' or 'GeV'; if omitted, this defaults to 'MeV'
Return value: int – 0 = success, otherwise error

int setDepths

(const dvector& vdDepths,
std::string& strDepthUnits = “mm”)

Usage: Specifies the list of aluminum shielding thickness depths, and their associated units. A minimum of three depth values are required for performing the dose calculations. Input depth values must be in increasing order, with no duplicates. Please consult User’s Guide for nominal range of depth values, dependent upon the units selected.

Parameters:

vdDepths – vector of depth values; in ascending order, no duplicates

strDepthUnits – unit specification: ‘millimeters’|’mm’, ‘mils’ or ‘gpercm2’; defaults to ‘mm’ if omitted

Return value: int – 0 = success, otherwise error

int setDetector

(const std::string& strDetector)

Usage: Specifies the dose detector material type, behind (or inside) the aluminum shielding.

Parameters:

strDetector – material name: ‘Aluminum’|’Al’, ‘Graphite’, ‘Silicon’|’Si’, ‘Air’, ‘Bone’, ‘Tissue’, ‘Calcium’|’Ca’, ‘Gallium’|’Ga’, ‘Lithium’|’Li’, ‘Glass’|’SiO2’, ‘Water’|’H2O’

Return value: int – 0 = success, otherwise error

int setGeometry

(const std::string& strGeometry)

Usage: Specifies the geometry of the aluminum shielding in front of (or around) the detector target.

Parameters:

strGeometry – configuration name: ‘Spherical’, ‘FiniteSlab’ or ‘SemiInfiniteSlab’

Return value: int – 0 = success, otherwise error

int setNuclearAttenMode

(const std::string& strNucAttenMode)

Usage: Specifies the ‘Nuclear Attenuation’ mode used during the ShieldDose2 model calculations.

Parameters:

strNucAttenMode – attenuation mode: ‘None’, ‘NuclearInteractions’ or ‘NuclearAndNeutrons’

Return value: int – 0 = success, otherwise error

Model Execution and Results:

int computeFluxDose

(const dvector& vdFluxData,
dvector& vdDoseData)

Usage: Returns the dose results, based on the input particle flux values and the previously-specified particle species, flux energies, shielding and detector parameters.

Parameters:

vdFluxData – vector of omni-directional differential flux values, over the energy levels previously specified via the *setEnergies*() method, for a single time; or may be fluence values.
These input values are expected to be in units of #/cm^2/sec/MeV (flux) or #/cm^2/MeV (fluence)

vdDoseData – returned vector of dose model results, over the shielding depths previously specified via the *setDepths*() method. When a flux value is input, these returned values are a 'dose rate'; an input of fluence values returns the associated accumulated dose value.

int computeFluxDoseRate

(const vdvector& vvdFluxData,
vdvector& vvdDoseRate)

Usage: Returns the modeled dose rate values, based on the input particle flux values and the previously-specified particle species, flux energies, shielding and detector parameters, for one or more times.

Parameters:

vvdFluxData – 2-dimensional vector of omni-directional differential flux values, over the energy levels previously specified via the *setEnergies*() method, for one or more times.[time,energy]
These input flux values are expected to be in units of #/cm^2/sec/MeV

v*vdDoseData* – returned 2-dimensional vector of dose model results, over the shielding depths previously specified via the *setDepths*() method. [time,depths]

Return value: int – 0 = success, otherwise error

int computeFluxDoseRate

(const vvdvector& vvvdFluxData,
vvdvector& vvvdDoseRate)

Usage: Returns the modeled dose rate values, based on the input particle flux values and the previously-specified particle species, flux energies, shielding and detector parameters, for one or more times.

Parameters:

vvvdFluxData – 3-dimensional vector of omni-directional differential flux values, over the energy levels previously specified via the *setEnergies*() method, for a single direction (*omni-directional only*), for one or more times.[time,energy,direction]
These input flux values are expected to be in units of #/cm^2/sec/MeV

vv*vdDoseData* – returned 3-dimensional vector of dose model results, for a single direction, over the shielding depths previously specified via the *setDepths*() method. [time,depths,direction]

Return value: int – 0 = success, otherwise error

int computeFluenceDose

(const vvdvector& vvvdFluenceData,
vvdvector& vvvdDoseVal)

Usage: Returns the modeled accumulated dose values, based on the input particle flux values and the previously-specified particle species, flux energies, shielding and detector parameters, for one or more times.

