IRENE: AE9/AP9/SPM Radiation Environment Model

C++ Application Programming Interface

Version 1.58.001

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], Yi-Jiun Su[1], Christopher Roth[5], Rick Quinn[5], Michael Starks[1], Paul Whelan[5], Reiner Friedel[6], Chad Lindstrom[1], Steve Morley[6] and Dan Madden[7].

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

V1.57.004 release: 21 July 2022

V1.58.001 release: 04 March 2024

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

IRENE: AE9/AP9/SPM Model Application Programming Interface
Version 1.58.001

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 calls to the environment models and/or data usage directly into a new or existing application, an Application Programming Interface (API) is available.

The IRENE model package supports programmatic access through a suite of APIs accessible from the C++, C and Python programming language. The main software is written in C++; the C and Python interfaces use "wrappers" for accessing the underlying C++ libraries. The base distribution provides all C and C++ header files, pre-compiled 64-bit Windows executables and library files, and Python modules. The source distribution also contains the complete set of source code files for building the executables and libraries on a Linux system; refer to the 'Build Instructions' document for more details. Either distribution can be used for the development of user applications employing the API libraries.

The IRENE API may be accessed at two different levels: the 'Application-Level' API provides higher-level programmatic access to most of the processing features and options available from the 'CmdLineIrene' application (or its GUI), including parallelization; the 'Model-Level' API provides lower-level programmatic access to each of the underlying models and components that comprise the IRENE 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.

DemoApp and DemoModel Programs

Included in the distribution of IRENE are two demonstration programs for showing how the IRENE API may be used in user applications. They are located in the 'Irene/api_demos' folder of the distribution, written in the three supported languages: C++, C, and Python. The *DemoApp* program demonstrates the use of the Application-level API, specifying an orbit ephemeris and performing several flux and fluence calculations, and then accessing the various forms of the results. The *DemoModel* program demonstrates the use of the IRENE API for several variations of calls to the Ephemeris, AE9, AP9, Adiabatic and Dose individual model components.

User applications in C++ that utilize the IRENE package API may be built using CMake, as was the various executable and library files of the IRENE package. 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), HDF5® (for internal databases), and Xerces-C++ (XML parser). Windows builds will require the use of Visual Studios 2017. The Intel MPI Library development product will be required for building of multi-threaded 64-bit Windows applications.

More information about the CMake build system can be found at https://cmake.org/documentation/ .

The C++ version of *DemoApp* application can be built using CMake using the provided configuration files. Make a build directory (such as '~/Irene/<*platform*>/demoApp') [here '~' refers to the path to the 'Irene' installation location, and <*platform*> may be 'linux' or 'win64']. Navigate to that directory, then enter the command:

```
cmake ~/Irene/api_demos/demoApp/CXX -DIRENE_ROOT=~/Irene/<platform>
```

The optional '`-DCMAKE_INSTALL_PREFIX=<path>`' can be used to specify a location for the 'install' step, if desired. The *DemoApp* executable will be copied to the "<*path*>/bin" directory. The multi-threaded version (*DemoAppMpi*) can be also be built by adding '`-DUSE_MPI=YES`' to the *cmake* command.
Then use the commands '`make`' and '`make install`' on Linux. On Windows, use the Visual Studios 'solution' file for the 'build' and 'install' steps, as described in the "Build Instructions" document.

This application performs a variety of model calculations and associated operations. Enter:

```
DemoApp -x ~/Irene/<platform>/bin -d ~/Irene/modelData
```

Other application argument options are available; enter '`DemoApp -h`' for the full list.
A 'launcher' may be required for executing the multi-threaded *DemoAppMpi* application:
Linux:

```
mpirun -np 1 DemoAppMpi -x ~/Irene/<platform>/bin -d ~/Irene/modelData -n 5
```

Windows:

```
mpiexec.exe -np 1 DemoAppMpi.exe -x ~/Irene/<platform>/bin -d ~/Irene/modelData -n 5
```

The C++ version of the *DemoModel* application build process is similar. Make a build directory (such as '~/Irene/<*platform*>/demoModel'). Navigate to that directory, then enter the command:

```
cmake ~/Irene/api_demos/demoModel/CXX -DIRENE_ROOT=~/Irene/<platform>
```

Refer to the "Build Instructions" document for more details.
The optional '`-DCMAKE_INSTALL_PREFIX=<path>`' can be used to specify a location for the 'install' step, if desired. The executable will be copied to the "<*path*>/bin" directory. (Multi-threaded operation is not available in the Model-Level API).
Then use the commands '`make`' and '`make install`' on Linux. On Windows, use the Visual Studios 'solution' file for the 'build' and 'install' steps, as described in the "Build Instructions" document.

This application also performs a variety of model calculations and associated operations. Enter:

```
DemoModel ~/Irene/modelData
```

It is hoped that these two demonstration applications, and their build scripts, will be helpful in building your own custom applications. Study of the various API routines being called in these codes against the API reference documents may be helpful in better understanding their effective usage.

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

Application-Level C++ API Reference

Application Class

Header file: CApplication.h

This class is the main entry point that provides methods to programmatically configure, execute, and then access the model results available from the CmdLineIrene application. This API does not directly access the models, but instead relies on the execution of the same set of supporting programs used by the CmdLineIrene application, and then queries the results from the set of datafiles that are generated by the model calculation calls. 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
Return values: -none-

~Application

Usage: Destructor
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 string& strExecDir)
Usage: (***Required***) Specifies the directory path to the executable programs that are needed for performing the prescribed model calculations.
Parameters:
strExecDir - path to installation executables, such as *CmdLineIrene(.exe)*
Return value: int – 0 = success, otherwise error

int setWorkDir

(const 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 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 another call is made to the *runModel*() method. When not specified, the default setting is *true*. Does not apply to calls to the *resetModelRun*() method.
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 'controller' 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 <u>cluster</u> system), specify the number as a *negative* value to bypass the query. When not specified, model calculations will be performed using a single processor. *On Windows machines with active VPN connections, multi-threaded model runs will <u>fail</u> or <u>stall</u> unless the environment variable* '`FI_TCP_IFACE`' *is set to* '`lo`'.
†Use of Intel CPU's 'Hyper-threaded' threads as a processor may *degrade* performance (YMMV).
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 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) or "Hydra")

Local:	for use on the local Windows machine with multiple processors
Hydra:	for use on a Windows cluster, relies on external 'hydra_service' for MPI communication. The 'hydra_service' utility program is included in the Irene/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: 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 960 by default. Use of this default value is highly recommended; for systems with limited available memory resources, a value of 120 is suggested to improve performance. However, regardless of ample memory resources, specifying values larger than 2400 will likely *degrade* processing performance.
This specification also governs the size of data 'chunks' that are returned from each call to the various *flyin**() and *get*Data*() 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 setTaskDelay

(const int& iTaskDelay)
Usage: This specifies the number seconds to delay between sets of multi-threaded processing 'tasks'. Values larger than 1 (the default) are only needed when using the maximum number of processors *and* MPI management-related errors occur (ie '*not enough slots*' or '*all nodes are already filled*'). A longer delay may be needed for the MPI "housekeeping" to be completed when executing on a busy system.
Parameters:
iTaskDelay – number of seconds to delay; valid values: 1 - 5.
Return value: int – 0 = success, otherwise error

string getExecDir

Usage: Returns the directory path to the executable programs as specified in the *setExecDir*() method.
Return value: string containing path to installation executables.

string getWorkDir

Usage: Returns the directory path of the temporary directory, as specified in the *setWorkDir*() method.
Return value: string containing path to location for the temporary directory.

string getBinDirName

Usage: Returns the name for the temporary directory for the intermediate binary files, as specified in the *setBinDirName*() method.
Return value: string – name for the temporary directory for the intermediate binary files, or 'default'.

int getDelBinDir

(bool bVerdict)
Usage: Returns the disposition of the temporary directory and its intermediate binary files, as specified in the *setDelBinDir*() method.
Return value: int – 0 = false, 1= true.

int getNumProc

Usage: Returns the number of processors to use, as specified in the *setNumProc*() method.
Return value: int – number of processors

int getNumFileIo

Usage: Returns the number of threads to use for the file I/O steps, as specified in the *setNumFileIo*() method.
Return value: int – number of processors.

string getWindowsMpiMode

Usage: Returns the Windows MPI mode, as specified in the *setWindowsMpiMode*() method.
Return value: string – Windows MPI mode.

int getChunkSize

Usage: Returns the current value of the 'chunk' size, as specified in the *setChunkSize*() method.
Return value: int – number of entries in processing chunk.

int getTaskDelay

Usage: Returns the current value of the 'task delay', as specified in the *setTaskDelay*() method.
Return value: int – number of seconds to delay between multi-threaded processing 'tasks'.

External Ephemeris Specification:

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

int setInCoordSys

(const string& strCoordSys,
  const string& strCoordUnits)

Usage: Specifies the coordinate system and units for the position values that are specified by the *setInEphemeris*() 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 setInEphemeris

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

Usage: Specifies the input 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 clearInEphemeris

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

Return value: -none-

string getInCoordSys

Usage: Returns the coordinate system, specified in the *setInCoordSys*() method for the set of position values as specified in the *setInEphemeris*() method.

Return value: string – coordinate system

string getInCoordSysUnits

Usage: Returns the coordinate system units, specified in the *setInCoordSys*() method for the set of position values as specified in the *setInEphemeris*() method.
Return value: string – coordinate system units ('Re' or 'km')

int getNumInEphemeris

Usage: Returns the number of currently defined ephemeris time and position entries.
Return value: int – number of ephemeris entries, specified in the *setInEphemeris*() method..

void getInEphemeris

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

Usage: Returns the input ephemeris time and positions, as specified in the *setInEphemeris*() method.
Returned parameters:
vdTimes – vector of time values, in Modified Julian Date form.
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 returned from the *getInCoordSys*() and *getInCoordSysUnits*() methods.
Returned 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 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 string& strMode,
  const 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-

---

string getPropagator

Usage: Returns the orbit propagator identifier, as specified in the *setPropagator*() method.
Return value: string – orbit propagator name.

string getSGP4Mode

Usage: Returns the SGP4 propagator mode setting, as specified in the *setSGP4Param*() method.
Return value: string – SGP4 mode setting.

string setSGP4Datum

Usage: Returns the SGP4 propagator WGS setting, as specified in the *setSGP4Param*() method.
Return value: string – SGP4 WGS setting.

bool getKeplerUseJ2

Usage: Returns the Kepler propagator 'J2' perturbation setting, as specified in the *setKeplerUseJ2*() method.
Return value: bool – True or False.

---

int setTimes

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

void getTimes

(double& dStartTime,
double& dEndTime,
double& dTimeStepSec)
Usage: Returns the start and stop times, and time step, for the ephemeris generation.
Returned parameters:
dStartTime, *dEndTime* - start and stop time values, in Modified Julian Date form
dTimeStepSec - time step size, in seconds

Return value: -none-

int setVarTimes

(const double& dStartTime,
const double& dEndTime,
const double& dTimeMinStepSec,
const double& dTimeMaxStepSec = 3600.0,
const double& dTimeRoundSec = 5.0)

Usage: Specifies the start and stop times (*inclusive*), and variable time step limits, of the ephemeris information to be generated by the orbit propagator from the defined orbital element values or TLE file. Variable time steps are calculated based on the orbital radial values, and are useful for the more elliptical orbits (ie eccentricity>0.25). See the User's Guide document "Orbit Ephemeris File Description" section for more information.

Parameters:

dStartTime, *dEndTime* - start and stop time values, in Modified Julian Date form
dTimeMinStepSec – lower limit of variable time steps, in seconds; must be ≥ 10 seconds
dTimeMaxStepSec – upper limit of variable time steps, in seconds; must be > min, ≤ 3600 seconds
dTimeRoundSec – rounding of variable time steps, in whole seconds; use 0 for no rounding; < min

Return value: int – 0 = success, otherwise error

void getVarTimes

(double& dStartTime,
double& dEndTime,
double& dTimeMinStepSec,
double& dTimeMaxStepSec,
double& dTimeRoundSec)

Usage: Returns the start and stop times, and variable time step limits, for the ephemeris generation.

Returned parameters:

dStartTime, *dEndTime* - start and stop time values, in Modified Julian Date form
dTimeMinStepSec – lower limit of variable time steps, in seconds
dTimeMaxStepSec – upper limit of variable time steps, in seconds
dTimeRoundSec – rounding of variable time steps, in seconds

Return value: -none-

int setTimesList

(const dvector& vdTimes)

Usage: Specifies the vector of times, in Modified Julian Date form, of the ephemeris information to be generated by the orbit propagator from the defined 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

void getNumTimesList

Usage: Returns the number of time entries defined for the ephemeris generation, from the specifications in the *setTimesList*() method.

Return value: int – number of ephemeris times defined

void getTimesList

(dvector& vdTimes)

Usage: Returns the vector of time values, in Modified Julian Date form, for the ephemeris generation, from the specifications in the *setTimesList*() method.

Returned parameters:

*vdTime*s – vector of time values, in Modified Julian Date form

Return value: -none-

void clearTimesList

Usage: Clears the time entries defined for the ephemeris generation, from the specifications in the *setTimesList*() methods.

Return value: -none-

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 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: Clears the specification of the TLE file.

Return value: -none-

string getTLEFile

Usage: Returns the name of the specified TLE file, as specified in the *setTLEFile*() method.

Return value: string – path and filename of the TLE file.

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 (>0, but <~20Re)
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 (>0, but <~20Re)
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 (1-75), 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 East longitude of satellite in a geosynchronous orbit.

Parameters:

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

Return value: int – 0 = success, otherwise error

int setStateVectors

(const dvector& vdPos,
const dvector& vdVel)

Usage: Specifies the satellite's position and velocity in the GEI coordinate system at the element's 'epoch' time. Alternatively, the *setPositionGEI*() and *setVelocityGEI*() method could be used instead.

Parameters:

v*dPos* – vector containing the GEI coordinate system satellite position values (X,Y,Z), in km

vdVel – vector containing the GEI coordinate system satellite velocity values (X,Y,Z), in km/sec

Return value: int – 0 = success, otherwise error

int setPositionGEI

(const double& dPosX,
const double& dPosY,
const double& dPosZ)

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:

dPosX, dPosY, dPosZ – GEI coordinate system satellite position values, in km (>1Re, but <~75Re)

Return value: int – 0 = success, otherwise error

int setVelocityGEI

(const double& dVelX,

const double& dVelY,
const double& dVelZ)

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:

dVelX, dVelY, dVelZ – GEI coordinate system satellite velocity values, in km/sec

Return value: int – 0 = success, otherwise error

int setCoordSys

(const string& strCoordSys,
const 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

---

double getElementTime

Usage: Returns the 'epoch' time associated with the set of orbital element values, as specified in the *setElementTime*() method, in Modified Julian Date form.

Return value: double – element 'epoch' time.

double getInclination

Usage: Returns the orbital element 'Inclination' value, in degrees, as specified in the *setInclination*() method.

Return value: double – orbit inclination angle, in degrees.

double getRightAscension

Usage: Returns the orbital element 'Right Ascension of the Ascending Node' value, as specified in the setRightAscension() method.

Return value: double – orbit ascending node position, in degrees.

double getEccentricity

Usage: Returns the orbital element 'Eccentricity' value, as specified in the *setEccentricity*() method.

Return value: double – orbit eccentricity value.

double getArgOfPerigee

Usage: Returns the orbital element 'Argument of Perigee' value, as specified in the *setArgOfPerigee*() method.

Return value: double – orbit perigee position, in degrees.

double getMeanAnomaly

Usage: Returns the orbital element 'Mean Anomaly' value, as specified in the *setMeanAnomaly*() method.
Return value: double – orbit mean anomaly value, in degrees.

double getMeanMotion

Usage: Returns the orbital element 'Mean Motion' value, as specified in the *setMeanMotion*() method.
Return value: double – orbit mean motion value, in revolutions per day.

double getMeanMotion1stDeriv

Usage: Returns the orbital element 'First Time Derivative of the Mean Motion' value, as specified in the *setMeanMotion1stDeriv*() method.
Return value: double – first derivative of mean motion

double getMeanMotion2ndDeriv

Usage: Returns the orbital element 'Second Time Derivative of the Mean Motion' value, as specified in the *setMeanMotion2ndDeriv*() method.
Return value: double – second derivative of mean motion.

double getBStar

Usage: Returns the orbital element 'B*' value, as specified in the *setBStar*() method.
Return value: double – ballistic coefficient value.

double getAltitudeOfApogee

Usage: Returns the orbital element 'Apogee Altitude' value, as specified in the *setAltitudeOfApogee*() method.
Return value: double – orbit apogee altitude, in km.

double getAltitudeOfPerigee

Usage: Returns the orbital element 'Perigee Altitude' value, as specified in the *setAltitudeOfPerigee*() method
Return value: double – orbit perigee altitude, in km.

double getLocalTimeOfApogee

Usage: Returns the local time of the orbit's apogee, as specified in the *setLocalTimeOfApogee*() method.
Return value: double – local time of orbit perigee, in hours + fraction.

double setLocalTimeMaxInclination

Usage: Returns the local time of the orbit's maximum inclination, as specified in the *setLocalTimeMaxInclination*() method.
Return value: double – local time of orbit maximum inclination, in hours + fraction.

double getTimeOfPerigee

Usage: Returns the time of the orbit's perigee, as specified in the *setTimeOfPerigee*() method.
Return value: double – orbit perigee time value, in Modified Julian Date form.

double getSemiMajorAxis

Usage: Returns the orbit's semi-major axis length, as specified in the *setSemiMajorAxis*() method.
Return value: double – orbit semi-major axis length, in units of Re (1 Re = 6371.2 km).

double getGeosynchLon

Usage: Returns the geographic longitude of satellite in a geosynchronous orbit, as specified in the *setGeosynchLon*() method.
Return value: double – orbit geosynchronous (East) longitude, in degrees.

void getStateVectors

```
( dvector& vdPos,
  dvector& vdVel )
```

Usage: Returns the satellite's position and velocity vector values, as specified in the *setStateVectors*() method, or in the *setPositionGEI*() and *setVelocityGEI*() methods.
Returned parameters:
v*dPos* – vector containing the GEI coordinate system satellite position values (X,Y,Z), in km
vdVel – vector containing the GEI coordinate system satellite velocity values (X,Y,Z), in km/sec
Return value: -none-

void getPositionGEI

```
( double& dPosX,
  double& dPosY,
  double& dPosZ )
```

Usage: Returns the satellite's position vector, as specified in either the *setPositionGEI*() or *setStateVectors*() method.
Returned parameters:
dPosX, dPosY, dPosZ – GEI coordinate system satellite position values, in km (>1Re, but <~75Re)
Return value: -none-

void getVelocityGEI

```
( double& dVelX,
  double& dVelY,
  double& dVelZ )
```

Usage: Returns the satellite's velocity vector, as specified in either the *setVelocityGEI*() or *setStateVectors*() method.
Returned parameters:
dVelX, dVelY, dVelZ – GEI coordinate system satellite velocity values, in km/sec
Return value: -none-

string getCoordSys

Usage: Returns the coordinate system name, as specified in the *setCoordSys*() method.
Return value: string – coordinate system name.

string getCoordSysUnits

Usage: Returns the coordinate system units, as specified in the *setCoordSys*() method.
Return value: string – coordinate system units (‘Re’ or ‘km’).

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 string& strModel)

Usage: Specifies the name of the flux model to be used in the calculations. Note the 'Plasma' model names now includes the species type.
See the following 'Legacy Model Parameter Inputs' section for Legacy model-specific options.

Parameters:

strModel – model name: 'AE9', 'AP9', 'PlasmaE', 'PlasmaH', 'PlasmaHe' or 'PlasmaO'
Legacy models: 'AE8', 'AP8', 'CRRESELE', 'CRRESPRO' or 'CAMMICE'

Return value: int – 0 = success, otherwise error

int setModelDBDir

(const string& strDataDir)

Usage: Specifies the directory that contains the collection IRENE model database files. The various database files required are automatically selected according to the model and parameters specified. <u>The use of this method is highly recommended</u>, as it *eliminates* the need for the other methods that specify the individual database files; those are only needed for using alternate or non-standard versions.

Parameters:

strDataDir – directory path for the IRENE database files.

Return value: int – 0 = success, otherwise error

int setModelDBFile

(const string& strDataSource)

Usage: Specifies the name of the database file (including path) for flux model calculations. The use of this method is *not needed* when *setModelDBDir*() is specified, except when an alternate or non-standard database file is to be used (this overrides any automatic file specification).

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 setKPhiDBFile

(const string& strDataSource)

Usage: Specifies the name of the file (including path) for the K/Phi database. The use of this method is *not needed* when *setModelDBDir*() is specified, except when an alternate or non-standard database file is to be used (this overrides any automatic file specification).

This database name is in the form of '<path>/fastPhi_net.mat'.

Parameters:

strDataSource – database filename, including path

Return value: int – 0 = success, otherwise error

int setKHMinDBFile

(const string& strDataSource)

Usage: Specifies the name of the file (including path) for the K/Hmin database. The use of this method is *not needed* when *setModelDBDir*() is specified, except when an alternate or non-standard database file is to be used (this overrides any automatic file specification).

This database name is in the form of '<path>/fast_hmin_net.mat'.

Parameters:

strDataSource – database filename, including path

Return value: int – 0 = success, otherwise error

int setMagfieldDBFile

(const string& strDataSource)

Usage: Specifies the name of the file (including path) for the magnetic field model database. The use of this method is *not needed* when *setModelDBDir*() is specified, except when an alternate or non-standard database file is to be used (this overrides any automatic file specification).

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 string& strDataSource)

Usage: Specifies the name of the file (including path) for the dose calculation model database. The use of this method is *not needed* when *setModelDBDir*() and/or *setMagfieldDBFile*() is specified, except when an alternate or non-standard database file is to be used (this overrides any automatic file specification).

This database name is in the form of '<path>/sd2DB.h5'.

Parameters:

strDataSource – dose calculation model database filename, including path

Return value: int – 0 = success, otherwise error

void setAdiabatic

(bool bVerdict = *true*)

Usage: Specifies if the full adiabatic invariant values are to be calculated and be available during model run. Default state is 'off'. Not applicable for Legacy model runs.

Parameters:

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

Return value: -none-

int setFluxType

(const string& strFluxType)

Usage: Specifies the type of flux values to be calculated by the model.
Note that use of '2PtDiff' type requires that the sets of lower and upper bound flux energies be defined using the *setFluxEnergies*() [or *setFluxEnergy*()] and *setFluxEnergies2*() [or *setFluxEnergy2*()] methods, respectively.

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.
If using the ‘2PtDiff’ flux type, these specify the *lower bounds* of energy bins.
Please consult User’s Guide for valid ranges/values, which depend on the model selected.
Parameters:
vdEnergies – vector of energy values [MeV]
Return value: int – 0 = success, otherwise error

int setFluxEnergy

(const double& dEnergy)
Usage: Specifies a single energy to be added to the list of defined flux energies.
If using the ‘2PtDiff’ flux type, this specifies a *lower bound* of the energy bins.
Please consult User’s Guide for valid ranges/values, which depend on the model selected.
Parameters:
dEnergy – energy value [MeV]
Return value: int – 0 = success, otherwise error

void clearFluxEnergies

Usage: Clears the flux energy list defined by previous calls to the *setFluxEnergy*() or *setFluxEnergies*() methods.
Return value: -none-

int setFluxEnergies2

(const dvector& vdEnergies2)
Usage: This is only needed when using the ‘2PtDiff’ flux type. Specifies the *upper bounds* of the energy bins, corresponding to the energies (as lower bounds) specified using the *setFluxEnergy*() or *setFluxEnergies*() methods.
Please consult User’s Guide for valid ranges/values, which depend on the model selected.
Parameters:
vdEnergies2 – vector of energy values [MeV]
Return value: int – 0 = success, otherwise error

int setFluxEnergy2

(const double& dEnergy)
Usage: This is only needed when using the ‘2PtDiff’ flux type. Specifies a single energy to be added to the list of defined *upper bound* flux energies, corresponding to the energies (as lower bounds) specified using the *setFluxEnergy*() or *setFluxEnergies*() methods.
Please consult User’s Guide for valid ranges/values, which depend on the model selected.
Parameters:
dEnergy – energy value [MeV]
Return value: int – 0 = success, otherwise error

void clearFluxEnergies2

Usage: Clears the *upper bound* flux energy list defined by previous calls to the *setFluxEnergy2*() or *setFluxEnergies2*() methods.
Return value: -none-

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 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 clearPitchAngles

Usage: Clears the pitch angle list defined by calls to *setPitchAngle*() or *setPitchAngles*() methods.
Return value: -none-

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 setFluxPercentiles

(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 clearFluxPercentiles

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

int setFluxPerturbedScenario

(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 setFluxPerturbedScenarios

(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 clearFluxPerturbedScenarios

Usage: Clears the current list of defined perturbed mean scenario numbers.

Return value: -none-

int setFluxMonteCarloScenario

(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 setFluxMonteCarloScenarios

(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 clearFluxMonteCarloScenarios

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

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. By default, perturbation mode is enabled. Disabling these perturbations is generally only useful for validation or where perturbations dwarf physical features of interest.

Parameters:

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

Return value: -none-

void setMonteCarloWorstCase

(bool bVerdict)

Usage: Enables (*true*) or disables (*false*) the tracking of the 'maximum-to-date' Monte Carlo flux average results *for the 'Boxcar' and 'Exponential' accumulation modes only*. By default, this tracking is disabled.

Parameters:

bVerdict – true or *false* for the tracking of the 'maximum-to-date' Monte Carlo results.

Return value: -none-

---

string getModel

Usage: Returns the name of the flux model, as specified in the *setModel*() method.

Return value: string – model name.

string getModelDBDir

Usage: Returns the directory name containing the collection of IRENE model database files that was specified in a previous call to the *setModelDBDir*() method; otherwise, blank.

Return value: string – model database directory.

string getModelDBFile

Usage: Returns the name of the database file for flux model calculations. This will be available immediately, when specified using the *setModelDBFile*() method. When the *setModelDBDir*() method is used, the automatically determined filename will be available after a call to the *validateParameters*() or *runModel*() methods.

Return value: string – model database filename.

string getKPhiDBFile

Usage: Returns the name of the file for the K/Phi database. This will be available immediately, when specified using the *setKPhiDBFile*() method. When the *setModelDBDir*() method is used, the automatically determined filename will be available after a call to the *validateParameters*() or *runModel*() methods.

Return value: string – K/Phi database filename.

string getKHMinDBFile

Usage: Returns the name of the file for the K/Hmin database. This will be available immediately, when specified using the *setKHMinDBFile*() method. When the *setModelDBDir*() method is used, the

automatically determined filename will be available after a call to the *validateParameters*() or *runModel*() methods.
Return value: string – K/Hmin database filename.

string getMagfieldDBFile

Usage: Returns the name of the file for the magnetic field model database. This will be available immediately, when specified using the *setMagfieldDBFile*() method. When the *setModelDBDir*() method is used, the automatically determined filename will be available after a call to the *validateParameters*() or *runModel*() methods.
Return value: string – magnetic field model database filename.

string getDoseModelDBFile

Usage: Returns the name of the file for the dose calculation model database. This will be available immediately, when specified using the *setDoseModelDBFile*() method. When the *setModelDBDir*() method is used, the automatically determined filename will be available after a call to the *validateParameters*() or *runModel*() methods.
Return value: string – dose calculation model database filename.

bool getAdiabatic

Usage: Returns the current setting for the calculation of the adiabatic invariant values, as specified in the *setAdiabatic*() method.
Return value: bool – True or False.

string getFluxType

Usage: Returns the type of flux values to be calculated, as specified in the *setFluxType*() method.
Return value: flux type identifier string.

int getNumFluxEnergies

Usage: Returns the number of currently defined flux energy levels, specified in the *setFluxEnergies*() or *setFluxEnergy*() methods.
Return value: int – number of energy levels.

void getFluxEnergies

(dvector& vdEnergies)
Usage: Returns the vector of energy levels, for the model flux calculations, as specified in the *setFluxEnergies*() or *setFluxEnergy*() methods.
Returned parameters:
vdEnergies – vector of energy values, in units of MeV.
Return value: -none-

int getNumFluxEnergies2

Usage: Returns the number of currently defined upper bounds of the flux energy levels, specified in the *setFluxEnergies2*() or *setFluxEnergy2*() methods.
Return value: int – number of energy levels.

void getFluxEnergies2

(dvector& vdEnergies2)

Usage: Returns the upper bounds of the energy bins, for '2PtDiff' flux type, as specified in the *setFluxEnergies2*() or *setFluxEnergy2*() methods.

Returned parameters:

vdEnergies2 – vector of energy values, in units of MeV.

Return value: -none-

int getNumPitchAngles

Usage: Returns the number of currently defined uni-directional pitch angles, specified in either *setPitchAngle*() or *setPitchAngles*() method.

Return value: int – number of pitch angles.

void getPitchAngles

(dvector& vdPitchAngles)

Usage: Returns the vector of the currently defined uni-directional pitch angles, specified in either *setPitchAngle*() or *setPitchAngles*() method.

Returned parameters:

vdPitchAngles – vector of pitch angles.

Return value: -none-

bool getFluxMean

Usage: Returns the current state for the calculation of the 'mean' flux values, as specified in the *setFluxMean*() method.

Return value: bool – True or False.

int getNumFluxPercentiles

Usage: Returns the number of flux percentiles defined for the model calculation, as specified in either *setFluxPercentile*() or *setFluxPercentiles*() method.

Return value: int – number of flux percentiles.

void getFluxPercentiles

(ivector& viPercent)

Usage: Returns the vector of defined flux percentile values, as specified either *setFluxPercentile*() or *setFluxPercentiles*() method.

Returned parameters:

viPercent – vector of percentile flux values.

Return value: -none-

int getNumFluxPerturbedScenarios

Usage: Returns the number of perturbed mean scenarios defined for the model calculation, as specified in the *setFluxPerturbedScenario*(), *setFluxPerturbedScenarios*(), or *setFluxPerturbedScenRange*() method.

Return value: int – number of scenarios.

void getFluxPerturbedScenarios

(ivector& viScenario)

Usage: Returns the vector of defined perturbed flux scenario numbers, as specified in the *setFluxPerturbedScenario*(), *setFluxPerturbedScenarios*(), or *setFluxPerturbedScenRange*() method.

Returned parameters:

viScenario – vector of scenario numbers.

Return value: -none-

int getNumFluxMonteCarloScenarios

Usage: Returns the number of monte carlo scenarios defined for the model calculation, as specified in the *setFluxMonteCarloScenario*() or *setFluxMonteCarloScenarios*(), or *setFluxMonteCarloScenRange*() method.

Return value: int – number of scenarios.

void getFluxMonteCarloScenarios

(ivector& viScenario)

Usage: Returns the vector of defined monte carlo flux scenario numbers, as specified in the *setFluxMonteCarloScenario*() or *setFluxMonteCarloScenarios*(), or *setFluxMonteCarloScenRange*() method.

Returned parameters:

viScenario – vector of scenario numbers.

Return value: -none-

double getMonteCarloEpochTime

Usage: Returns the reference time used in the time progression of the Monte Carlo flux calculations, as specified in the *setMonteCarloEpochTime*() method.

Return value: double – Monte Carlo reference time, in Modified Julian Date form.

bool getMonteCarloFluxPerturb

Usage: Returns the current Monte Carlo perturbation mode, as specified in the *setMonteCarloFluxPerturb*() method.

Return value: bool – True or False.

bool getMonteCarloWorstCase

Usage: Returns the current state for tracking of 'maximum-to-date' Monte Carlo results, as specified in the *setMonteCarloWorstCase*() method.

Return value: bool – True or False.

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 string& strAccumMode)

Usage: Specifies an accumulation mode to be used for the processing of the model flux results; this method may be called multiple times for defining multiple modes. Note: these are saved in the order in which they are defined. When no modes are 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 string: 'Cumul'|'Cumulative', 'Intv'|'Interval', 'Full', 'Boxcar' or 'Expon'|'Exponential' [the 'Boxcar' mode requires both interval and increment values to be defined]

Return value: int – 0 = success, otherwise error

void clearAccumModes

Usage: Removes all previous accumulation modes specified via the *setAccumMode*() method.

Return value: -none-

int setAccumInterval

(const double& dVal)

Usage: Specifies a 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; this method may be called multiple times for defining multiple intervals (a maximum of 9 intervals are allowed); these are saved in ascending order. When no intervals are specified, the accumulation interval 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 time duration value is specified in seconds. When no intervals are specified, the accumulation interval defaults to 86400 seconds (=1.0 days).

Parameters:

dVal – time duration, in units of seconds.

Return value: int – 0 = success, otherwise error

void clearAccumIntervals

Usage: Removes all previous accumulation intervals specified via the *setAccumInterval[Sec]*() method.

Return value: -none-

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. The value may be zero, for self-advancing at the input ephemeris timesteps, or greater than zero, but is required to be less than the specified interval duration.