Parameters:

vvvdFluenceData – 3-dimensional vector of omni-directional differential fluence values, over the energy levels previously specified via the *setEnergies*() method, for a single direction (*omni-directional only*), for one or more times.[time,energy,direction]

These input fluence values are expected to be in units of #/cm^2/MeV

vv*vdDoseVal* – returned 3-dimensional vector of dose model results, for a single direction, over the shielding depths previously specified via the *setDepths*() method. [time,depths,direction]

Return value: int – 0 = success, otherwise error

AggregModel Class

Header file: CAggregModel.h

This class is the entry point that provides direct programmatic access to the aggregation model. This is used for the collection (or 'aggregation') of data values from multiple scenarios of the 'Perturbed Mean' or 'Monte Carlo' calculations. Once all of the scenario data sets have been loaded into the data aggregation, via the *addScenToAgg*() method, the confidence levels may be calculated using the various *compute**() methods. It is recommended that the aggregation contain at least ten sets of scenario data in order to produce statistically meaningful results.
Important Note: the confidence levels of 0 and 100 percent are excluded from the normal percentile calculations, as they are the 'endpoints', and return simply the minimum or maximum scenario value. When the number of scenarios is less than 100, additional neighboring percent values are also excluded. See the 'Accumulation and Aggregation Inputs' section of the User's Guide document for more details.

General:

AggregModel

Usage: Default constructor
Parameters: -none-
Return values: -none-

~AggregModel

Usage: Destructor
Parameters: -none-
Return values: -none-

Model Parameter Inputs:

void resetAgg

Usage: Clears all time-tagged data from the scenario data aggregation.
Parameters: -none-
Return value: -none-

int addScenToAgg

(const dvector& vdScenTimes,
const vvdvector& vvvdScenData)

Usage: Loads the sets of input time-tagged 'Perturbed Mean' or 'Monte Carlo' scenario data into a new or existing aggregation. The input data sizes and dates of subsequent calls must match those of the first call following a call to the *resetAgg*() method.
Parameters:
vdScenTimes – vector of time values, in Modified Julian Date form
vvvdScenData – 3-dimensional vector of scenario data values to be loaded into the aggregation. [time,energy|depth,direction]
Return value: int – 0 = success, otherwise error

Model Execution and Results:

int computeConfLevel

(const int& iPercent,
dvector& vdPercTimes,
vvdvector& vvvdPercData)

Usage: Calculates the specified confidence level values from the current contents of the scenario data aggregation, at each time direction and energy level or shield depth.

Parameters:

iPercent – input confidence level percent value (0-100)

vdPercTimes – returned vector of times, in Modified Julian Date form

vvvdPercData – returned 3-dimensional vector of the calculated confidence level values [time,energy|depth,direction]

Return value: int – 0 = success, otherwise error

int computeMedian

(dvector& vdPercTimes,
vvdvector& vvvdPercData)

Usage: Calculates the median (50% confidence level) data values from the current contents of the scenario data aggregation, at each time direction and energy level or shield depth.

Parameters:

vdPercTimes – returned vector of times, in Modified Julian Date form

vvvdPercData – returned 3-dimensional vector of the calculated aggregation median values [time,energy|depth,direction]

Return value: int – 0 = success, otherwise error

int computeMean

(dvector& vdPercTimes,
vvdvector& vvvdPercData)

Usage: Calculates the average ('mean') data values from the current contents of the scenario data aggregation, at each time direction and energy level or shield depth.

This is NOT a confidence level. The results of this calculation are of indeterminate meaning.
Use of this method is strongly discouraged.

Parameters:

vdPercTimes – returned vector of times, in Modified Julian Date form

vvvdPercData – returned 3-dimensional vector of the calculated aggregation mean values [time,energy|depth,direction]

Return value: int – 0 = success, otherwise error

AdiabatModel Class

Header file: CAdiabatModel.h

This class is the entry point that provides direct programmatic access to the Adiabat model, for the calculation of the adiabatic invariant values associated with an input set of times, spatial coordinates and pitch angles. Please note that all time values, both input and output, are in Modified Julian Date (MJD) form. Conversions to and from MJD times are available from the DateTimeUtil class, described elsewhere in this Model-Level API section. Position coordinates are always used in sets of three values, in the coordinate system and units that are specified. Please consult the User's Guide document, "Supported Coordinate Systems" for more details; in particular, note the order of the coordinate values for non-Cartesian coordinate systems.
More information about the geomagnetic and adiabatic invariant values may be found in Appendix D and E of the User's Guide document.

General:

AdiabatModel

Usage: Default constructor
Parameters: -none-
Return values: -none-

~AdiabatModel

Usage: Destructor
Parameters: -none-
Return values: -none-

Model Parameter Inputs:

int setKPhiNNetDBFile

(const std::string& strKPhiNNetDBFile)
Usage: Specifies the name of the file (including path) for the K/Phi neural network database. This database name is in the form of '<path>/fastPhi_net.mat'.
Parameters:
strKPhiNNetDBFile – neural network database filename, including path
Return value: int – 0 = success, otherwise error

int setKHMinNNetDBFile

(const std::string& strKHMinNNetDBFile)
Usage: Specifies the name of the file (including path) for the K/Hmin neural network database. This database name is in the form of '<path>/fast_hmin_net.mat'.
Parameters:
strKHMinNNetDBFile – neural network database filename, including path
Return value: int – 0 = success, otherwise error

int setMagfieldDBFile

(const std::string& strMagfieldDBFile)
Usage: Specifies the name of the file (including path) for the magnetic field model database.

This database name is in the form of '<path>/igrfDB.h5'.

Parameters:

strMagfieldDBFile – magnetic field model database filename, including path

Return value: int – 0 = success, otherwise error

Adiabatic invariant limit defaults: K = 0.0 – 25.0; Hmin = -500.0 – 50000.0 [km]; Phi = 0.15849 – 2.0
Any K, Hmin or Phi values calculated to be outside of their respective limits are set to -1.0×10^{-31} fill value.

void setK_Min

(const double& dK_Min = 0.0)

Usage: Specifies the minimum 'K' value returned from *computeCoordinateSet*() methods.

Parameters:

dK_Min – 'K' adiabatic value minimum (0.0 if not specified)

Return value: -none-

void setK_Max

(const double& dK_Max = 25.0)

Usage: Specifies the maximum 'K' value returned from *computeCoordinateSet*() methods.

Parameters:

dK_Min – 'K' adiabatic value maximum (25.0 if not specified)

Return value: -none-

void setHminMin

(const double& dHminMin = -500.0)

Usage: Specifies the minimum 'Hmin' value, altitude in km, returned from *computeCoordinateSet*() methods.

Parameters:

dHminMin – 'Hmin' adiabatic value minimum (-500.0 if not specified)

Return value: -none-

void setHminMax

(const double& dHminMax = 50000.0)

Usage: Specifies the maximum 'Hmin' value, altitude in km, returned from *computeCoordinateSet*() methods.

Parameters:

dHminMin – 'Hmin' adiabatic value maximum (50000.0 if not specified)

Return value: -none-

void setPhiMin

(const double& dPhiMin = 0.15849)

Usage: Specifies the minimum 'Phi' value returned from *computeCoordinateSet*() methods.

Parameters:

dPhiMin – 'Phi' adiabatic value minimum (0.15849 if not specified)

Return value: -none-

void setPhiMax

(const double& dPhiMax = 2.0)

Usage: Specifies the maximum 'Phi' value returned from *computeCoordinateSet*() methods.

Parameters:

dPhiMin – 'Phi' adiabatic value maximum (2.0 if not specified)

Return value: -none-

int updateLimits

Usage: Implements any changes to the K, Hmin or Phi limit specifications. Use of this method is needed only if these limits are changed *after* initial call to one of the *computeCoordinateSet*() methods.

Parameters: -none-

Return value: int – 0 = success, otherwise error

Model Execution and Results:

int computeCoordinateSet

(const std::string& strCoordSys,
const std::string& strCoordUnits,
const dvector& vdTimes,
const dvector& vdCoord1,
const dvector& vdCoord2,
const dvector& vdCoord3,
const dvector& vdPitchAngles,
vdvector& vvdAlpha,
vdvector& vvdLm,
vdvector& vvdK,
vdvector& vvdPhi,
vdvector& vvdHmin,
vdvector& vvdLstar,
dvector& vdBmin,
dvector& vdBlocal,
dvector& vdMagLT)

Usage: Calculates the adiabatic invariant and magnetic field values associated with the input times, position and fixed set of pitch angles. The magnetic field values are independent of pitch angle.

Parameters:

strCoordSys – coordinate system identifier: 'GEI','GEO','GDZ','GSM','GSE','SM','MAG','SPH' or 'RLL'; Please consult the User's Guide document, "Supported Coordinate Systems" for more details.

strCoordUnits - 'km' or 'Re'; 'GDZ' is set to always use 'km' for the altitude value.

vdTimes – vector of time values, in Modified Julian Date form

vdCoord1, vdCoord2, vdCoord3 –vectors of position values, in the coordinate system and units specified by the *strCoordSys* and *strCoordUnit* parameter values.
Please consult the User's Guide document, "Supported Coordinate Systems" for more details; in particular, note the expected 'standard' ordering of the input coordinate values for non-Cartesian coordinate systems.

vdPitchAngles – vector of fixed pitch angles, to be used for all time/position coordinates in the adiabatic invariant value calculations. Valid range: 0.0 - 180.0 degrees.

vvdAlpha – returned 2-dimensional vector of equatorial pitch angles (‘alpha’) associated with the pitch angles at the ephemeris locations. [time,direction]

vvdLm – returned 2-dimensional vector of McIllwain L-shell value associated with the pitch angles at the ephemeris locations. [time,direction]

vvdK – returned 2-dimensional vector of adiabatic invariant ‘K’ value associated with the pitch angles at the ephemeris locations. [time,direction]

vvdPhi – returned 2-dimensional vector of adiabatic invariant ‘Phi’ associated with the pitch angles at the ephemeris locations. [time,direction]

vvdHmin – returned 2-dimensional vector of adiabatic invariant ‘Hmin’ associated with the pitch angles at the ephemeris locations. [time,direction]

vvdLstar – returned 2-dimensional vector of adiabatic invariant ‘L*’ associated with the pitch angles at the ephemeris locations. [time,direction]

vdBmin – returned 1-dimensional vector of minimum IGRF model magnetic field strength value (nanoTeslas) along field line containing the ephemeris location. [time].