Return value: int – 0 = success, otherwise error

int setAccumIncrementFrac

(const double& dVal)

Usage: An alternate way to specify the time delta for the shift of the Boxcar accumulation time windows, here expressed as a fraction of the first (smallest) accumulation interval duration (as 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

int setReportTimes

(const double& dTimeRef,
const double& dCadence)

Usage: Specifies the reference time and cadence (in days) for the periodic output of the Boxcar and/or Exponential flux average accumulation results. This method (and/or *setReportTimesSec*) may be called multiple times to build a sequence of successive reporting periods with different cadences.

Parameters:

dTimeRef – Reporting *reference* time, in Modified Julian Date form (this time is <u>not</u> included).

dCadence – cadence of reporting the Boxcar and/or Exponential flux results, in units of days. A cadence value of '0' will halt further reporting of the values at the specified time.

Return value: int – 0 = success, otherwise error

int setReportTimesSec

(const double& dTimeRef,
const double& dCadenceSec)

Usage: Specifies the reference time and cadence (in seconds) for the periodic output of the Boxcar and/or Exponential flux average accumulation results. This method (and/or *setReportTimes*) may be called multiple times to build a sequence of successive reporting periods with different cadences.

Parameters:

dTimeRef – Reporting *reference* time, in Modified Julian Date form (this time is <u>not</u> included).

dCadence – cadence of reporting the Boxcar and/or Exponential flux results, in units of seconds. A cadence value of '0' will halt further reporting of the values at the specified time.

Return value: int – 0 = success, otherwise error

int setReportAtTime

(const double& dTimeVal)

Usage: Specifies a discrete time for the output of the Boxcar and/or Exponential flux average accumulation results. This method may be called multiple times, and may be used in coordination with other calls to the *setReportTimes*() and/or s*etReportTimesSec*() methods.

Parameters:

dTimeVal – Report time, in Modified Julian Date form, of the Boxcar and/or Exponential flux results.

Return value: int – 0 = success, otherwise error

void clearReportTimes

Usage: Removes any previously defined 'ReportTimes' specifications (reference time & cadence).

Return value: -none-

void clearReportAtTime

Usage: Removes any previously defined 'ReportAt' time specifications.
Return value: -none-

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-

---

int getNumAccumModes

Usage: Returns the number of accumulation mode entries defined, as specified in calls to the *setAccumMode*() method.
Return value: int – number of accumulation mode entries.

string getAccumMode

Usage: Returns the *first* accumulation mode, as specified in calls to the *setAccumMode*() method. [If none were specified, the appropriate default mode will be set within a call to the optional *validateParameters*() method, when all parameter specifications have been completed.]
Return value: string – accumulation mode; '-none-' if no modes are defined.

string getAccumModeEntry

(const int& iIdent)
Usage: Returns the accumation mode according to a numeric identifer, based on the *order* of calls to the *setAccumMode*() method.
Parameters:
iIdent – accumulation mode identifier, starting at 1; the maximum valid identifier is equal to the result from the *getNumAccumModes*() method.
Return value: string – accumulation mode; '-none-' if no modes are defined, 'error' if invalid identifier.

void getAccumModes

(vector<string>& vstrAccumModes)
Usage: Returns the vector of the defined accumulation mode entries, as specified in calls to the *setAccumMode*() method.
Returned parameter:
v*strAccumModes* – vector of defined accumulation mode strings; in the order specified.
Return value: -none-

int getNumAccumIntervals

Usage: Returns the number of accumulation intervals defined, as specified in calls to the *setAccumInterval[Sec]*() method.
Return value: int – number of accumulation interval entries.

double getAccumIntervalSec

Usage: Returns the length of the *smallest* accumulation interval specified in calls to the *setAccumInterval[Sec]*() method.
[If none were specified, the default interval will be set within a call to the optional *validateParameters*() method, when all parameter specifications have been completed.]
Return value: double – accumulation interval length, in seconds; negative if an error.

double getAccumIntervalSecEntry

(const int& iIdent)
Usage: Returns the length of the accumulation interval according to a numeric identifier, based on the *ascending* order of the intervals specified in calls to the *setAccumInterval[Sec]*() method.
Parameters:
iIdent – accumulation mode identifier, starting at 1; the maximum valid identifier is equal to the result from the *getNumAccumIntervals*() method.
Return value: double – accumulation interval length, in seconds; negative if an error.

void getAccumIntervalSecs

(dvector& vdAccumIntervals)
Usage: Returns the vector of the defined accumulation interval entries (in ascending order), as specified in calls to the *setAccumInterval[Sec]*() method.
Returned parameter:
v*dAccumIntervals* – vector of defined accumulation interval times, in seconds.
Return value: -none-

double getAccumIncrementSec

Usage: Returns the time delta for the shift of the 'Boxcar' accumulation mode time windows, as specified in the *setAccumIncrementSec*() method.
Return value: double – time delta, in seconds.

double getAccumIncrementFrac

Usage: Returns the interval length fraction for the shift of the Boxcar accumulation time windows, as specified in the *setAccumIncrementFrac*() method.
Return value: double – interval length fraction.

int getNumReportTimes

Usage: Returns the number of 'Report Time' entries defined, as specified in calls to *setReportTimes*() and/or *setReportTimesSec*() methods.
Return value: int – number of 'Report Time' entries.

void getReportTimesSec

(dvector& vdTimeRef,

dvector& vdCadenceSec)

Usage: Returns the vectors of the defined 'Report Time' entries, as specified in calls to *setReportTimes*() and/or *setReportTimesSec*() methods.

Returned parameters:

vdTimeRef – Vector of defined reporting *reference* times, in Modified Julian Date form.

vdCadence – Vector of associated cadence values, in units of seconds.

Return value: -none-

int getNumReportAtTime

Usage: Returns the number of 'Report At' entries defined, as specified in calls to the *setReportAtTime*() method.

Return value: int – number of 'Report At' entries.

void getReportAtTime

(dvector& vdTimeVal)

Usage: Returns the vector of the defined 'Report At' entries, as specified in calls to *setReportAtTime*() method.

Returned parameter:

v*dTimeVal* – Vector of defined report times, in Modified Julian Date form.

Return value: -none-

bool getFluence

Usage: Returns the current state for the calculation of fluence values from the flux.

Return value: bool – True or False, as specified in the *setFluence*() method

---

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. Nominal range: 0.100 – 111.1 mm; 3.937 – 4374 mils; 0.027 – 30.0 g/cm^2 .
Parameters:
vdDepths – vector of depth values
Return value: int – 0 = success, otherwise error

int setDoseDepthUnits

(const 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 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.
Nominal range: 0.100 – 111.1 mm; 3.937 – 4374 mils; 0.027 – 30.0 g/cm^2 .
Parameters:
vdDepths – vector of depth values
strDepthUnits – unit specification: 'millimeters'|'mm', 'mils' or 'gpercm2'
Return value: int – 0 = success, otherwise error

int setDoseDetector

(const 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'|'CaF2', 'Gallium'|'Ga'|'GaAs', 'Lithium'|'Li'|'LiF', 'Glass'|'SiO2', 'Water'|'H2O'
Return value: int – 0 = success, otherwise error

int setDoseGeometry

(const string& strGeometry)
Usage: Specifies the geometry of the aluminum shielding in front of (or around) the detector target.
Parameters:
strGeometry – configuration name: 'Spherical4pi', 'Spherical2pi', 'FiniteSlab' or 'SemiInfiniteSlab'
Return value: int – 0 = success, otherwise error

int setDoseNuclearAttenMode

(const 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

void setDoseWithBrems

(bool bVerdict = *true*)
Usage: Specifies whether the electron dose calculations are to include the bremsstrahlung contributions or not. The default model state is to *include* the bremsstrahlung contributions.
Parameters:
bVerdict – optional flag for including(default) or excluding the bremsstrahlung contributions.
Return value: -none-

void setUseDoseKernel

(bool bVerdict = *true*)
Usage: Specifies the calculation of the dose values using the kernel-based method.
Parameters:
bVerdict – optional flag for activating(default) or deactivating the use of the dose kernel.
Return value: -none-

int setDoseKernelDir

(const string& strDataDir)
Usage: Specifies the directory that contains the collection of detector- and geometry-specific dose kernel XML files. The proper file is automatically selected based on the detector and geometry settings.
Parameters:
strDataDir – directory path for the dose kernel XML files.
Return value: int – 0 = success, otherwise error

int setDoseKernelFile

(const string& strDataSource)
Usage: Explicitly specifies the dose kernel XML file to be used. This XML file ***must*** match the detector and geometry settings, or incorrect results may be produced.
Parameters:
strDataSource – specific Dose kernel XML filename (including path)
Return value: int – 0 = success, otherwise error

---

bool getDoseRate

Usage: Returns the current state for the calculation of dose rate values, as specified in the *setDoseRate*() method.
Return value: bool – True or False.

bool getDoseAccum

Usage: Returns the current state for the calculation of cumulative or accumulated dose, as specified in the *setDoseAccum*() method.
Return value: bool – True or False.

int getNumDoseDepthValues

Usage: Returns the number of aluminum shielding thickness depths, as specified in the *setDoseDepths*() or *setDoseDepthValues*() methods.
Return value: int – number of dose depths.

void getDoseDepthValues

(dvector& vdDepths)
Usage: Returns the list of aluminum shielding thickness depths, as specified in the *setDoseDepths*() or *setDoseDepthValues*() methods. The units of these values may be obtained from the *getDoseDepthUnits*() method.
Returned parameters:
vdDepths – vector of depth values.
Return value: -none-

string getDoseDepthUnits

Usage: Returns the measurement units associated with the depth values, as specified in the *setDoseDepths*() or *setDoseDepthUnits*() methods.
Return value: string – Dose depth units.

string getDoseDetector

Usage: Returns the dose detector material type that lies behind the aluminum shielding, as specified in the *setDoseDetector*() method.
Return value: string – material name.

string getDoseGeometry

Usage: Returns the geometry of the aluminum shielding in front of (or around) the detector target, as specified in the *setDoseGeometry*() method.
Return value: string – Dose shielding geometry.

string getDoseNuclearAttenMode

Usage: Returns the 'Nuclear Attenuation' mode used during the ShieldDose2 model calculations, as specified in the *setDoseNuclearAttenMode*() method.
Return value: string – Dose calculation nuclear attenuation mode.

bool getDoseWithBrems

Usage: Returns the current state for the inclusion of the bremsstrahlung contribution in the electron dose values calculated, as specified in the *setDoseWithBrems*() method.
Return value: bool – True or False.

bool getUseDoseKernel

Usage: Returns the current state for the dose values calculated using the kernel method, as specified in the *setUseDoseKernel*() method.
Return value: bool – True or False.

string getDoseKernelDir

Usage: Returns the directory that contains the dose kernel XML files, as specified in the *setDoseKernelDir*() method.
Return value: string – directory for the dose kernel XML files.

string getDoseKernelFile

Usage: Returns the XML filename to be used for kernel-based dose calculations, as specified in the *setDoseKernelFile*() method.
Return value: string – Dose kernel XML filename.

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 setAggregMedian

Usage: Adds the 50% value to the list for aggregation confidence level calculations.

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

int setAggregConfLevels

(const ivector& viPercent)
Usage: Specifies the calculation of one or more confidence level values. The list of values supersedes any prior calls for defining these confidence level percentages.
Parameters:
viPercent – vector of percent values for the aggregation confidence level calculations (0-100 valid).
Return value: int – 0 = success, otherwise error

void clearAggregConfLevels

Usage: Clears the accumulated list of percent values for aggregation confidence level calculations.

Return value: -none-

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. <u>Use of this method is strongly discouraged</u>.*

Return value: int – 0 = success, otherwise error

---

int getNumAggregConfLevels

Usage: Returns the number of confidence levels for the aggregation calculations, as specified in *setAggregConfLevel*(), *setAggregConfLevels*(), or through *setAggregMean*() or *setAggregMedian*() methods.
Return value: int – number of confidence levels.

void getAggregConfLevels

(ivector& viPercent)
Usage: Returns the list of confidence level values, as specified in *setAggregConfLevel*(), *setAggregConfLevels*(), or through *setAggregMean*() or *setAggregMedian*() methods.
Returned parameters:
viPercent – vector of percent values for the aggregation confidence level calculations
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 setLegActivityLevel

(const 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 setLegActivityRange

(const 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 setLeg15DayAvgAp

(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 setLegFixedEpoch

(bool bFixedEpoch)

Usage: Specifies the use of the model-specific fixed epoch (year) for the magnetic field model in the flux calculations. It is <u>*highly recommended*</u> 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 setLegShiftSAA

(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-

---

string getLegActivityLevel

Usage: Returns the geomagnetic activity level parameter for the CRRESPRO, AE8 or AP8 Legacy model, as specified in the *seLegtActivityLevel*() method.
Return value: string – geomagnetic activity level specification.

string getLegActivityRange

Usage: Returns the geomagnetic activity level parameter for the CRRESELE Legacy model, as specified in the *setLegActivityRange*() method.
Return value: string – geomagnetic activity level specification.

double getLeg15DayAvgAp

Usage: Returns the 15-day average Ap value for the CRRESELE Legacy model, as specified in the *setLeg15DayAvgAp*() method.
Return value: double – 15-day average Ap value.

bool getLegFixedEpoch

Usage: Returns the current setting for the use of the model-specific fixed epoch, as specified in the *setLegFixedEpoch*() method.
Return value: bool – True or False.

bool getLegShiftSAA

Usage: Returns the current setting of shifting the SAA from its fixed-epoch location, as specified in the *setLegShiftSAA*() method.
Return value: bool – True or False.

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 setCamMagfieldModel

(const 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 setCamDataFilter

(const 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 setCamPitchAngleBin

(const 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 setCamSpecies

(const 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

---

string getCamMagfieldModel

Usage: Returns the magnetic field option for the CAMMICE Legacy model, as specified in the *setCamMagfieldModel*() method.
Return value: string – magnetic field model specification.

string getCamDataFilter

Usage: Returns the data filter option for the CAMMICE Legacy model, as specified in the *setCamDataFilter*() method.
Return value: string – data filter specification.

string getCamPitchAngleBin

Usage: Returns the pitch angle bin for the CAMMICE Legacy model, as specified in the *setCamPitchAngleBin*() method
Return value: string – pitch angle bin identification.

string getCamSpecies

Usage: Returns the particle species for the CAMMICE Legacy model, as specified in the *setCamSpecies*() method.
Return value: string – species identification.

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*Data*() methods.

All returned 'integral flux' values are in units of [#/cm^2/sec]; and their fluences in [#/cm^2].
All returned 'differential flux' values are in units of [#/cm^2/sec/MeV]; and their fluences in [#/cm^2/MeV].
All returned 'dose rate' values are in units of [rads/sec]; 'accumulated dose' values are in units of [rads].

Important: Please note that these *flyin**() and *get*Data*() methods return the requested data in 'chunks', defaulting to 960 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. Note that the 'flyin' methods ultimately call the 'getModelData' method under the hood, so is accessing the same data 'stream' (based on the data type and percentile/scenario identifier, as well as the accumulation mode and interval identifier).

The *flyin**() and *get*Data*() methods include optional arguments for specifying an accumulation mode and accumulation interval identifier. These are used to distinguish exactly which set of data is to be returned when there are accumulation(s) active. For the 'flux' data type, the 'default' accumulation mode translates to 'cumul', but for all other data types, it translates to the *first* accumulation mode that was defined. The values returned for the 'Cumul' accumulation mode are the "raw" values that are calculated at the input ephemeris times. The values returned for any other accumulation mode (with one or more defined time interval periods) will be computed averages or summations, depending on the data type; their associated time is for the end of the interval.

Some of the data access methods also return the ephemeris position information. These returned vectors of ephemeris position values are in the coordinate system and units previously specified (or accessed via the *getCoordSys*() and *getCoordUnit*() methods). Please consult the User's Guide document, "Supported Coordinate Systems" table for more details; in particular, note the 'standard' ordering of these returned coordinate values for non-Cartesian coordinate systems. For the accumulation data requests which are averages or summations (ie flux average, fluence, dose accumulation), the associated ephemeris values are purposely set to zero, as the data values do not correspond to a single discrete position.

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.