vdBlocal – returned 1-dimensional vector of IGRF model magnetic field strength value (nanoTeslas) at the ephemeris location. [time]

vdMagLT – returned 1-dimensional vector of magnetic local time (hours) at the Bmin positions. [time]

Return value: int – 0 = success, otherwise error

int computeCoordinateSet

```
( const std::string& strCoordSys,
  const std::string& strCoordUnits,
  const dvector& vdTimes,
  const dvector& vdCoord1,
  const dvector& vdCoord2,
  const dvector& vdCoord3,
  const vdvector& vvdPitchAngles,
  vdvector& vvdAlpha,
  vdvector& vvdLm,
  vdvector& vvdK,
  vdvector& vvdPhi,
  vdvector& vvdHmin,
  vdvector& vvdLstar,
  dvector& vdBmin,
  dvector& vdBlocal,
  dvector& vdMagLT )
```

Usage: Calculates the adiabatic invariant and magnetic field values associated with the input times, position and a *time-varying* set of pitch angles. The magnetic field values are independent of pitch angle.

Parameters:

strCoordSys – coordinate system identifier: 'GEI','GEO','GDZ','GSM','GSE','SM','MAG','SPH' or 'RLL'; Please consult the User's Guide document, "Supported Coordinate Systems" for more details.

strCoordUnits – coordinate units, 'km' or 'Re'; 'GDZ' is set to always use 'km' for the altitude value.

vdTimes – vector of time values, in Modified Julian Date form

vdCoord1, vdCoord2, vdCoord3 – vectors of position values, in the coordinate system and units specified by the *strCoordSys* and *strCoordUnit* parameter values.

Please consult the User's Guide document, "Supported Coordinate Systems" for more details; in particular, note the expected 'standard' ordering of the input coordinate values for non-Cartesian coordinate systems.

v*vdPitchAngles* – 2-dimensional vector of pitch angles, to be used for each time/position coordinates in the adiabatic invariant value calculations. Valid range: 0.0 - 180.0 degrees. [time,direction]

vvdAlpha – returned 2-dimensional vector of equatorial pitch angles ('alpha') associated with the pitch angles at the ephemeris locations. [time,direction]

vvdLm – returned 2-dimensional vector of McIllwain L-shell value associated with the pitch angles at the ephemeris locations. [time,direction]

vvdK – returned 2-dimensional vector of adiabatic invariant 'K' value associated with the pitch angles at the ephemeris locations. [time,direction]

vvdPhi – returned 2-dimensional vector of adiabatic invariant 'Phi' associated with the pitch angles at the ephemeris locations. [time,direction]

vvdHmin – returned 2-dimensional vector of adiabatic invariant 'Hmin' associated with the pitch angles at the ephemeris locations. [time,direction]

vvdLstar – returned 2-dimensional vector of adiabatic invariant 'L*' associated with the pitch angles at the ephemeris locations. [time,direction]

vdBmin – returned 1-dimensional vector of minimum IGRF model magnetic field strength value (nanoTeslas) along field line containing the ephemeris location. [time].

vdBlocal – returned 1-dimensional vector of IGRF model magnetic field strength value (nanoTeslas) at the ephemeris location. [time]

vdMagLT – returned 1-dimensional vector of magnetic local time (hours) at the Bmin positions. [time]

Return value: int – 0 = success, otherwise error

int calcDirPitchAngles

```
( const std::string& strCoordSys,
  const std::string& strCoordUnits,
  const dvector& vdTimes,
  const dvector& vdCoordX,
  const dvector& vdCoordY,
  const dvector& vdCoordZ,
  const vdvector& vvdDirX,
  const vdvector& vvdDirY,
  const vdvector& vvdDirZ,
  vdvector& vvdPitchAngles )
```

Usage: Calculates the pitch angles corresponding to the input times, position and direction vectors.

Parameters:

strCoordSys – *Cartesian* coordinate system identifier: 'GEI','GEO','GSM','GSE','SM' or 'MAG';
Please consult the User's Guide document, "Supported Coordinate Systems" for more details.

strCoordUnits – coordinate units, 'km' or 'Re'

vdTimes – vector of time values, in Modified Julian Date form

vdCoordX, vdCoordY, vdCoordZ – vectors of position values, in the Cartesian coordinate system and units specified by the *strCoordSys* and *strCoordUnit* parameter values.

vvdDirX, vvdDirY, vvdDirZ – 2-dimensional vectors of direction values, in the Cartesian coordinate system specified by the *strCoordSys* parameter value. These direction values may be full magnitude or unit vectors. [time,direction]

v*vdPitchAngles* – returned 2-dimensional vector of pitch angles corresponding to the input time, position and direction information. [time,direction]

Return value: int – 0 = success, otherwise error

int convertCoordinates

(const std::string& strCoordSys,
const std::string& strCoordUnits,
const dvector& vdTimes,
const dvector& vdCoord1,
const dvector& vdCoord2,
const dvector& vdCoord3,
const std::string& strNewCoordSys,
const std::string& strNewCoordUnits,
dvector& vdNewCoord1,
dvector& vdNewCoord2,
dvector& vdNewCoord3)

Usage: Converts the set of input times, position coordinates from one coordinate system to another.

Parameters:

strCoordSys – coordinate system identifier: 'GEI','GEO','GDZ','GSM','GSE','SM','MAG','SPH' or 'RLL'; Please consult the User's Guide document, "Supported Coordinate Systems" for more details.

strCoordUnits - 'km' or 'Re'; 'GDZ' is set to always use 'km' for the altitude value.

vdTimes – vector of time values, in Modified Julian Date form

vdCoord1, vdCoord2, vdCoord3 –vectors of position values, in the coordinate system and units specified by the *strCoordSys* and *strCoordUnit* parameter values.
Consult the User's Guide document, "Supported Coordinate Systems" for more details; in particular, note the expected 'standard' ordering of the input and output coordinate values for non-Cartesian coordinate systems.

strNewCoordSys – 'new' coordinate system identifier: 'GEI','GEO','GDZ','GSM','GSE','SM','MAG','SPH' or 'RLL'; Please consult the User's Guide document, "Supported Coordinate Systems" for more details.

strNewCoordUnits – 'new' units: 'km' or 'Re'; 'GDZ' is set to always use 'km' for the altitude value.

vdNewCoord1, vdNewCoord2, vdNewCoord3 –vectors of position values, in the 'new' coordinate system and units specified by the *strNewCoordSys* and *strNewCoordUnit* parameter values.

Return value: int – 0 = success, otherwise error

int convertCoordinates

(const std::string& strCoordSys,
const std::string& strCoordUnits,
const double& dTime,
const double& dCoord1,
const double& dCoord2,
const double& dCoord3,
const std::string& strNewCoordSys,
const std::string& strNewCoordUnits,
double& dNewCoord1,
double& dNewCoord2,
double& dNewCoord3)

Usage: Converts a single input time, position coordinates from one coordinate system to another.

Parameters:

strCoordSys – coordinate system identifier: 'GEI','GEO','GDZ','GSM','GSE','SM','MAG','SPH' or 'RLL'; Please consult the User's Guide document, "Supported Coordinate Systems" for more details.

strCoordUnits - 'km' or 'Re'; 'GDZ' is set to always use 'km' for the altitude value.

dTimes – time value, in Modified Julian Date form

dCoord1, dCoord2, dCoord3 –position values, in the coordinate system and units specified by the *strCoordSys* and *strCoordUnit* parameter values.

Consult the User's Guide document, "Supported Coordinate Systems" for more details; in particular, note the expected 'standard' ordering of the input and output coordinate values for non-Cartesian coordinate systems.

strNewCoordSys – 'new' coordinate system identifier: 'GEI','GEO','GDZ','GSM','GSE','SM','MAG','SPH' or 'RLL'; Please consult the User's Guide document, "Supported Coordinate Systems" for more details.

strNewCoordUnits – 'new' units: 'km' or 'Re'; 'GDZ' is set to always use 'km' for the altitude value.

dNewCoord1, dNewCoord2, dNewCoord3 – position values, in the 'new' coordinate system and units specified by the *strNewCoordSys* and *strNewCoordUnit* parameter values.

Return value: int – 0 = success, otherwise error

RadEnvModel Class

Header file: CRadEnvModel.h

This class is the entry point that provides direct programmatic access to the AE8, AP8, CRRESELE and CRRESPRO 'legacy' radiation belt models, providing 'mean' omni-directional particle flux values for the given time and position, with the specified model options and/or conditions.