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)

int getCoordSys

```
( 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: int – 0 = success, otherwise error

int getCoordSysUnits

```
( 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: int – 0 = success, otherwise error

int getNumDir

Usage: Returns the number of data directions in the generated flux data.
Return value: int – number of directions: 1 when 'omnidirectional', otherwise, number of defined pitch angles. (when 'omnidirectional', the *getNumPitchAngles*() method returns 0).

int flyinMean

```
( vdvector& vvdFluxData,
   const string& strAccumMode = "default",
   const int& iAccumIntvId = 1 )
```

Usage: Returns the 'Mean' model flux values for omni-directional model runs. A previous call to the *setFluxMean(<true>)* method is required for these results to be available for the Ae9/Ap9/SPM models. This method may also be used for accessing flux results from the 'Legacy' models. Flux accumulated average values may be obtained using this method when specifying the optional arguments.
The returned flux values are in units of [#/cm2/sec] (integral) or [#/cm2/sec/MeV] (differential).
Parameters:

vvvdFluxData – returned **2**-dimensional vector of the 'mean' flux values. [time, energy]
strAccumMode – (optional) accumulation mode of the flux values to return. The availability of the flux accumulated average values for the specified accumulation mode is dependent on those modes defined in previous call(s) to the *setAccumMode*() method. See that method description for more information. The default mode of 'default' translates to 'Cumul'; in this context is the raw (*non*-accumulated) flux values.
iAccumIntvId – (optional) accumulation interval identifier for the flux accumulated average values; valid values: 1 through result from *getNumAccumIntervals*() method. This argument is only needed when requesting values for intervals other than the smallest, and is dependent on those intervals defined in previous calls to the *setAccumInterval[Sec]*() method.
Return value: int – number of records returned (≥0 = success, <0 = error)

int flyinMean

```
( vvdvector& vvvdFluxData,
  const string& strAccumMode = "default",
  const int& iAccumIntvId = 1 )
```

Usage: Returns the 'Mean' model flux values. A previous call to the *setFluxMean(<true>)* method is required for these results to be available for the Ae9/Ap9/SPM models. This method may also be used for accessing flux results from the 'Legacy' models. Flux accumulated average values may be obtained using this method when specifying the optional arguments.
The returned flux values are in units of [#/cm2/sec] (integral) or [#/cm2/sec/MeV] (differential).
Parameters:
vvvdFluxData – returned **3**-dimensional vector of the 'mean' flux values. [time,energy,direction]
strAccumMode – (optional) accumulation mode of the flux values to return. The availability of the flux accumulated average values for the specified accumulation mode is dependent on those modes defined in previous call(s) to the *setAccumMode*() method. See that method description for more information. The default mode of 'default' translates to 'Cumul'; in this context is the raw (*non*-accumulated) flux values.
iAccumIntvId – (optional) accumulation interval identifier for the flux accumulated average values; valid values: 1 through result from *getNumAccumIntervals*() method. This argument is only needed when requesting values for intervals other than the smallest, and is dependent on those intervals defined in previous calls to the *setAccumInterval[Sec]*() method.
Return value: int – number of records returned (≥0 = success, <0 = error)

int flyinMean

```
( dvector& vdTimes
  dvector& vdCoord1,
  dvector& vdCoord2,
  dvector& vdCoord3,
  vdvector& vvdPitchAngles,
  vvdvector& vvvdFluxData,
  const string& strAccumMode = "default",
  const int& iAccumIntvId = 1 )
```

Usage: Returns the 'Mean' model flux values, along with the associated ephemeris values and pitch angles. A previous call to the *setFluxMean(<true>)* method is required for these results to be available for the Ae9/Ap9/SPM models. This method may also be used for accessing flux results from the 'Legacy' models. Flux accumulated average values may be obtained using this method when specifying the optional arguments.
The returned flux values are in units of [#/cm2/sec] (integral) or [#/cm2/sec/MeV] (differential).

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. When the accumulation mode is other than 'Cumul', *all* coordinate values will be zero.
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]
strAccumMode – (optional) accumulation mode of the flux values to return. The availability of the flux accumulated average values for the specified accumulation mode is dependent on those modes defined in previous call(s) to the *setAccumMode*() method. See that method description for more information. The default mode of 'default' translates to 'Cumul'; in this context is the raw (*non*-accumulated) flux values.
iAccumIntvId – (optional) accumulation interval identifier for the flux accumulated average values; valid values: 1 through result from *getNumAccumIntervals*() method. This argument is only needed when requesting values for intervals other than the smallest, and is dependent on those intervals defined in previous calls to the *setAccumInterval[Sec]*() method.
Return value: int – number of records returned (≥0 = success, <0 = error)

int flyinPercentile

```
( const int& iPercentile,
  vvdvector& vvvdFluxData,
  const string& strAccumMode = "default",
  const int& iAccumIntvId = 1 )
```

Usage: Returns the 'Percentile' model flux values for the specified percentile. The percentile number must be one of those included in previous calls to the *setFluxPercentile()* methods. Flux accumulated average values may be obtained using this method when specifying the optional arguments.
The returned flux values are in units of [#/cm2/sec] (integral) or [#/cm2/sec/MeV] (differential).
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]
strAccumMode – (optional) accumulation mode of the flux values to return. The availability of the flux accumulated average values for the specified accumulation mode is dependent on those modes defined in previous call(s) to the *setAccumMode*() method. See that method description for more information. The default mode of 'default' translates to 'Cumul'; in this context is the raw (*non*-accumulated) flux values.
iAccumIntvId – (optional) accumulation interval identifier for the flux accumulated average values; valid values: 1 through result from *getNumAccumIntervals*() method. This argument is only needed when requesting values for intervals other than the smallest, and is dependent on those intervals defined in previous calls to the *setAccumInterval[Sec]*() method.
Return value: int – number of records returned (≥0 = success, <0 = error)

int flyinPercentile

```
( const int& iPercentile,
  dvector& vdTimes,
  dvector& vdCoord1,
  dvector& vdCoord2,
  dvector& vdCoord3,
```

vdvector& vvdPitchAngles,
vvdvector& vvvdFluxData,
const string& strAccumMode = “default”,
const int& iAccumIntvId = 1)

Usage: Returns the ‘Percentile’ model flux values for the specified percentile, along with the associated ephemeris values and pitch angles. The percentile number must be one of those included in previous calls to the *setFluxPercentile()* methods. Flux accumulated average values may be obtained using this method when specifying the optional arguments.
The returned flux values are in units of [#/cm2/sec] (integral) or [#/cm2/sec/MeV] (differential).

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. When the accumulation mode is other than ‘Cumul’, *all* coordinate values will be zero.

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]

strAccumMode – (optional) accumulation mode of the flux values to return. The availability of the flux accumulated average values for the specified accumulation mode is dependent on those modes defined in previous call(s) to the *setAccumMode*() method. See that method description for more information. The default mode of ‘default’ translates to ‘Cumul’; in this context is the raw (*non*-accumulated) flux values.

iAccumIntvId – (optional) accumulation interval identifier for the flux accumulated average values; valid values: 1 through result from *getNumAccumIntervals*() method. This argument is only needed when requesting values for intervals other than the smallest, and is dependent on those intervals defined in previous calls to the *setAccumInterval[Sec]*() method.

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

int flyinPerturbedMean

(const int& iScenario,
vvdvector& vvvdFluxData,
const string& strAccumMode = “default”,
const int& iAccumIntvId = 1)

Usage: Returns the ‘Perturbed Mean’ model flux values for the specified scenario number. The scenario number must be one of those included in previous calls to the *setFluxPerturbedScen*() or *setFluxPerturbedScenRange*() methods. Flux accumulated average values may be obtained using this method when specifying the optional arguments.
The returned flux values are in units of [#/cm2/sec] (integral) or [#/cm2/sec/MeV] (differential).

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]

strAccumMode – (optional) accumulation mode of the flux values to return. The availability of the flux accumulated average values for the specified accumulation mode is dependent on those modes defined in previous call(s) to the *setAccumMode*() method. See that method description for more information. The default mode of ‘default’ translates to ‘Cumul’; in this context is the raw (*non*-accumulated) flux values.

iAccumIntvId – (optional) accumulation interval identifier for the flux accumulated average values; valid values: 1 through result from *getNumAccumIntervals*() method. This argument is only needed when requesting values for intervals other than the smallest, and is dependent on those intervals defined in previous calls to the *setAccumInterval[Sec]*() method.

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

int flyinPerturbedMean

```
( const int& iScenario,
  dvector& vdTimes,
  dvector& vdCoord1,
  dvector& vdCoord2,
  dvector& vdCoord3,
  vdvector& vvdPitchAngles,
  vvdvector& vvvdFluxData,
  const string& strAccumMode = "default",
  const int& iAccumIntvId = 1 )
```

Usage: Returns the 'Perturbed Mean' model flux values for the specified scenario number, along with the associated ephemeris values and pitch angles. The scenario number must be one of those included in previous calls to the *setFluxPerturbedScen[Range]*() methods. Flux accumulated average values may be obtained using this method when specifying the optional arguments.
The returned flux values are in units of [#/cm2/sec] (integral) or [#/cm2/sec/MeV] (differential).

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. When the accumulation mode is other than 'Cumul', *all* coordinate values will be zero.

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]

strAccumMode – (optional) accumulation mode of the flux values to return. The availability of the flux accumulated average values for the specified accumulation mode is dependent on those modes defined in previous call(s) to the *setAccumMode*() method. See that method description for more information. The default mode of 'default' translates to 'Cumul'; in this context is the raw (*non*-accumulated) flux values.

iAccumIntvId – (optional) accumulation interval identifier for the flux accumulated average values; valid values: 1 through result from *getNumAccumIntervals*() method. This argument is only needed when requesting values for intervals other than the smallest, and is dependent on those intervals defined in previous calls to the *setAccumInterval[Sec]*() method.

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

int flyinMonteCarlo

```
( const int& iScenario,
  vvdvector& vvvdFluxData,
  const string& strAccumMode = "default",
  const int& iAccumIntvId = 1 )
```

Usage: Returns the 'Monte Carlo' model flux values for the specified scenario number. The scenario number must be one of those included in previous calls to the *setFluxMonteCarloScen*() or *setFluxMonteCarloScenRange()* methods. Flux accumulated average values may be obtained using this method when specifying the optional arguments.
The returned flux values are in units of [#/cm2/sec] (integral) or [#/cm2/sec/MeV] (differential).
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]
strAccumMode – (optional) accumulation mode of the flux values to return. The availability of the flux accumulated average values for the specified accumulation mode is dependent on those modes defined in previous call(s) to the *setAccumMode*() method. See that method description for more information. The default mode of 'default' translates to 'Cumul'; in this context is the raw (*non*-accumulated) flux values.
iAccumIntvId – (optional) accumulation interval identifier for the flux accumulated average values; valid values: 1 through result from *getNumAccumIntervals*() method. This argument is only needed when requesting values for intervals other than the smallest, and is dependent on those intervals defined in previous calls to the *setAccumInterval[Sec]*() method.
Return value: int – number of records returned (≥0 = success, <0 = error)

int flyinMonteCarlo

```
( const int& iScenario,
  dvector& vdTimes,
  dvector& vdCoord1,
  dvector& vdCoord2,
  dvector& vdCoord3,
  vdvector& vvdPitchAngles,
  vvdvector& vvvdFluxData,
  const string& strAccumMode = "default",
  const int& iAccumIntvId = 1 )
```

Usage: Returns the 'Monte Carlo' model flux values for the specified scenario number, along with the associated ephemeris values and pitch angles. The scenario number must be one of those included in previous calls to the *setFluxMonteCarloScen*() or *setFluxMonteCarloScenRange()* methods. Flux accumulated average values may be obtained using this method when specifying the optional arguments.
The returned flux values are in units of [#/cm2/sec] (integral) or [#/cm2/sec/MeV] (differential).
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. When the accumulation mode is other than 'Cumul', *all* coordinate values will be zero.
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]
strAccumMode – (optional) accumulation mode of the flux values to return. The availability of the flux accumulated average values for the specified accumulation mode is dependent on those modes defined

in previous call(s) to the *setAccumMode*() method. See that method description for more information. The default mode of 'default' translates to 'Cumul'; in this context is the raw (*non*-accumulated) flux values.

iAccumIntvId – (optional) accumulation interval identifier for the flux accumulated average values; valid values: 1 through result from *getNumAccumIntervals*() method. This argument is only needed when requesting values for intervals other than the smallest, and is dependent on those intervals defined in previous calls to the *setAccumInterval[Sec]*() method.

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

int getModelData

```
( const string& strDataType,
  const string& strFluxMode,
  const int& iCalcVal,
  dvector& vdTimes,
  dvector& vdCoord1,
  dvector& vdCoord2,
  dvector& vdCoord3,
  vdvector& vvdPitchAngles,
  vvdvector& vvvdData,
  const string& strAccumMode = "default",
  const int& iAccumIntvId = 1 )
```

Usage: Returns the model results from for the specified data type, flux mode and, if applicable, the percentile or scenario identifier, and the accumulation mode and interval identifier. See the following routine for accessing *aggregation* results. The associated ephemeris and pitch angles are also returned. The returned flux values are in units of [#/cm2/sec] (integral) or [#/cm2/sec/MeV] (differential). The returned dose values are in units of [rads/sec] (doserate) or [rads] (doseaccum).

Parameters:

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

strFluxMode – flux mode identifier: "mean"|"percent"|"perturbed"|"montecarlo"|"montecarloWC"

iCalcVal – additional model data qualifier: *ignored* for 'mean' flux mode; model percentile (1-99) for 'percent'; model scenario number (1-999) for 'perturbed' or 'montecarlo'. The number must be one of those included in previous calls to the appropriate flux mode parameter specification methods.

vdTimes – returned vector of ephemeris time values, in Modified Julian Date form; when using an accumulation mode other than 'Cumul', 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. When the accumulation mode is other than 'Cumul', *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]

strAccumMode – (optional) accumulation mode of the flux values to return. The availability of the flux accumulated average values for the specified accumulation mode is dependent on those modes defined in previous call(s) to the *setAccumMode*() method. See that method description for more information. The default mode of 'default' translates to 'Cumul'; in this context is the raw (*non*-accumulated) flux values.

iAccumIntvId – (optional) accumulation interval identifier for the flux accumulated average values; valid values: 1 through result from *getNumAccumIntervals*() method. This argument is only needed when requesting values for intervals other than the smallest, and is dependent on those intervals

defined in previous calls to the *setAccumInterval[Sec]*() method.
Return value: int – number of records returned (≥0 = success, <0 = error)

int getAggregData

(const string& strDataType,
const string& strFluxMode,
const int& iPercent,
dvector& vdTimes,
dvector& vdCoord1,
dvector& vdCoord2,
dvector& vdCoord3,
vdvector& vvdPitchAngles,
vvdvector& vvvdData,
const string& strAccumMode = “default”,
const int& iAccumIntvId = 1)

Usage: Returns the confidence level results from multiple scenarios of data input, for the specified data type, flux mode, confidence level percent, and accumulation mode and interval identifier. The associated ephemeris and pitch angles are also returned.
The returned flux values are in units of [#/cm2/sec] (integral) or [#/cm2/sec/MeV] (differential).
The returned dose values are in units of [rads/sec] (doserate) or [rads] (doseaccum).
Parameters:
strDataType – data type identifier: "flux"|"fluence"|"doserate"|"doseaccum"
strFluxMode – flux mode identifier: "perturbed"|"montecarlo"|"montecarloWC"
iPercent – aggregation confidence level percent (0-100 *or -1 for mean*). The number 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; for an accumulation mode other than ‘Cumul’, 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. When the accumulation mode is other than ‘Cumul’, *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]
strAccumMode – (optional) accumulation mode of the flux values to return. The availability of the flux accumulated average values for the specified accumulation mode is dependent on those modes defined in previous call(s) to the *setAccumMode*() method. See that method description for more information. The default mode of ‘default’ translates to ‘Cumul’; in this context is the raw (*non*-accumulated) flux values.
iAccumIntvId – (optional) accumulation interval identifier for the flux accumulated average values; valid values: 1 through result from *getNumAccumIntervals*() method. This argument is only needed when requesting values for intervals other than the smallest, and is dependent on those intervals defined in previous calls to the *setAccumInterval[Sec]*() method.
Return value: int – number of records returned (≥0 = success, <0 = 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. The availability of the adiabatic coordinates and magnetic field information is dependent on a previous call to the *setAdiabatic*(<true>) method.

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 – number of records returned (≥0 = success, <0 = 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. If omnidirectional, values returned are for pitch angle of 90. The availability of the adiabatic coordinates and magnetic field information is dependent on a previous call to the *setAdiabatic*(<true>) method.
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.
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 McIlwain 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 – number of records returned (≥0 = success, <0 = 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*Dat*() data access methods will return data starting at the

beginning of the ephemeris input times and positions.
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. *Special Note*: the *setDelBinDir* setting only applies to when the Application class object is destroyed when the program completes, or on subsequent calls to *runModel*.

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.

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

void resetOrbitParameters

Usage: Resets input parameters related to orbit specifications to their initial default values.

Return value: -none-

void resetParameters

Usage: Resets *all* model run input parameters to their initial default values. Only the ExecDir and WindowsMpiMode parameters retain their settings.