General:

RadEnvModel

Usage: Default constructor
Parameters: -none-
Return values: -none-

~RadEnvModel

Usage: Destructor
Parameters: -none-
Return values: -none-

Model Parameter Inputs:

int setModel

(const std::string& strModel)
Usage: Specifies the name of the flux model to be used in the calculations.
Parameters:
strModel – model name: 'AE8', 'AP8', 'CRRESELE' or 'CRRESPRO'
Return value: int – 0 = success, otherwise error

int setModelDBFile

(const std::string& strModelDBFile)
Usage: Specifies the name of the database file (including path) for legacy flux model calculations.
This database name is in the form of '<path>/radiationBeltDB.h5'.
Parameters:
strModelDBFile – model database filename, including path
Return value: int – 0 = success, otherwise error

int setMagfieldDBFile

(const std::string& strMagfieldDBFile)
Usage: Specifies the name of the file (including path) for the magnetic field model database.
This database name is in the form of '<path>/igrfDB.h5'.
Parameters:
strMagfieldDBFile – magnetic field model database filename, including path
Return value: int – 0 = success, otherwise error

int setShieldDose2DBFile

(const std::string& strShieldDose2DBFile)

Usage: Specifies the name of the file (including path) for the dose calculation model database. This database name is in the form of '<path>/sd2DB.h5'. *Use of this method is optional*. When not specified, this 'sd2DB.h5' file is assumed to be in the same directory as the 'igrfDB.h5' file, specified using the *setMagfieldDBfile*() method.

Parameters:

strShieldDose2DBFile – dose calculation model database filename, including path

Return value: int – 0 = success, otherwise error

int setFluxType

(const std::string& strFluxType)

Usage: Specifies the type of flux values to be calculated by the model.

Parameters:

strFluxType – flux type identifier: '1PtDiff'|'Differential'|'Diff' or 'Integral'

Return value: int – 0 = success, otherwise error

int setEnergies

(const dvector& vdEnergies)

Usage: Specifies the set energies at which the flux values are calculated by the selected model. Please consult User's Guide, Appendix A for valid ranges/values, which depend on the model selected.

Parameters:

vdEnergies – vector of energy values

Return value: int – 0 = success, otherwise error

int setActivityLevel

(const std::string& strActivityLevel)

Usage: Specifies the geomagnetic activity level parameter for the CRRESPRO, AE8 or AP8 models.

Parameters:

strActivityLevel – geomagnetic activity level specification:
for CRRESPRO model, 'active' or 'quiet';
for AE8 or AP8 model, 'min' or 'max'.

Return value: int – 0 = success, otherwise error

int setActivityRange

(const std::string& strActivityRange)

Usage: Specifies the geomagnetic activity level parameter for the CRRESELE model. Only one of the *setActivityRange*() or *set15DayAvgAp*() methods may be used, otherwise an error is flagged.

Parameters:

strActivityRange – geomagnetic activity level specification, in terms of Ap values:
'5-7.5', '7.5-10', '10-15', '15-20', '20-25', '>25', 'avg', 'max', or 'all'.

Return value: int – 0 = success, otherwise error

int set15DayAvgAp

(const double& d15DayAvgAp)

Usage: Specifies the 15-day average Ap value for the CRRESELE model. Only one of the *setActivityRange*() or *set15DayAvgAp*() methods may be used, otherwise an error is flagged.

Parameters:

d15DayAvgAp – 15-day average Ap value.
Return value: int – 0 = success, otherwise error

void setFixedEpoch

(bool bFixedEpoch)

Usage: Specifies the use of the model-specific fixed epoch (year) for the magnetic field model in the flux calculations. It is *highly recommended* to set this to 'true'. Unphysical results may be produced (especially at low altitudes) if set to 'false'.

Parameters:

bFixedEpoch – *true* or *false*; when *false*, the ephemeris year is used for the magnetic field model.

Return value: -none-

void setShiftSAA

(bool bShiftSAA)

Usage: Shifts the SAA from its fixed-epoch location to the location for the current year of the ephemeris. This setting is ignored if the ***setFixedEpoch*** method is set to 'false'.

Parameters:

bShiftSAA – *true* or *false*.

Return value: -none-

int setCoordSys

(const std::string& strCoordSys,
const std::string& strCoordUnits)

Usage: Specifies the coordinate system and units for the position values that are specified by the *setEphemeris*() method. When not specified, these settings default to 'GEI' and 'Re'. "Re" = radius of the Earth, defined as 6371.2 km.

Parameters:

strCoordSys – coordinate system identifier: 'GEI','GEO','GDZ','GSM','GSE','SM','MAG','SPH' or 'RLL';
Please consult the User's Guide document, "Supported Coordinate Systems" for more details.

strCoordUnits – 'km' or 'Re'; 'GDZ' is set to always use 'km' for the altitude value

Return value: int – 0 = success, otherwise error

int setEphemeris

(const dvector& vdTimes,
const dvector& vdCoords1,
const dvector& vdCoords2,
const dvector& vdCoords3)

Usage: Specifies the ephemeris time and positions to be used for the model calculations.

Parameters:

vdTimes - vector of time values, in Modified Julian Date form. May be identical times or times in chronological order, associated with position coordinates.

vdCoords1, *vdCoords2*, *vdCoords3* - vectors of position coordinate values associated with times vector. These position values are assumed to be in the coordinate system and units specified by *setCoordSys*().

Please consult the User's Guide document, "Supported Coordinate Systems" for more details; in particular, note the expected 'standard' order of the coordinate values for non-Cartesian coordinate systems.

Return value: int – 0 = success, otherwise error

Model Execution and Results:

int computeFlux

(vdvector& vvdFluxData)

Usage: Returns the mean model flux from the specified model, based on the various model parameter inputs.

Parameters:

vvdFluxData – returned 2-dimensional vector of the mean flux values. [time, energy]

Return value: int – 0 = success, otherwise error

CammiceModel Class

Header file: CCammiceModel.h

This class is the entry point that provides direct programmatic access to CAMMICE/MICS 'legacy' plasma particle model. This model is set to produce flux values for twelve pre-defined energy bins (see Appendix B of the User's Guide). *These results cannot be used as input to the DoseModel.*

General:

CammiceModel

Usage: Default constructor
Parameters: -none-
Return values: -none-

~CammiceModel

Usage: Destructor
Parameters: -none-
Return values: -none-

Model Parameter Inputs:

int setModelDBFile

(const std::string& strModelDBFile)
Usage: Specifies the name of the database file (including path) for legacy flux model calculations. This database name is in the form of '<path>/cammiceDB.h5'.
Parameters:
strModelDBFile – model database filename, including path
Return value: int – 0 = success, otherwise error

int setMagfieldDBFile

(const std::string& strMagfieldDBFile)
Usage: Specifies the name of the file (including path) for the magnetic field model database. This database name is in the form of '<path>/igrfDB.h5'.
Parameters:
strMagfieldDBFile – magnetic field model database filename, including path
Return value: int – 0 = success, otherwise error

int setMagfieldModel

(const std::string& strMFModel)
Usage: Specifies the magnetic field option for the CAMMICE model run. 'igrf' uses the IGRF model without an external field model. 'igrfop' adds Olson-Pfitzer/Quiet as the external field model.
Parameters:
strMFModel – magnetic field model specification: 'igrf' or 'igrfop'.
Return value: int – 0 = success, otherwise error

int setDataFilter

(const std::string& strDataFilter)

Usage: Specifies the data filter option for the CAMMICE model run. 'Filtered' excludes data collected during periods when the DST index was below -100.

Parameters:

strDataFilter – data filter specification: 'all' or 'filtered' .

Return value: int – 0 = success, otherwise error

int setPitchAngleBin

(const std::string& strPitchAngleBin)

Usage: Specifies the pitch angle bin for the CAMMICE model run.

Parameters:

strPitchAngleBin – pitch angle bin identification: '0-10','10-20','20-30','30-40','40-50','50-60','60-70','70-80','80-90', '100-110','110-120','120-130','130-140','140-150','150-160','160-170','170-180' or 'omni'.

Return value: int – 0 = success, otherwise error

int setSpecies

(const std::string& strSpecies)

Usage: Specifies the (single) particle species for the CAMMICE model run.

Parameters:

strSpecies – species identification: 'H+', 'He+', 'He+2', 'O+', 'H', 'He', 'O', or 'Ions'.

Return value: int – 0 = success, otherwise error

int setCoordSys

(const std::string& strCoordSys,
const std::string& strCoordUnits)

Usage: Specifies the coordinate system and units for the position values that are specified by the *setEphemeris*() method. When not specified, these settings default to 'GEI' and 'Re'. "Re" = radius of the Earth, defined as 6371.2 km.

Parameters:

strCoordSys – coordinate system identifier: 'GEI','GEO','GDZ','GSM','GSE','SM','MAG','SPH' or 'RLL'; Please consult the User's Guide document, "Supported Coordinate Systems" for more details.

strCoordUnits – 'km' or 'Re'; 'GDZ' is set to always use 'km' for the altitude value

Return value: int – 0 = success, otherwise error

int setEphemeris

(const dvector& vdTimes,
const dvector& vdCoords1,
const dvector& vdCoords2,
const dvector& vdCoords3)

Usage: Specifies the ephemeris time and positions to be used for the model calculations.

Parameters:

vdTimes - vector of time values, in Modified Julian Date form. May be identical times or times in chronological order, associated with position coordinates.

vdCoords1, *vdCoords2*, *vdCoords3* - vectors of position coordinate values associated with times vector. These position values are assumed to be in the coordinate system and units specified by *setCoordSys*().
Please consult the User's Guide document, "Supported Coordinate Systems" for more details; in particular, note the expected 'standard' order of the coordinate values for non-Cartesian coordinate systems.
Return value: int – 0 = success, otherwise error

Model Execution and Results:

int computeFlux

(vdvector& vvdFluxData)

Usage: Returns the mean model flux from the CAMMICE model, based on the various model parameter inputs.

Parameters:

vvdFluxData – returned 2-dimensional vector of the mean flux values. [time, energy]

Return value: int – 0 = success, otherwise error

DateTimeUtil Class

Header file: CDateTimeUtil.h

This class is the entry point that provides direct programmatic access to date and time conversion utilities.