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 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 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

Model-Level C++ API Reference

These classes provide direct programmatic access to each of the model/processing components that comprise the CmdLineIrene 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 and/or directories.

EphemModel Class

Header file: CEphemModel.h

This class is the entry point that provides direct programmatic access to the ephemeris generation model. Additional supporting methods are available to perform coordinate conversions and calculate associated magnetic field model parameter values.
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.

Important: 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 <u>*entire*</u> period as defined in the *setTimes*(), *setVarTimes*() or *setTimesList*() 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
Return values: -none-

~EphemModel

Usage: Destructor
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 960, 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.
Return value: number of entries in processing chunk

Model Parameter Inputs:

int setModelDBDir

(const string& strDataDir)
Usage: Specifies the directory that contains the collection IRENE model database files. The various database files required are automatically selected according to the model and parameters specified. The use of this method is highly recommended, as it *eliminates* the need for the other methods that specify the individual database files; those are only needed for using alternate or non-standard versions.
Parameters:
strDataDir – directory path for the IRENE database files.
Return value: int – 0 = success, otherwise error

int setMagfieldDBFile

(const string& strDataSource)
Usage: Specifies the name of the file (including path) for the magnetic field model database. The use of this method is *not needed* when *setModelDBDir*() is specified, except when an alternate or non-standard database file is to be used (this overrides any automatic file specification).
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 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 string& strMode,
const 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-

---

string getModelDBDir

Usage: Returns the directory name containing the collection of IRENE model database files that was specified in a previous call to the *setModelDBDir*() method; otherwise, blank.
Return value: string – model database directory.

string getMagfieldDBFile

Usage: Returns the name of the file for the magnetic field model database. This will be available immediately, when specified using the *setMagfieldDBFile*() method. When the *setModelDBDir*() method is used, the automatically determined filename will be available after a call to the *computeEphemeris*() method.
Return value: string – magnetic field model database filename.

string getPropagator

Usage: Returns the orbit propagator identifier, as specified in the *setPropagator*() method.
Return value: string – orbit propagator name.

string getSGP4Mode

Usage: Returns the SGP4 propagator mode setting, as specified in the *setSGP4Param*() method.
Return value: string – SGP4 mode setting.

string getSGP4WGS

Usage: Returns the SGP4 propagator WGS setting, as specified in the *setSGP4Param*() method.
Return value: string – SGP4 WGS setting.

bool getKeplerUseJ2

Usage: Returns the Kepler propagator 'J2' perturbation setting, as specified in the *setKeplerUseJ2*() method.
Return value: bool – True or False.

---

int setTimes

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

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

Parameters:

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

dTimeStepSec - time step size, in seconds; must be greater than zero.

Return value: int – 0 = success, otherwise error

int setVarTimes

(const double& dStartTime,
 const double& dEndTime,
 const double& dTimeMinStepSec,
 const double& dTimeMaxStepSec = 3600.0,
 const double& dTimeRoundSec = 5.0)

Usage: Specifies the start and stop times (*inclusive*), and variable time step limits, of the ephemeris information to be generated by the orbit propagator from the defined orbital element values or TLE file. Variable time steps are calculated based on the orbital radial values, and are useful for the more elliptical orbits (ie eccentricity>0.25). See the User's Guide document "Orbit Ephemeris File Description" section for more information.

Parameters:

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

dTimeMinStepSec – lower limit of variable time steps, in seconds; must be ≥ 10 seconds

dTimeMaxStepSec – upper limit of variable time steps, in seconds; must be > min, ≤ 3600 seconds

dTimeRoundSec – rounding of variable time steps, in whole seconds; use 0 for no rounding; < min

Return value: int – 0 = success, otherwise error

int setStartTime

(const double& dStartTime)

Usage: Specifies the start time of the ephemeris information to be generated by the orbit propagator from the defined orbital element values or TLE file.

Parameters:

dStartTime - start time value, in Modified Julian Date form

Return value: int – 0 = success, otherwise error

int setEndTime

(const double& dEndTime)

Usage: Specifies the end time of the ephemeris information to be generated by the orbit propagator from the defined orbital element values or TLE file.

Parameters:

dEndTime - end time value, in Modified Julian Date form

Return value: int – 0 = success, otherwise error

int setTimeStep

(const double& dTimeStepSec)

Usage: Specifies the fixed time step, in seconds, of the ephemeris information to be generated by the orbit propagator from the defined orbital element values or TLE file.

Parameters:

dTimeStepSec - time step size, in seconds; must be greater than zero.

Return value: int – 0 = success, otherwise error

int setVarTimeStep

(const double& dTimeMinStepSec,
const double& dTimeMaxStepSec = 3600.0,
const double& dTimeRoundSec = 5.0)

Usage: Specifies the variable time step limits, of the ephemeris information to be generated by the orbit propagator from the defined orbital element values or TLE file. Variable time steps are calculated based on the orbital radial values, and are useful for the more elliptical orbits (ie eccentricity>0.25). See the *User's Guide* document "Orbit Ephemeris File Description" section for more information.

Parameters:

dTimeMinStepSec – lower limit of variable time steps, in seconds; must be ≥ 10 seconds
dTimeMaxStepSec – upper limit of variable time steps, in seconds; must be > min, ≤ 3600 seconds
dTimeRoundSec – rounding of variable time steps, in whole seconds; use 0 for no rounding; < min

Return value: int – 0 = success, otherwise error

int setTimesList

(const dvector& vdTimes)

Usage: Specifies a specific set of time values for the ephemeris information to be generated by the orbit propagator from the defined 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

---

void getTimes

(double& dStartTime,
double& dEndTime,
double& dTimeStepSec)

Usage: Returns the start and stop times, and fixed time step, for the ephemeris generation.

Returned parameters:

dStartTime, *dEndTime* - start and stop time values, in Modified Julian Date form
dTimeStepSec – fixed time step, in seconds

Return value: -none-

void getVarTimes

(double& dStartTime,
double& dEndTime,
double& dTimeMinStepSec,
double& dTimeMaxStepSec,

double& dTimeRoundSec)

Usage: Returns the start and stop times, and variable time step limits, for the ephemeris generation.

Returned parameters:

dStartTime, *dEndTime* - start and stop time values, in Modified Julian Date form
dTimeMinStepSec – lower limit of variable time steps, in seconds
dTimeMaxStepSec – upper limit of variable time steps, in seconds
dTimeRoundSec – rounding of variable time steps, in seconds

Return value: -none-

double getStartTime

Usage: Returns start time defined for the ephemeris generation, as specified in the *setTimes*() or *setStartTime()* methods.

Return value: double – start time, in Modified Julian Date form.

double getEndTime

Usage: Returns end time, defined for the ephemeris generation, as specified in the *setTimes*() or *setEndTime()* methods.

Return value: double – end time, in Modified Julian Date form.

double getTimeStep

Usage: Returns fixed time step defined for the ephemeris generation, as specified in the *setTimeStep*() or *setTimeStep()* methods.

Return value: double – time step, in seconds.

void getVarTimeStep

(double& dTimeMinStepSec,
double& dTimeMaxStepSec,
double& dTimeRoundSec)

Usage: Returns the variable time step limits, if defined, for the ephemeris generation.

Returned parameters:

dTimeMinStepSec – lower limit of variable time steps, in seconds
dTimeMaxStepSec – upper limit of variable time steps, in seconds
dTimeRoundSec – rounding of variable time steps, in seconds

Return value: -none-

int getNumTimesList

Usage: Returns the number of time entries defined for the ephemeris generation, from the specifications in *setTimes*() or *setTimesList*() methods. Note: number of *variable* timestep times (from *setVarTimes*() specifications) is available only when the orbital parameters are sufficiently defined.

Return value: int – number of ephemeris times defined; if negative, an error

int getTimesList

(dvector& vdTimes)

Usage: Returns the vector of time values, in Modified Julian Date form, for the ephemeris generation, from the specifications in *setTimes*() or *setTimesList*() methods. Note: the *variable* timestep times (from *setVarTimes*() specifications) are available only when the orbital parameters are sufficiently defined.

Returned parameters:
*vdTime*s – vector of time values, in Modified Julian Date form
Return value: int – number of ephemeris times defined; if negative, an error

void clearTimesList

Usage: Clears the time entries defined for the ephemeris generation, from the specifications in *setTimes*() or *setTimesList*() methods, or *setVarTimes*(), under certain conditions.
Return value: -none-

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 file may contain multiple entries (for the same satellite), but must be in chronological order.

int setTLEFile

(const 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

string getTLEFile

Usage: Returns the name of the specified TLE file, if any.
Return value: string – path and filename of the TLE file, as specified in the *setTLEFile*() method.

---

int setTLE

(const string strTLELine1,
const string strTLELine2)
Usage: Specifies a pair of strings to be used as a two-line element (TLE) set that describes an orbit. It is assumed that the contents of these strings are properly formatted, as described in Appendix F of the User's Guide. Improperly formatted strings can sometimes produce valid, but unintended orbits. This routine may be called multiple times so to define multiple TLEs, provided that they are for the same object, and are added in chronologically order (with no duplicate epoch times).
Parameters:
strTLELine1 – first line of TLE set
strTLELine2 – second line of TLE set
Return value: int – 0 = success, otherwise error

int setTLE

(const vector<string> vstrTLELine1s,
const vector<string> vstrTLELine2s)
Usage: Specifies a pair of vectors of strings to be used as two-line element (TLE) sets that describes an orbit over time. It is assumed that the contents of these strings are properly formatted, as described in Appendix F of the User's Guide. Improperly formatted strings can sometimes produce valid, but

unintended orbits. When multiple TLEs are defined, they must be for the same object, and in chronological order (with no duplicate epoch times).

Parameters:

vstrTLELine1s – vector of strings for the first line of each TLE set
vstrTLELine2s – vector of strings for the second line of each TLE set

Return value: int – 0 = success, otherwise error

void getTLE

(vector<string> vstrTLELine1s,
vector<string> vstrTLELine2s)

Usage: Returns the vectors of strings of the TLEs that were defined in calls to either form of the *setTLE*() method.

Parameters:

vstrTLELine1s – vector of strings for the first line of each TLE set
vstrTLELine2s – vector of strings for the second line of each TLE set

Return value: int – 0 = success, otherwise error

void resetTLE

Usage: Clears the list of TLEs that were defined in calls to the *setTLE*() method.
Return value: -none-

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 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 (-10 – 10)

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 (-1 – 1)

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 (-1 – 1)
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 (>0, but <~20Re)
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 (>0, but <~20Re)
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, in unit of Re.
Parameters:

dSemiMajorAxis – semi-major axis length (1-75), 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 setStateVectors

(const dvector& vdPos,
const dvector& vdVel)
Usage: Specifies the satellite's position and velocity in the GEI coordinate system at the element's 'epoch' time. Alternatively, the *setPositionGEI*() and *setVelocityGEI*() method could be used instead.
Parameters:
v*dPos* – vector containing the GEI coordinate system satellite position values (X,Y,Z), in km
vdVel – vector containing the GEI coordinate system satellite velocity values (X,Y,Z), in km/sec
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 (>1Re, but <~75Re)
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 resetOrbitParameters

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.
Return value: -none-

double getElementTime

Usage: Returns the 'epoch' time associated with the set of orbital element values, as specified in the *setElementTime*() method.
Return value: double – element 'epoch' time, in Modified Julian Date form.

double getInclination

Usage: Returns the orbital element 'Inclination' value, as specified in the *setInclination*() method.
Return value: double – orbit inclination angle, in degrees.

double getRightAscension

Usage: Returns the orbital element 'Right Ascension of the Ascending Node' value, as specified in the setRightAscension() method.
Return value: double – orbit ascending node position, in degrees.

double getEccentricity

Usage: Returns the orbital element 'Eccentricity' value, as specified in the *setEccentricity*() method.
Return value: double – orbit eccentricity value (unitless).

double getArgOfPerigee

Usage: Returns the orbital element 'Argument of Perigee' value, as specified in the *setArgOfPerigee*() method.
Return value: double – orbit perigee position, in degrees.

double getMeanAnomaly

Usage: Returns the orbital element 'Mean Anomaly' value, as specified in the *setMeanAnomaly*() method.
Return value: double – orbit mean anomaly value, in degrees.

double getMeanMotion

Usage: Returns the orbital element 'Mean Motion' value, as specified in the *setMeanMotion*() method.
Return value: double – orbit mean motion value, in revolutions per day.

double getMeanMotion1stDeriv

Usage: Returns the orbital element 'First Time Derivative of the Mean Motion' value, as specified in the *setMeanMotion1stDeriv*() method.
Return value: double – first derivative of mean motion.

double getMeanMotion2ndDeriv

Usage: Returns the orbital element 'Second Time Derivative of the Mean Motion' value, as specified in the *setMeanMotion2ndDeriv*() method
Return value: double – second derivative of mean motion.

double getBStar

Usage: Returns the orbital element 'B*' value, as specified in the *setBStar*() method.

Return value: double – ballistic coefficient value.

double getAltitudeOfApogee

Usage: Returns the orbital element 'Apogee Altitude' value, as specified in the *setAltitudeOfApogee*() method.
Return value: double – orbit apogee altitude, in km.

double getAltitudeOfPerigee

Usage: Returns the orbital element 'Perigee Altitude' value, as specified in the *setAltitudeOfPerigee*() method.
Return value: double – orbit perigee altitude, in km.

double getLocalTimeOfApogee

Usage: Returns the local time of the orbit's apogee, as specified in the *setLocalTimeOfApogee*() method.
Return value: double – local time of orbit perigee, in hours + fraction.

double setLocalTimeMaxInclination

Usage: Returns the local time of the orbit's maximum inclination, as specified in the *setLocalTimeMaxInclination*() method.
Return value: double – local time of orbit maximum inclination, in hours + fraction.

double getTimeOfPerigee

Usage: Returns the time of the orbit's perigee, as specified in the *setTimeOfPerigee*() method.
Return value: double – orbit perigee time value, in Modified Julian Date form.

double getSemiMajorAxis

Usage: Returns the orbit's semi-major axis length, as specified in the *setSemiMajorAxis*() method.
Return value: double – orbit semi-major axis length, in units of Re (1 Re = 6371.2 km)

double getGeosynchLon

Usage: Returns the geographic longitude of satellite in a geosynchronous orbit, as specified in the *setGeosynchLon*() method.
Return value: double – orbit geosynchronous (East) longitude, in degrees

void getStateVectors

```
( dvector& vdPos,
  dvector& vdVel )
```

Usage: Returns the satellite's position and velocity vector values, as specified in the *setStateVectors*() method, or in the *setPositionGEI*() and *setVelocityGEI*() methods.
Parameters:
v*dPos* – vector containing the GEI coordinate system satellite position values (X,Y,Z), in km
vdVel – vector containing the GEI coordinate system satellite velocity values (X,Y,Z), in km/sec
Return value: -none-

void getPositionGEI

(double& dPosX,
double& dPosY,
double& dPosZ)

Usage: Returns the satellite's position vector, as specified in either the *setPositionGEI*() or *setStateVectors*() method.

Parameters:

dPosX, dPosY, dPosZ – GEI coordinate system satellite position values, in km

Return value: -none-

void getVelocityGEI

(double& dVelX,
double& dVelY,
double& dVelZ)

Usage: Returns the satellite's velocity vector, as specified in either the *setVelocityGEI*() or *setStateVectors*() method.

Parameters:

dVelX, dVelY, dVelZ – GEI coordinate system satellite velocity values, in km/sec

Return value: -none-

int setMainField

(const string& strMainField)

Usage: Defines the main magnetic field model to be used in coordinate conversions and magnetic field model calculations. The IRENE standard uses the 'FastIGRF' model, and is the default setting. Note that the use of the dipole main field models require the external field model to be set to 'None'.

Parameters:

strMainField – identifier for the 'main' magnetic field model:

'FastIGRF', 'IGRF', 'OffsetDipole' | 'Offset', 'TitltedDipole' | 'Tilted'

Return value: int – 0 = success, otherwise error

int setExternalField

(const string& strExternalField)

Usage: Defines the external magnetic field model to be used in coordinate conversions and magnetic field model calculations. The IRENE standard uses the 'OlsonPfitzer' model, and is the default setting. Note that the use of the dipole main field models require the external field model to be set to 'None'. The use of the Tsyganenko89 external field model also requires the specification of the Kp index value, defined with the *setKpValue*() or *setKpValues*() methods.

Parameters:

strExternalField – identifier for the 'external' magnetic field model:

'None', 'OlsonPfitzer' | 'OP', 'Tsyganenko89' | 'Tsyg89' | 'T89'

Return value: int – 0 = success, otherwise error

string getMainField

Usage: Returns the identifier for the 'main' magnetic field model that was specified in a previous call to the *setMainField*() method; otherwise, the default setting.

Return value: string – 'main' magnetic field model identifier.

string getExternalField

Usage: Returns the identifier for the 'external' magnetic field model that was specified in a previous call to the *setExternalField*() method; otherwise, the default setting.
Return value: string – 'external' magnetic field model identifier.

int setKpValue

(const double& dKpVal)
Usage: Specifies the constant Kp index value to be used in all subsequent calculations (as needed). Any previously defined list of time history Kp values using the *setKpValues*() method is cleared.
Parameters:
dKpVal – Kp Index value. Valid range = 0.0 – 9.0.
Return value: int – 0 = success, otherwise error

int setKpValues

(const double& dRefTime,
const dvector& vdKpVals)
Usage: Defines a list of time history of three-hour Kp index values to be used in all subsequent calculations (as needed); the appropriate Kp index value will be determined from the current calculation's time setting vs the specified reference time. The first Kp index value will be used when the current time is prior to the reference time; the last Kp index value will be used when the current time is beyond the defined time history limit. This automated determination of Kp supersedes any previously defined constant Kp value using the *setKpValue*() method.
Parameters:
dRefTime - reference time value, in Modified Julian Date form; must be at 0000 GMT of day.
v*dKpVals* – vector containing time history of three-hour Kp index values. Valid range = 0.0 – 9.0.
Return value: int – 0 = success, otherwise error

double getKpValue

Usage: Returns the current defined Kp value. This may be the constant value specified in the *setKpValue*() method, or the appropriate entry from the time history value defined in the *setKpValues*() method, according to the most recently used calculation time value.
Return value: double – current Kp index value.

double getKpValuesRefTime

Usage: Returns the reference time of the currently defined time history of Kp values, as specified in the *setKpValues*() method. A value of -1.0 will be returned if no time history is defined.
Return value: double – reference time, in MJD form, for the current Kp index history.

double getKpValuesEndTime

Usage: Returns the end time of the currently defined time history of Kp values, as specified in the *setKpValues*() method. A value of -1.0 will be returned if no time history is defined.
Return value: double – end time, in MJD form, for the current Kp index history.

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 generation time range and (fixed or variable) time step be defined (or list of discrete times), and the orbit described by either using TLEs or an appropriate set of element values and reference 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*(), *setVarTimes*() or *setTimesList*() 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 string& strCoordSys,
  const 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*(), *setVarTimes*() or *setTimesList*() 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.

Return value: -none-

int convertCoordinates

```
( const string& strCoordSys,
  const string& strCoordUnits,
  const dvector& vdTimes,
  const dvector& vdCoord1,
  const dvector& vdCoord2,
  const dvector& vdCoord3,
  const string& strNewCoordSys,
  const 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 string& strCoordSys,
  const string& strCoordUnits,
  const double& dTime,
  const double& dCoord1,
  const double& dCoord2,
  const double& dCoord3,
  const string& strNewCoordSys,
  const 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.
dTime – 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

int computeBfield

```
( const string& strCoordSys,
  const string& strCoordUnits,
  const dvector& vdTimes,
  const dvector& vdCoord1,
  const dvector& vdCoord2,
  const dvector& vdCoord3,
  vdvector& vvdBVecGeo,
  dvector& vdBMag,
  dvector& vdBMin,
  dvector& vdLm )
```

Usage: Calculates several magnetic field parameters for the specified time and position inputs. Results will be affected by specifications for the 'main' and 'external' magnetic field model, using the *setMainField*() and *setExternalField*() methods, respectively. When the 'Tsyg89' external field model is being used, an additional specification of the Kp index value will be required, using the *setKpValue*() or *setKpValues*() methods. Depending on the position(s) specified, some or all return values may not be able to be determined; when this occurs, *fill* values are returned instead (ie, -1.0e31 or ±100.0). In specific cases, the returned valid Lm value(s) may be negated; this indicates that the underlying calculations included a magnetic field line trace that briefly went subterranean.

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.

vvdBVecGeo – returned 2-dimensional vector of B vector component values [nT], at each of the times+positions specified; B vectors are in the GEO coordinate system. [time,coord]

vdBMag – returned vector of B vector magnitude values [nT], at each of the times+positions specified.

vdBMin – returned vector of B minimum vector magnitude values [nT], at the magnetic equator associated with each of the times+positions specified.

vdLm – returned vector of Lshell values [unitless], associated with each of the times+positions specified.

Return value: int – 0 = success, otherwise error

int computeBfield

```
( const string& strCoordSys,
  const string& strCoordUnits,
  const double& dTime,
  const double& dCoord1,
  const double& dCoord2,
  const double& dCoord3,
  dvector& vdBVecGeo,
  double& dBMag,
  double& dBMin,
  double& dLm )
```

Usage: Calculates several magnetic field parameters for the specified time and position input. Results will be affected by specifications for the 'main' and 'external' magnetic field model, using the *setMainField*() and *setExternalField*() methods, respectively. When the 'Tsyg89' external field model is being used, an additional specification of the Kp index value will be required, using the *setKpValue*() or *setKpValues*() methods. Depending on the position specified, some or all return values may not be able to be determined; when this occurs, *fill* values are returned instead (ie, -1.0e31 or ±100.0). In specific cases, the returned valid Lm value may be negated; this indicates that the underlying calculations included a magnetic field line trace that briefly went subterranean.

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.

dTime – 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.

vdBVecGeo – returned vector of B vector component values [nT], at the time+position specified; B vector is in the GEO coordinate system.

dBMag – returned B vector magnitude value [nT], at the time+position specified.

dBMin – returned B minimum vector magnitude value [nT], at the magnetic equator associated with the time+position specified.

dLm – returned Lshell value [unitless], associated with the time+position specified.

Return value: int – 0 = success, otherwise error

int computeInvariants

```
( const string& strCoordSys,
  const string& strCoordUnits,
  const dvector& vdTimes,
  const dvector& vdCoord1,
  const dvector& vdCoord2,
  const dvector& vdCoord3,
  const dvector& vdPitchAngles,
  dvector& vdBMin,
  vdvector& vvdBMinPosGeo,
  vdvector& vvdBVecGeo,
  vdvector& vvdLm,
  vdvector& vvdI )
```

Usage: Calculates magnetic field parameters as a function of pitch angle for the specified time and position inputs. Results will be affected by specifications for the 'main' and 'external' magnetic field model, using the *setMainField*() and *setExternalField*() methods, respectively. When the 'Tsyg89' external field model is being used, an additional specification of the Kp index value will be required, using the *setKpValue*() or *setKpValues*() methods. Depending on the position(s) specified, some or all return values may not be able to be determined; when this occurs, *fill* values are returned instead (ie, -1.0e31, ±100.0 or -1.0). In specific cases, the returned valid Lm value(s) may be negated; this indicates that the underlying calculations included a magnetic field line trace that briefly went subterranean.

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.

vdPitchAngles – vector of pitch angles, in degrees (0-90). Must be in *descending* order.

vdBMin – returned vector of B minimum vector magnitude values [nT], at the magnetic equator associated with each of the times+positions specified.

vvdBMinPosGeo – returned 2-dimensional vector of B minimum positions, associated with each of the times+positions specified; BMin positions are in the GEO/km coordinate system. [time,coord]

vvdBVecGeo – returned 2-dimensional vector of B vector component values [nT], at each of the times+positions specified; B vectors are in the GEO coordinate system. [time,coord]

vvdLm – returned 2-dimensional vector of Lshell values [unitless], associated with each of the times+positions specified, for each of the pitch angles specified. [time,pitchAngle]

vvdI – returned 2-dimensional vector of “I” values [unitless], associated with each of the times+positions specified, for each of the pitch angles specified. [time,pitchAngle]

Return value: int – 0 = success, otherwise error

int computeInvariants

```
( const string& strCoordSys,
  const string& strCoordUnits,
  const double& dTime,
  const double& dCoord1,
  const double& dCoord2,
  const double& dCoord3,
  const dvector& vdPitchAngles,
  double& dBMin,
  dvector& vdBMinPosGeo,
  dvector& vdBVecGeo,
  dvector& vdLm,
  dvector& vdI )
```

Usage: Calculates magnetic field parameters as a function of pitch angle for the specified time and position input. Results will be affected by specifications for the ‘main’ and ‘external’ magnetic field model, using the *setMainField*() and *setExternalField*() methods, respectively. When the ‘Tsyg89’ external field model is being used, an additional specification of the Kp index value will be required, using the *setKpValue*() or *setKpValues*() methods. Depending on the position specified, some or all return values may not be able to be determined; when this occurs, *fill* values are returned instead (ie, -1.0e31, ±100.0 or -1.0). In specific cases, the returned valid Lm value(s) may be negated; this indicates that the underlying calculations included a magnetic field line trace that briefly went subterranean.

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.

dTime – 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.

vdPitchAngles – vector of pitch angles, in degrees (0-90). Must be in *descending* order.

dBMin – returned B minimum vector magnitude value [nT], at the magnetic equator associated with the time+position specified.

vdBMinPosGeo – returned vector of B minimum position, associated with the time+position specified; BMin position is in the GEO/km coordinate system.

vdBVecGeo – returned vector of B vector component values [nT], at the time+position specified; B vector is in the GEO coordinate system.

vdLm – returned vector of Lshell values [unitless], associated with the time+position specified, for each of the pitch angles specified.

vdI – returned vector of "I" values [unitless], associated with the times+positions specified, for each of the pitch angles specified.

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 models. 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;`
Return values: -none-

~Ae9Ap9Model

Usage: Destructor
Return values: -none-

Model Parameter Inputs:

int setModel

(const string& strModel)
Usage: Specifies the name of the flux model to be used in the calculations. Note the 'Plasma' model names now includes the species type.
Parameters:
strModel – model name: 'AE9', 'AP9', 'PlasmaE', 'PlasmaH', 'PlasmaHe' or 'PlasmaO'
Return value: int – 0 = success, otherwise error

int setModelDBDir

(const string& strDataDir)
Usage: Specifies the directory that contains the collection IRENE model database files. The various database files required are automatically selected according to the model and parameters specified. <u>The use of this method is highly recommended</u>, as it *eliminates* the need for the other methods that specify the individual database files; those are only needed for using alternate or non-standard versions.
Parameters:
strDataDir – directory path for the IRENE database files.
Return value: int – 0 = success, otherwise error

int setModelDBFile

(const string& strModelDBFile)

Usage: Specifies the name of the database file (including path) for flux model calculations. The use of this method is *not needed* when *setModelDBDir*() is specified, except when an alternate or non-standard database file is to be used (this overrides any automatic file specification).

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 setKPhiDBFile

(const string& strKPhiDBFile)

Usage: Specifies the name of the file (including path) for the K/Phi database. The use of this method is *not needed* when *setModelDBDir*() is specified, except when an alternate or non-standard database file is to be used (this overrides any automatic file specification).

This database name is in the form of '<path>/fastPhi_net.mat'.

Parameters:

strDataSource – database filename, including path

Return value: int – 0 = success, otherwise error

int setKHMinDBFile

(const string& strKHMinDBFile)

Usage: Specifies the name of the file (including path) for the K/Hmin database. The use of this method is *not needed* when *setModelDBDir*() is specified, except when an alternate or non-standard database file is to be used (this overrides any automatic file specification).

This database name is in the form of '<path>/fast_hmin_net.mat'.

Parameters:

strDataSource – database filename, including path

Return value: int – 0 = success, otherwise error

int setMagfieldDBFile

(const string& strMagfieldDBFile)

Usage: Specifies the name of the file (including path) for the magnetic field model database. The use of this method is *not needed* when *setModelDBDir*() is specified, except when an alternate or non-standard database file is to be used (this overrides any automatic file specification).

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 specifications *cannot* be changed.

Return value: int – 0 = success, otherwise error

string getModelName

Usage: Returns the name of the model that is described by the model database file, either specified in a previous call to *setModelDBFile*() method, or automatically determined using the specifications in calls to the *setModel*() and *setModelDBDir*() methods. This returned model name is available only after a call to either the *loadModelDB*() or *setFluxEnvironment*() methods .

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

string getModelSpecies

Usage: Returns the name of the particle species of the model database file, either specified in the previous call to *setModelDBFile*() method, or automatically determined using the specification in calls to the *setModel*() and *setModelDBDir*() methods. This returned species name is available only after a call to either the *loadModelDB*() or *setFluxEnvironment*() methods.

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

string getModel

Usage: Returns the name of the flux model, as specified in the *setModel*() method.

Return value: string – model name.

string getModelDBDir

Usage: Returns the directory name containing the collection of IRENE model database files that was specified in a previous call to the *setModelDBDir*() method; otherwise, blank.

Return value: string – model database directory.

string getModelDBFile

Usage: Returns the name of the database file for flux model calculations. This will be available immediately, when specified using the *setModelDBFile*() method. When the *setModelDBDir*() method is used, the automatically determined filename will be available after a call to either the *loadModelDB*() or *setFluxEnvironment*() methods.

Return value: string – string – model database filename.

string getKPhiDBFile

Usage: Returns the name of the file for the K/Phi database. This will be available immediately, when specified using the *setKPhiDBFile* () method. When the *setModelDBDir*() method is used, the automatically determined filename will be available after a call to either the *loadModelDB*() or *setFluxEnvironment*() methods.

Return value: string – string – K/Phi database filename.

string getKHMinDBFile

Usage: Returns the name of the file for the K/Hmin database. This will be available immediately, when specified using the *setKHMinDBFile* () method. When the *setModelDBDir*() method is used, the automatically determined filename will be available after a call to either the *loadModelDB*() or *setFluxEnvironment*() methods.

Return value: string – K/Hmin database filename.

string getMagfieldDBFile

Usage: Returns the name of the file for the magnetic field model database. This will be available immediately, when specified using the *setMagfieldDBFile* () method. When the *setModelDBDir*() method is used, the automatically determined filename will be available after a call to either the *loadModelDB*() or *setFluxEnvironment*() methods

Return value: string – magnetic field model database filename.

Model Execution and Results:

The *setFluxEnvironment*() method is used to specify the ephemeris (time and position) and particle energies [MeV] 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**() (or *computeFlux**()) methods return the requested type and mode of flux values using the most recently defined 'flux environment' specifications.
The returned flux values are in units of [#/cm^2/sec] (for integral) or [#/cm^2/sec/MeV] (for differential).

For best model performance, it is recommended that the amount of ephemeris information being supplied as input to the *setFluxEnvironment*() method be controlled. 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 efficiently use the system memory resources, set the number of entries in the time and coordinate vectors for each call: a value of 120 is advised for systems with limited memory, 960 for typical systems, but no larger than 2400, even with ample amounts available memory. The subsequent calls to the desired *flyin**() 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 string& strFluxType,
  const dvector& vdEnergies,
  const dvector& vdEnergies2,
  const dvector& vdTimes,
  const string& strCoordSys,
  const string& strCoordUnits,
  const dvector& vdCoords1,
  const dvector& vdCoords2,
  const dvector& vdCoords3 )
```

Usage: Specifies the flux type, energies [MeV] 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 [MeV]; if ‘2PtDiff’ flux type, these specify lower bounds of energy bins.
vdEnergies2 – ignored unless flux type is ‘2ptDiff’; vector of energy values [MeV], 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 string& strFluxType,
  const dvector& vdEnergies,
  const dvector& vdEnergies2,
  const dvector& vdTimes,
  const string& strCoordSys,
  const string& strCoordUnits,
  const dvector& vdCoords1,
  const dvector& vdCoords2,
  const dvector& vdCoords3,
  const dvector& vdPitchAngles )
```

Usage: Specifies the flux type, energies [MeV] 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 [MeV]; if ‘2PtDiff’ flux type, these specify lower bounds of energy bins.
vdEnergies2 – ignored unless flux type is ‘2ptDiff’; vector of energy values [MeV], 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 string& strFluxType,
  const dvector& vdEnergies,
  const dvector& vdEnergies2,
  const dvector& vdTimes,
  const string& strCoordSys,
  const string& strCoordUnits,
  const dvector& vdCoords1,
  const dvector& vdCoords2,
  const dvector& vdCoords3,
  const vdvector& vvdPitchAngles )
```

Usage: Specifies the flux type, energies [MeV] 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 [MeV]; if '2PtDiff' flux type, these specify lower bounds of energy bins.

vdEnergies2 – ignored unless flux type is '2ptDiff'; vector of energy values [MeV], 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*vdPitchAngles* – 2-dimensional vector of pitch angles, in degrees (0-180). [time,pitch angles]