General:

DateTimeUtil

Usage: Default constructor
Parameters: -none-
Return values: -none-

~DateTimeUtil

Usage: Destructor
Parameters: -none-
Return values: -none-

Model Execution and Results:

double getGmtSeconds

(const int& iHours,
const int& iMinutes,
const double& dSeconds)

Usage: Determines the GMT seconds of day for the input hours, minutes and seconds.
Parameters:
iHours – hours of day (0-23)
iMinutes – minutes of hour (0-59)
dSeconds – seconds of minute (0-59.999)
Return value: double – GMT seconds of day

int getDayOfYear

(const int& iYear,
const int& iMonth,
const int& iDay)

Usage: Determines the day number of year for the input year, month and day number.
Parameters:
iYear – year (1950-2049)
iMonth – month (1-12)
iDay – day of month (1-28|29|30|31)
Return value: int – day number of year

double getModifiedJulianDate

(const int& iYear,
const int& iDdd,
const double& dGmtsec)

Usage: Determines the Modified Julian Date for the input year, day of year and GMT seconds.

Parameters:
iYear – year (1950-2049)
iDdd – day of year (1-365|366)
dGmtsec – GMT seconds of day (0-86399.999)
Return value: double – Modified Julian Date (33282.0 - 69806.999)

double getModifiedJulianDate

(const int& iUnixTime)
Usage: Determines the Modified Julian Date for the input UNIX time value.
(due to limitations of Unix time, this will be valid only between 01 Jan 1970 – 19 Jan 2038).
Parameters:
iUnixTime – Unix Time, in seconds from 01 Jan 1970, 0000 GMT; (0 – MaxInt)
Return value: double – Modified Julian Date (40587.0 - 65442.134)

int getDateTime

(const double& dModJulDate,
int& iYear,
int& iDdd,
double& dGmtsec)
Usage: Determines the year, day of year and GMT seconds for the input Modified Julian Date.
Parameters:
dModJulDate – Modified Julian Date (33282.0 - 69806.999)
iYear – returned year (1950-2049)
iDdd – returned day of year (1-365|366)
dGmtsec – returned GMT seconds of day (0-86399.999)
Return value: int – 0 = success, otherwise error

int getDateTime

(const double& dModJulDate,
int* piYear,
int* piDdd,
double* pdGmtsec)
Usage: Determines the year, day of year and GMT seconds for the input Modified Julian Date.
Parameters:
dModJulDate – Modified Julian Date value (33282.0 - 69806.999)
piYear – *pointer* to returned year (1950-2049)
piDdd – *pointer* to returned day of year (1-365|366)
pdGmtsec – *pointer* to returned GMT seconds of day (0-86399.999)
Return value: int – 0 = success, otherwise error

int getHoursMinSec

(const double& dGmtsec,
int& iHours,
int& iMinutes,
double& dSeconds)
Usage: Determines the hours, minutes and seconds for the input GMT seconds.
Parameters:

dGmtsec – GMT seconds of day (0-86399.999)
iHours – returned hours of day (0-23)
iMinutes – returned minutes of hour (0-59)
dSeconds – returned seconds of minute (0-59.999)
Return value: int – 0 = success, otherwise error

int getHoursMinSec

(const double& dGmtsec,
int* piHours,
int* piMinutes,
double* pdSeconds)
Usage: Determines the hours, minutes and seconds for the input GMT seconds.
Parameters:
dGmtsec – GMT seconds of day (0-86399.999)
piHours – *pointer* to returned hours of day (0-23)
piMinutes – *pointer* to returned minutes of hour (0-59)
pdSeconds – *pointer* to returned seconds of minute (0-59.999)
Return value: int – 0 = success, otherwise error

int getMonthDay

(const int& iYear,
const int& iDdd,
int& iMonth,
int& iDay)
Usage: Determines the month and day number for the input year and day of year.
Parameters:
iYear – year (1950-2049)
iDdd – day of year (1-365|366)
iMonth – returned month (1-12)
iDay – returned day of month (1-28|29|30|31)
Return value: int – 0 = success, otherwise error

int getMonthDay

(const int& iYear,
const int& iDdd,
int* piMonth,
int* piDay)
Usage: Determines the month and day number for the input year and day of year.
Parameters:
iYear – year (1950-2049)
iDdd – day of year (1-365|366)
piMonth – *pointer* to returned month (1-12)
piDay – *pointer* to returned day of month (1-28|29|30|31)
Return value: int – 0 = success, otherwise error

To contact the IRENE (AE9/AP9/SPM) model development team, email ae9ap9@vdl.afrl.af.mil .

The IRENE model package and related information can be obtained from AFRL's Virtual Distributed Laboratory (VDL) website: https://www.vdl.afrl.af.mil/programs/ae9ap9