Return value: int – 0 = success, otherwise error

int setFluxEnvironment

```
( const string& strFluxType,
  const dvector& vdEnergies,
  const dvector& vdEnergies2,
  const dvector& vdTimes,
  const string& strCoordSys,
  const 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 [MeV] 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 [MeV]; if '2PtDiff' flux type, these specify lower bounds of energy bins.

vdEnergies2 – ignored unless flux type is '2ptDiff'; vector of energy values [MeV], 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 as the position vectors. These may be a unit vector, or in the same 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

The following routines are used to compute the flux values in the specific mode, according to the most recent call to one of the '*setFluxEnvironment()*' methods (described above).
All returned 'integral flux' values are in units of [#/cm^2/sec].
All returned 'differential flux' values are in units of [#/cm^2/sec/MeV].

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.
The returned flux values are in units of [#/cm^2/sec] (integral) or [#/cm^2/sec/MeV] (differential).
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.
The returned flux values are in units of [#/cm^2/sec] (integral) or [#/cm^2/sec/MeV] (differential).
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.
The returned flux values are in units of [#/cm^2/sec] (integral) or [#/cm^2/sec/MeV] (differential).
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.
The returned flux values are in units of [#/cm^2/sec] (integral) or [#/cm^2/sec/MeV] (differential).
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.
The returned flux values are in units of [#/cm^2/sec] (integral) or [#/cm^2/sec/MeV] (differential).
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 *loadBuffer*() method is used collect data for each 'chunk'; the desired *compute**() methods *must be called <u>each</u> 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
Return values: -none-

~AccumModel

Usage: Destructor
Return values: -none-

Model Parameter Inputs:

int setTimeInterval

(const double& dIntervalDays)
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:
dIntervalDays – time duration, in units of days+fraction *[must be greater than smallest timestep of ephemeris]*
Return value: int – 0 = success, otherwise error

int setTimeIntervalSec

(const double& dInterval)
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:
dInterval – time duration, in seconds *[must be greater than smallest timestep of ephemeris]*
Return value: int – 0 = success, otherwise error

int setTimeIncrement

(const double& dIncrem)
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:
dIncrem – time delta, in seconds, for the increment of time between the start of adjacent Boxcar time windows. May be zero, for self-advancing at the input ephemeris timesteps, or be greater than zero, but less than the value specified in the *setTimeInterval*() method.
Return value: int – 0 = success, otherwise error

---

double getTimeInterval

Usage: Returns the time duration of the accumulation of time-tagged data for use in the calculation of integrated data results.
Return value: double – time duration, in units of days+fraction, as specified in the *setTimeInterval*() method.

double getTimeIncrement

Usage: Returns the time delta for the shift of the 'Boxcar' accumulation mode time windows.
Return value: double – time delta, in seconds, for the increment of time between the start of adjacent Boxcar time windows, as specified in the *setTimeIncrement*() method.

void setWcLogging

(bool bVerdict)
Usage: Specifies the state of the collection of 'worst case' logging information. When not specified, the default state is *false*. When active, this information is determined and collected during calls to the *applyWorstToDate*() method; the results are obtained via subsequent calls to *getWorstCaseLog*() method.
Parameters:
bVerdict – set to *true* to activate the logging option, or *false* to deactivate it.
Return value: -none-

Model Execution and Results:

int loadBuffer

(const dvector& vdTimes,
const vvdvector& vvvdData)
Usage: Loads the sets of input time-tagged data into the accumulation buffer, replacing any previous buffer contents.
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: Adds a single set of input time-tagged data to the current contents of the accumulation buffer.

Parameters:
dTime – time value, in Modified Julian Date form
vvdData – 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* <u>*no*</u> *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 <u>*completed*</u> 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 <u>no</u> 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 <u>*completed*</u> 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 the oldest incomplete boxcar accumulation interval, if any. Set to *true* only after *all* data has been processed. Also causes the boxcar accumulation information to be reset.

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 bFinal = false )
```

Usage: Processes the current contents of the data buffer, then returns the exponential average flux data results at the input data times; the calculations are dependent on the interval value specified in a call to the *setTimeInterval[Sec]*() method.

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 input data entries. [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.

bFinal – optional flag for process completion. 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. A prior call to the *setWcLogging*(true) method makes a log of this information available, accessible via the *getWorstCaseLog*() method.

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

int getWorstCaseLog

(const dvector &vdTimes,
const dvector &vdEnergies,
const vdvector &vvdPitchAngles,
vdvector &vvdWorstCaseLog)

Usage: Returns log entries of the information for each of the 'worst case' events (if any) detected during the preceeding call to the *applyWorstToDate*() method. Each log entry contains the time, energy, pitch angle (-1 if omni-directional) and data value of the 'worst case' event detected. A prior call to the *setWcLogging*(true) method is required for this information to be available; if not, an error is returned. A call to this method will sometimes return *no* log entries; this is normal, as the occurrence of 'worst case' events is wholly dependant on the characteristics of the processed data values with respect to any of the previous data values.

Parameters:

vdTimes – vector of times associated with the data values processed in the *applyWorstToDate*() method, in Modified Julian Date form

vdEnergies – vector of the energy values associated with the data values.

vvdPitchAngles – 2-dimensional vector of the pitch angles associated with the data values. When the same set of pitch angles are used for all times, this parameter may be a single vector entry.

vvvdWorstCaseLog – returned 2-dimensional vector of the corresponding 'worst case' entries, where each vector entry contains the time, energy value, pitch angle and 'worst case' data value.

Return value: int – 0 or greater = number of log entries returned, 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.

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.

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.

void resetBoxcarFluence

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

void resetExponentialFlux

Usage: clears the internally stored exponential flux average data. Subsequent calls to c*omputeExponentialFlux*() method will start a new set of exponential flux average calculations.

---

double getFluenceStartTime

Usage: Returns the starting time for the cumulative fluence accumulation data.
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.
Return value: double – fluence accumulation data start time, in MJD form.

double getFullFluenceStartTime

Usage: Returns the starting time for the full fluence accumulation data.
Return value: double – fluence accumulation data start time, in MJD form.

double getBoxcarFluenceStartTime

Usage: Returns the starting time for the oldest active boxcar interval of fluence accumulation data.
Return value: double – boxcar fluence accumulation data start time, in MJD form.

double getLastLength

Usage: Returns the last interval time duration of the most recent *compute*Fluence*|*Flux*() method call in which the 'bReturnPartial' input parameter was set to *true*.
Return value: double – last interval time duration, in seconds. -1.0 if no partial interval occurred.

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 <u>*must*</u> be 1pt Differential, Omni-directional fluxes, otherwise dose results are invalid.

General:

DoseModel

Usage: Default constructor
Return values: -none-

~DoseModel

Usage: Destructor
Return values: -none-

Model Parameter Inputs:

int setModelDBDir

(const string& strDataDir)
Usage: Specifies the directory that contains the collection IRENE model database files. The various database files required are automatically selected according to the model and parameters specified. <u>The use of this method is highly recommended</u>, as it *eliminates* the need for the other methods that specify the individual database files; those are only needed for using alternate or non-standard versions.
Parameters:
strDataDir – directory path for the IRENE database files.
Return value: int – 0 = success, otherwise error

int setModelDBFile

(const string& strModelDBFile)
Usage: Specifies the name of the file (including path) for the dose calculation model database. The use of this method is *not needed* when *setModelDBDir*() is specified, except when an alternate or non-standard database file is to be used (this overrides any automatic file specification).
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 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,
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,
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.

Nominal range: 0.100 – 111.1 mm; 3.937 – 4374 mils; 0.027 – 30.0 g/cm^2 .

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 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’ | ‘CaF2’, ‘Gallium’|’Ga’ | ‘GaAs’, ‘Lithium’|’Li’ | ‘LiF’, ‘Glass’|’SiO2’, ‘Water’|’H2O’

Return value: int – 0 = success, otherwise error

int setGeometry

(const string& strGeometry)

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

Parameters:

strGeometry – configuration name: ‘Spherical4pi’, ‘Spherical2pi’, ‘FiniteSlab’ or ‘SemiInfiniteSlab’

Return value: int – 0 = success, otherwise error

int setNuclearAttenMode

(const string& strNucAttenMode)

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

Parameters:

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

Return value: int – 0 = success, otherwise error

void setWithBrems

(bool bVerdict = *true*)

Usage: Specifies whether the electron dose calculations are to include the bremsstrahlung contributions or not. The default model state is to *include* the bremsstrahlung contributions.
Parameters:
bVerdict – optional flag for including(default) or excluding the bremsstrahlung contributions.
Return value: -none-

---

string getModelDBDir

Usage: Returns the directory name containing the collection of IRENE model database files that was specified in a previous call to the *setModelDBDir*() method; otherwise, blank.
Return value: string – model database directory.

string getModelDBFile

Usage: Returns the name of the database file used for ShieldDose2 model calculations. This will be available immediately, when specified using the *setModelDBFile*() method. When the *setModelDBDir*() method is used, the automatically determined filename will be available after a call to one of the *compute**() methods.
Return value: string – model database filename.

string getSpecies

Usage: Returns the particle species for the dose calculations, as specified in the *setSpecies*() method.
Return value: string – species identification.

int getNumEnergies

Usage: Returns the number of energies, as specified in the *setEnergies*() method.
Return value: int – number of energies.

int getNumDepths

Usage: Returns the number of aluminum shielding thickness depths, as specified in the *setDepths*() method.
Return value: int – number of dose depths.

string getDetector

Usage: Returns the dose detector material type that lies behind the aluminum shielding, as specified in the *setDetector*() method.
Return value: string – material name.

string getGeometry

Usage: Returns the geometry of the aluminum shielding in front of (or around) the detector target, as specified in the *setGeometry*() method.
Return value: string – Dose shielding geometry.

string getNuclearAttenMode

Usage: Returns the ‘Nuclear Attenuation’ mode used during the dose calculations, as specified in the *setNuclearAttenMode*() method.

Return value: string – Dose calculation nuclear attenuation mode.

bool getWithBrems

Usage: Returns the current state for the inclusion of the bremsstrahlung contribution in the electron dose values calculated, as specified in the *setWithBrems*() method.
Return value: bool – True or False.

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 <u>omni-directional</u> <u>differential</u> 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' [rads/sec]; an input of fluence values returns the associated accumulated dose value [rads].
Return value: int – 0 = success, otherwise error

int computeFluxDoseRate

(const vdvector& vvdFluxData,
vdvector& vvdDoseRate)

Usage: Returns the modeled dose rate values [rads/sec], 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 <u>omni-directional</u> <u>differential</u> 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 rates [rads/sec], 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 [rads/sec], 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 rates [rads/sec], 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 [rads], 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 values [rads], for a single direction, over the shielding depths previously specified via the *setDepths*() method. [time,depths,direction]
Return value: int – 0 = success, otherwise error

DoseKernel Class

Header file:	CDoseKernel.h

This class is the entry point that provides direct programmatic access to the kernel-based method 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. The Dose Kernel xml files were produced based on results from the ShieldDose2 model. For more information, see Appendix K of the User's Guide document.

General:

DoseKernel

Usage: Default constructor
Return values: -none-

~DoseKernel

Usage: Destructor
Return values: -none-

Model Parameter Inputs:

int setKernelXmlDir

(const string& strKernelDir)
Usage: Specifies the directory path for the collection of dose-specific kernel xml files.
(The proper file is automatically selected based on the specified geometry and detector material.)
Parameters:
strKernelDir – dose kernel xml file collection directory path
Return value: int – 0 = success, otherwise error

int setKernelXmlFile

(const string& strKernelXmlFile)
Usage: Specifies the name of the file (including path) for the kernel-based dose calculation method.
The dose xml file specified *must* correspond to the specified geometry and detector material.
Parameters:
strKernelXmlFile – dose kernel xml filename, including path
Return value: int – 0 = success, otherwise error

int setSpecies

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

int setEnergies

(const dvector& vdEnergies,

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,
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.

Nominal range: 0.100 – 111.1 mm;   3.937 – 4374 mils;   0.027 – 30.0 g/cm^2 .

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 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’ | ‘CaF2’, ‘Gallium’|’Ga’ | ‘GaAs’, ‘Lithium’|’Li’ | ‘LiF’, ‘Glass’|’SiO2’, ‘Water’|’H2O’

Return value: int – 0 = success, otherwise error

int setGeometry

(const string& strGeometry)

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

Parameters:

strGeometry – configuration name: ‘Spherical4pi’, ‘Spherical2pi’, ‘FiniteSlab’ or ‘SemiInfiniteSlab’

Return value: int – 0 = success, otherwise error

int setNuclearAttenMode

(const string& strNucAttenMode)

Usage: Specifies the ‘Nuclear Attenuation’ mode of the kernel dose calculations.

Parameters:

strNucAttenMode – attenuation mode: ‘None’

(other modes of ‘NuclearInteractions’ and ‘NuclearAndNeutrons’ are currently unsupported)

Return value: int – 0 = success, otherwise error

void setWithBrems

(bool bVerdict = *true*)

Usage: Specifies whether the electron dose calculations are to include the bremsstrahlung contributions or not. The default model state is to *include* the bremsstrahlung contributions.
Parameters:
bVerdict – optional flag for including(default) or excluding the bremsstrahlung contributions.
Return value: -none-

---

string getKernelXmlPath

Usage: Returns the path for the dose kernel xml files used for dose calculations, as specified in the *setKernelXmlPath*() method.
Return value: string – dose kernel xml path.

string getKernelXmlFile

Usage: Returns the name of the dose kernel xml file used for dose calculations, as specified in the *setKernelXmlFile*() method.
Return value: string – dose kernel xml filename.

string getSpecies

Usage: Returns the particle species for the dose calculations, as specified in the *setSpecies*() method.
Return value: string – species identification.

int getNumEnergies

Usage: Returns the number of energies, as specified in the *setEnergies*() method.
Return value: int – number of energies.

int getNumDepths

Usage: Returns the number of aluminum shielding thickness depths, as specified in the *setDepths*() method.
Return value: int – number of dose depths.

string getDetector

Usage: Returns the dose detector material type that lies behind the aluminum shielding, as specified in the *setDetector*() method.
Return value: string – material name.

string getGeometry

Usage: Returns the geometry of the aluminum shielding in front of (or around) the detector target, as specified in the *setGeometry*() method.
Return value: string – Dose shielding geometry.

string getNuclearAttenMode

Usage: Returns the 'Nuclear Attenuation' mode used for the kernel dose calculations, as specified in the *setNuclearAttenMode*() method.
Return value: string – Dose calculation nuclear attenuation mode.

bool getWithBrems

Usage: Returns the current state for the inclusion of the bremsstrahlung contribution in the electron dose values calculated, as specified in the *setWithBrems*() method.
Return value: bool – True or False.

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' [rads/sec]; an input of fluence values returns the associated accumulated dose value [rads].
Return value: int – 0 = success, otherwise error

int computeFluxDoseRate

(const vdvector& vvdFluxData,
vdvector& vvdDoseRate)

Usage: Returns the modeled dose rate values [rads/sec], 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 rates [rads/sec], 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 [rads/sec], 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 rates [rads/sec], 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 [rads], 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 <u>omni-directional</u> <u>differential</u> fluence values, over the energy levels previously specified via the *setEnergies*() method, for a <u>single</u> 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 values [rads], 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.
<u>Important Note</u>: 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
Return values: -none-

~AggregModel

Usage: Destructor
Return values: -none-

Model Parameter Inputs:

void resetAgg

Usage: Clears all time-tagged data from the scenario data aggregation.
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

int getAggDimensions

(int& iDim1,
int& iDim2,
int& iDim3)

Usage: Returns the data dimensions of current scenario data.

Parameters:
iDim1, iDim2, iDim3 – returned data dimensions of current scenario data [time,energy|depth,direction]
Return value: int – 0 = success, otherwise error

void getAggNumScenarios

Usage: Returns the number of aggregation scenarios defined.
Return value: int – 0 or greater = number of aggreg scarios, otherwise error

void setWcLogging

(bool bVerdict)
Usage: Specifies the state of the collection of 'worst case' logging information. When not specified, the default state is *false*. When active, this information is determined and returned in calls to the *getConfLevelWcLog*() method.
Parameters:
bVerdict – set to *true* to activate the logging option, or *false* to deactivate it.
Return value: -none-

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 data 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 data 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

int findConfLevelWc

(const vvdvector &vvvdPercData,
vdvector &vvdPercMaxData)

Usage: Generates internal log entries of the information for each of the 'worst case' events (if any) detected within the input confidence level data. Each log entry contains the time, energy, pitch angle (-1 if omni-directional), associated scenario number and data value of the 'worst case' event detected. A prior call to the *setWcLogging*(true) method is required for this logging information to be collected; if not, an error is returned. The number of log entries is returned; this can be zero, as the occurrence of 'worst case' events is wholly dependant on the characteristics of the processed data values with respect to any of the previous data values (via the vvdPercMaxData information). The collected log information may be accessed using the *getConfLevelWcLog*() method, called immediately after this method.

Parameters:

vvvdPercData – 3-dimensional vector of the calculated confidence level data values, as returned from the *computeConfLevel*() or *computeMedian*() methods

vvdPercMaxData – input and returned 2-dimensional vector of the (current and previously) determined maximum data values [energy, direction]. A 'reset' of these maximum values is implied when an empty vvdPercMaxData vector is passed as input. Note that separate vvdPercMaxData vectors will need to be maintained when data from multiple confidence level percentages are being processed.

Return value: int – 0 or greater = number of log entries collected, otherwise error

int getConfLevelWcLog

(const dvector &vdPercTimes,
const dvector &vdEnergies,
const vdvector &vvdPitchAngles,
const ivector &viScenarios,
vdvector &vvdWorstCaseLog)

Usage: Returns log entries of the information for each of the 'worst case' events (if any) detected within the input confidence level data, following the most recent call to the *findConfLevelWc*() method. Each log entry contains the time, energy, pitch angle (-1 if omni-directional), associated scenario number and data value of the 'worst case' event detected. The fifth value of the log entry is set to zero, as a placeholder for the time associated with the specific scenario's worst case flux; this time value cannot be determined within this model API, as it is only available from the individual scenarios' data, far outside the scope of these methods.

Parameters:

vdPercTimes – vector of times, in Modified Julian Date form

vdEnergies – vector of the energy values associated with the data values.

vvdPitchAngles – 2-dimensional vector of the pitch angles associated with the data values. When the same set of pitch angles are used for all times, this parameter may be a single vector entry.

viScenarios – vector of the scenario identification numbers of the data loaded into the aggregration.

vvvdWorstCaseLog – returned 2-dimensional vector of the corresponding 'worst case' entries, where each vector entry contains the time, energy value, pitch angle, scenario id, 'worst case' data value, and zeroed-out scenario time.

Return value: int – 0 or greater = number of log entries returned, 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
Return values: -none-

~AdiabatModel

Usage: Destructor
Return values: -none-

Model Parameter Inputs:

int setModelDBDir

(const string& strDataDir)
Usage: Specifies the directory that contains the collection IRENE model database files. The various database files required are automatically selected according to the model and parameters specified. The use of this method is highly recommended, as it *eliminates* the need for the other methods that specify the individual database files; those are only needed for using alternate or non-standard versions.
Parameters:
strDataDir – directory path for the IRENE database files.
Return value: int – 0 = success, otherwise error

int setKPhiDBFile

(const string& strKPhiDBFile)
Usage: Specifies the name of the file (including path) for the K/Phi database. The use of this method is *not needed* when *setModelDBDir*() is specified, except when an alternate or non-standard database file is to be used (this overrides any automatic file specification).
This database name is in the form of '<path>/fastPhi_net.mat'.
Parameters:
strKPhiDBFile –database filename, including path
Return value: int – 0 = success, otherwise error

int setKHMinDBFile

(const string& strKHMinDBFile)

Usage: Specifies the name of the file (including path) for the K/Hmin database. The use of this method is *not needed* when *setModelDBDir*() is specified, except when an alternate or non-standard database file is to be used (this overrides any automatic file specification).

This database name is in the form of '<path>/fast_hmin_net.mat'.

Parameters:

strKHMinDBFile –database filename, including path

Return value: int – 0 = success, otherwise error

int setMagfieldDBFile

(const string& strMagfieldDBFile)

Usage: Specifies the name of the file (including path) for the magnetic field model database. The use of this method is *not needed* when *setModelDBDir*() is specified, except when an alternate or non-standard database file is to be used (this overrides any automatic file specification).

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

---

string getModelDBDir

Usage: Returns the directory name containing the collection of IRENE model database files that was specified in a previous call to the *setModelDBDir*() method; otherwise, blank.

Return value: string – model database directory.

string getKPhiDBFile

Usage: Returns the name of the file for the K/Phi database. This will be available immediately, when specified using the *setKPhiDBFile*() method. When the *setModelDBDir*() method is used, the automatically determined filename will be available after a call to one of the *computeCoordinate**() methods.

Return value: string – K/Phi database filename.

string getKHMinDBFile

Usage: Returns the name of the file for the K/Hmin database. This will be available immediately, when specified using the *setKHMinlDBFile*() method. When the *setModelDBDir*() method is used, the automatically determined filename will be available after a call to the *computeCoordinate**() methods.

Return value: string – K/Hmin database filename.

string getMagfieldDBFile

Usage: Returns the name of the file for the magnetic field model database. This will be available immediately, when specified using the *setMagfieldDBFile*() method. When the *setModelDBDir*() method is used, the automatically determined filename will be available after a call to the *computeCoordinate**() or *convertCoordinates*() methods..

Return value: string – magnetic field model database filename.

Adiabatic invariant limit defaults: K = 0.0 – 25.0; Hmin = -1500.0 – 50000.0 [km]; Phi = 0.125 – 2.5
Any K, Hmin or Phi values calculated to be outside of their respective limits are set to -1.0x10^{-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 = -1500.0)
Usage: Specifies the minimum 'Hmin' value, altitude in km, returned from *computeCoordinateSet*() methods.
Parameters:
dHminMin – 'Hmin' adiabatic value minimum (-1500.0 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 = 0.125)
Usage: Specifies the minimum 'Phi' value returned from *computeCoordinateSet*() methods.
Parameters:
dPhiMin – 'Phi' adiabatic value minimum (0.125 if not specified)
Return value: -none-

void setPhiMax

(const double& dPhiMax = 2.5)
Usage: Specifies the maximum 'Phi' value returned from *computeCoordinateSet*() methods.
Parameters:
dPhiMin – 'Phi' adiabatic value maximum (2.5 if not specified)
Return value: -none-

int updateLimits

Usage: Implements any changes to the K, Hmin or Phi limit specifications, but requires that the database files have already been specified via setKPhiDBFile(), setKHMinDBFile() and setMagfieldDBFile() methods. Use of this method is needed only if any of these limits are changed *after* the initial call to one of the *computeCoordinateSet*() methods.
Return value: int – 0 = success, otherwise error

---

double getK_Min

Usage: Returns the minimum 'K' value returned from *computeCoordinateSet*() methods.
Return value: double – 'K' adiabatic value minimum, as specified in the *setK_Min*() method.

double getK_Max

Usage: Returns the maximum 'K' value returned from *computeCoordinateSet*() methods.
Return value: double – 'K' adiabatic value maximum, as specified in the *setK_Max*() method.

double getHminMin

Usage: Returns the minimum 'Hmin' value, altitude in km, returned from *computeCoordinateSet*() methods.
Return value: double – 'Hmin' adiabatic value minimum, as specified in the *setHminMin*() method.

double getHminMax

Usage: Returns the maximum 'Hmin' value, altitude in km, returned from *computeCoordinateSet*() methods.
Return value: double – 'Hmin' adiabatic value maximum, as specified in the *setHminMax*() method.

double getPhiMin

Usage: Returns the minimum 'Phi' value returned from *computeCoordinateSet*() methods.
Return value: double – 'Phi' adiabatic value minimum, as specified in the *setPhiMin*() method.

double getPhiMax

Usage: Returns the maximum 'Phi' value returned from *computeCoordinateSet*() methods.
Return value: double – 'Phi' adiabatic value maximum, as specified in the *setPhiMax*() method.

Model Execution and Results:

int computeCoordinateSet

```
( const string& strCoordSys,
  const 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,
  vdvector& vvdB
  vdvector& vvdI )
```

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 locations. [time].

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

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

vvdB – returned 2-dimensional vector of the local magnetic field vector components at the ephemeris locations. [time,components]

vvdI – returned 2-dimensional vector of the local magnetic field current "I" value associated with the pitch angles at the ephemeris locations. [time,direction]

Return value: int – 0 = success, otherwise error

int computeCoordinateSet

```
( const string& strCoordSys,
  const 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,
  vdvector& vvdB
  vdvector& vvdI )
```

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]
vvdB – returned 2-dimensional vector of the local magnetic field vector components at the ephemeris locations. [time,components]
vvdI – returned 2-dimensional vector of the local magnetic field current "I" value associated with the pitch angles at the ephemeris locations. [time,direction]
Return value: int – 0 = success, otherwise error

int calcDirPitchAngles

```
( const string& strCoordSys,
  const 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 – <u>Cartesian</u> 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 string& strCoordSys,
  const string& strCoordUnits,
  const dvector& vdTimes,
  const dvector& vdCoord1,
  const dvector& vdCoord2,
  const dvector& vdCoord3,
  const string& strNewCoordSys,
  const 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 string& strCoordSys,
          const string& strCoordUnits,
          const double& dTime,
          const double& dCoord1,
          const double& dCoord2,
          const double& dCoord3,
          const string& strNewCoordSys,
          const 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
Return values: -none-

~RadEnvModel

Usage: Destructor
Return values: -none-

Model Parameter Inputs:

int setModel

(const 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 setModelDBDir

(const string& strDataDir)
Usage: Specifies the directory that contains the collection IRENE model database files. The various database files required are automatically selected according to the model and parameters specified. The use of this method is highly recommended, as it *eliminates* the need for the other methods that specify the individual database files; those are only needed for using alternate or non-standard versions.
Parameters:
strDataDir – directory path for the IRENE database files.
Return value: int – 0 = success, otherwise error

int setModelDBFile

(const string& strModelDBFile)
Usage: Specifies the name of the database file (including path) for legacy flux model calculations. The use of this method is *not needed* when *setModelDBDir*() is specified, except when an alternate or non-standard database file is to be used (this overrides any automatic file specification).
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 string& strMagfieldDBFile)

Usage: Specifies the name of the file (including path) for the magnetic field model database. The use of this method is *not needed* when *setModelDBDir*() is specified, except when an alternate or non-standard database file is to be used (this overrides any automatic file specification).

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

string getModelDBDir

Usage: Returns the directory name containing the collection of IRENE model database files that was specified in a previous call to the *setModelDBDir*() method; otherwise, blank.

Return value: string – model database directory.

string getModelDBFile

Usage: Returns the name of the database file for flux model calculations. This will be available immediately, when specified using the *setModelDBFile*() method. When the *setModelDBDir*() method is used, the automatically determined filename will be available after a call to the *computeFlux*() method.

Return value: string – model database filename, as specified in the *setModelDBFile*() method.

string getMagfieldDBFile

Usage: Returns the name of the file for the magnetic field model database. This will be available immediately, when specified using the *setModelDBFile*() method. When the *setModelDBDir*() method is used, the automatically determined filename will be available after a call to the *computeFlux*() method.

Return value: string – magnetic field model database filename.

---

int setFluxType

(const 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 [MeV] at which the flux values are calculated by the selected model. Please consult Appendix A of the User's Guide for the valid ranges/values of the model selected.

Parameters:

vdEnergies – vector of energy values, in units of MeV

Return value: int – 0 = success, otherwise error

int setActivityLevel

(const 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 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 getFluxType

Usage: Returns the type of flux values to be calculated by the model, as specified in the *setFluxType*() method.
Return value: string – flux type identifier.

int getNumEnergies

Usage: Returns the number of energies, as specified in the *setEnergies*() method.
Return value: int – number of energies.

int getEnergies

(dvector& vdEnergies)
Usage: Returns the set energies at which the flux values are calculated, as specified in the *setEnergies*() method.
Parameters:
vdEnergies – returned vector of energy values, in MeV.
Return value: int – 0 = success, otherwise error

string getActivityLevel

Usage: Returns the geomagnetic activity level parameter for the CRRESPRO, AE8 or AP8 Legacy model, as specified in the *setActivityLevel*() method.
Return value: string – geomagnetic activity level specification.

string getActivityRange

Usage: Returns the geomagnetic activity level parameter for the CRRESELE Legacy model, as specified in the *setActivityRange*() method..
Return value: string – geomagnetic activity level specification

double get15DayAvgAp

Usage: Returns the 15-day average Ap value for the CRRESELE Legacy model, as specified in the *set15DayAvgAp*() method.
Return value: double – 15-day average Ap value.

bool getFixedEpoch

Usage: Returns the current setting for the use of the model-specific fixed epoch, as specified in the *setFixedEpoch*() method.
Return value: bool – True or False.

bool getShiftSAA

Usage: Returns the current setting of shifting the SAA from its fixed-epoch location, as specified in the *setShiftSAA*() method.
Return value: bool – True or False.

---

int setCoordSys

(const string& strCoordSys,
const 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

---

string getCoordSys

Usage: Returns the coordinate system name, as specified in the *setCoordSys*() method.
Return value: string – coordinate system name.

string getCoordSysUnits

Usage: Returns the coordinate system units, as specified in the *setCoordSys*() method.
Return value: string – coordinate system units ('Re' or 'km').

int getNumEphemeris

Usage: Returns the number of currently defined ephemeris, specified in the *setEphemeris*) method.
Return value: int – number of ephemeris.

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.
The returned flux values are in units of [#/cm2/sec] (integral) or [#/cm2/sec/MeV] (differential).
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 always produce flux values for twelve pre-defined energy bins: 1.0-1.3, 1.8-2.4, 3.2-4.2, 5.6-7.4, 9.9-13.2, 17.5-23.3, 30.9-41.1, 54.7-72.8, 80.3-89.7, 100.1-111.7, 124.7-139.1, 155.3-193.4 **keV**). *The returned flux results cannot be used for dose calculations.*

General:

CammiceModel

Usage: Default constructor
Return values: -none-

~CammiceModel

Usage: Destructor
Return values: -none-

Model Parameter Inputs:

int setModelDBDir

(const string& strDataDir)
Usage: Specifies the directory that contains the collection IRENE model database files. The various database files required are automatically selected according to the model and parameters specified. <u>The use of this method is highly recommended</u>, as it *eliminates* the need for the other methods that specify the individual database files; those are only needed for using alternate or non-standard versions.
Parameters:
strDataDir – directory path for the IRENE database files.
Return value: int – 0 = success, otherwise error

int setModelDBFile

(const string& strModelDBFile)
Usage: Specifies the name of the database file (including path) for legacy flux model calculations. The use of this method is *not needed* when *setModelDBDir*() is specified, except when an alternate or non-standard database file is to be used (this overrides any automatic file specification).
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 string& strMagfieldDBFile)
Usage: Specifies the name of the file (including path) for the magnetic field model database. The use of this method is *not needed* when *setModelDBDir*() is specified, except when an alternate or non-standard database file is to be used (this overrides any automatic file specification).
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 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 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 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’, ’90-100’,‘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 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

---

string getModelDBDir

Usage: Returns the directory name containing the collection of IRENE model database files that was specified in a previous call to the *setModelDBDir*() method; otherwise, blank.
Return value: string – model database directory.

string getModelDBFile

Usage: Returns the name of the database file for flux model calculations. This will be available immediately, when specified using the *setModelDBFile*() method. When the *setModelDBDir*() method is used, the automatically determined filename will be available after a call to the *computeFlux*() method.
Return value: string – model database filename, as specified in the *setModelDBFile*() method.

string getMagfieldDBFile

Usage: Returns the name of the file for the magnetic field model database. This will be available immediately, when specified using the *setMagfieldDBFile*() method. When the *setModelDBDir*() method is used, the automatically determined filename will be available after a call to the *computeFlux*() method.
Return value: string – magnetic field model database filename.

string getMagfieldModel

Usage: Returns the magnetic field option for the CAMMICE model run, as specified in the *setMagfieldModel*() method.
Return value: string – magnetic field model specification: 'igrf' or 'igrfop'.

string getDataFilter

Usage: Returns the data filter option for the CAMMICE model, as specified in the *setDataFilter*() method.
Return value: string – data filter specification string: 'all' or 'filtered'.

string getPitchAngleBin

Usage: Returns the pitch angle bin for the CAMMICE model, as specified in the *setPitchAngleBin*() method.
Return value: string – pitch angle bin identification string.

string getSpecies

Usage: Returns the particle species for the CAMMICE model, as specified in the *setSpecies*() method.
Return value: string – species identification.

---

int setCoordSys

(const string& strCoordSys,
const 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

---

string getCoordSys

Usage: Returns the coordinate system name, as specified in the *setCoordSys*() method.
Return value: string – coordinate system name.

string getCoordSysUnits

Usage: Returns the coordinate system units, as specified in the *setCoordSys*() method.
Return value: string – coordinate system units ('Re' or 'km').

int getNumEphemeris

Usage: Returns the number of currently defined ephemeris, specified in the *setEphemeris*) method.
Return value: int – number of ephemeris.

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, for the 12 fixed energy bins.
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
Return values: -none-

~DateTimeUtil

Usage: Destructor
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