AstroMath Reference

Class: **AstroMath.Celestial**

NameSpace: **AstroMath**

Celestial Fields

**FIRST\_GREGORIAN\_YEAR** => <double> 1583.0;

**JULIAN\_BIAS**  => <double> 2200000.0;

**SIDEREAL\_A**  => <double> 0.0657098;

**PI**  => <double> Math.PI;

**TWOPI** => <double> Math.PI \* 2.0;

**EPOCH2000** => <double> 2451545.0;

**EPOCHMJD** => <double> 2400000.5;

**COSEPS** => <double> 0.91748;

**SINEPS** => <double> 0.39778;

**ARC**  => <double> 206264.8062;

Celestial Structures

**Celestial.LatLon** Struct: Represents a terrestrial location in radians of latitude, longitude

Constructors:

**LatLon()** == LatLon(0,0)

**LatLon** (<double> **latitude,** <double> **longitude)**

Properties:

<double> **Lat** => latitude in radians [-pi to +pi]

<double> **Lon** => longitude in radians [0 to +2\*pi]

Methods:

<string> **GetLatitudeString()** => in format “XX.XXX N” or “XX.XXX S”

<string> **GetLongitudeString()** => in format “XX.XXX E” or “XX.XXX W”

**Celestial.RADec** Struct: Represents a sky location in radians of equatorial coordinates

RADec Constructors:

**RADec()**  == RADec(0,0)

**Celestial.RADec (**<double> **RA,** <double> **Dec)**

RADec Properties:

<double> **RA** => Right Ascension in radians [0,+2\*pi]

<double> **Dec** => Declination in radians [-pi,+pi ]

RADec Methods:

<AltAz> **MakeAltAz (**<double> **haR,** <LatLon> **loc)**

Creates a new AltAz instance from an RADec instance, i.e Converts from equatorial to horizontal coordinates

<double> **Altitude (**<double> **haR,** <LatLon> **location)**

Computes the altitude (in radians) of the object at RADec for the given latitude and hour angle.

<double> **Azimuth (**<double> **haR,** <LatLon> **loc)**

Computes Azimuth (radians) of the RADec object for the given latitude and hour angle

<double> **HourAngle (**<DateTime> **utcdate,** <LatLon> **location)**

Calculate the hour angle(radians) for the current time & location for RADec object

<double> **HourAngle (<double> altitude,** <LatLon> **location)**

Calculate the hour angle(radians) for the given altitude,at location for RADec object..

<double> **TransitTime (**<DateTime> **UTCDate,** <LatLon> **location)**

Calculate the transit time for current RADEC at terrestrial location in UTC for RADec object

**Celestial.AltAz** Struct: Represents a sky location in horizon coordinates:,altitude/azimuth

Constructors:

**AltAz ()**

**AltAz (** <double> **alt**, <double> **az)**

Properties:

<double> **Alt** => Altitude in radians [0 to pi]

<double> **Azm** => Azimuth in radians [0 to 2pi]

Methods:

<RADec> **MakeRaDec**(<double> **haR**, <LatLon> **loc**)

Converts hour angle (radians) at location (Latitude/Longitude) to equatorial (RA/Dec)

<double> **RightAscension**(<double> **ha**, <LatLon> **loc**)

Converts hour angle (hours) at location (Latitude/Longitude) to Right Ascension (radians)

<double> **Declination**(<double> **ha**, <LatLon> **loc**)

Converts hour angle (hours) at location (Latitude/Longitude) to Declination (radians)

Celestial Methods

<double> **DateToJulian** (<DateTime> **aDate**)

Converts civil date/time to the Julian Days

<double> **DateToMJD** (<DateTime> **aDate)**

Converts civil date/time to Modified Julian Date

<double> **DateToJ2kD** (<DateTime> **aDate**)

Converts civil current date/time to J2000

<double> **DateToJ2kC** (<DateTime> **aDate**)

Converts civil date/time to J2000 centuries

<DateTime> **JulianToDate (**<double> **jd**)

Converts Julian Days To a civil date

<double> **JulianToJ2kD (**<double> **jd**)

Converts Julian Days To J2000 days

<double> **JulianToJ2KC (**<double> **jd)**

Converts Julian Days To J2000 centuries

<double> **JulianToMJD (**<double> **jd**)

Converts Modified Julian days to Julian days

<DateTime> **J2kDToDate (**<double> j2k)

Converts J2000 Days To a civil date

<double> **J2kDToJulian (**<double> **j2k**)

Converts J2000 days To Julian Days

<DateTime> **J2kCToDate (**<double> **j2kc**)

Converts J2000 centuries to date/time

<double> **J2kCToJulian (**<double> **j2kc**)

Converts J2000 Centuries to Julian Days

<DateTime> **MJDToDate (**<double> **mjd)**

Converts Modified Julian Days To a civil date

<double> **MJDToJulian (**<double> **mjd)**

Converts Modified Julian Days to Julian days

<double> **MJDToJ2kD (**<double> **mjd)**

Converts Modified Julian Days to Julian 2000 days

<double> **J2kDToLMST (**<double> **j2k,** <double> **longR)**

Converts J2000 Days at longitude (radians) to Local Mean Sidereal Time in hours

<double> **MJDToLMST (**<double> **mjd,** <double> **longR)**

Converts Modified Julian Days at longitude (radians) to Local Mean Sidereal Time (hours)

<double> **DateUTCToGST (**<DateTime> **userdate)**

Converts Universal Time (date/time) to Greenich Sidereal Time (hours)

<double> **LSTToLocalTime (**<double> **lst,** <double> **longitudeD)**

Converts Local Sidereal time (hours) to date/time at Longitude (degrees). Not corrected for DST.

<double> **GSTToLST (**<double> **gst,** <double> **longitude)**

Converts Local Sideral Time (hours) from Greenich Sidereal Time (hours) at a longitude (radians)

<double> **LSTToGST (**<double> **lst**, <LatLon> **location**)

Converts Local Sideral Time (hours) from Greenich Sidereal Time in hours at a longitude (radians)

<DateTime> **DayPlusHours (**<DateTime> **aDate,** <double> **someHours)**

Creates date/time composed of a day and hours in a day

<bool> **TimeInBetween (**<DateTime> **earliestTime,** <DateTime> **latestTime,** <DateTime> **thisTime)**

Determines if *thisTime* is later than the *earliestTime* but sooner than the *latestTime*, ignoring the dates

<double> **IntervalOverlap (**<DateTime> **iDusk,** <DateTime> **iDawn,** <DateTime> **iRise,** <DateTime> **iSet)**

Calculates the hours between rise and set that overlap the hours between dusk and dawn on same day

<int> **TimeMachine (**<DateTime> **up,** <DateTime> **down**,

<DateTime> **rise1,** <DateTime> **set1,**

<DateTime> **rise2**, <DateTime> **set2,**

<DateTime> **rise3**, <DateTime> **set3)**

Note that, as DateTime intervals, up < down, rise1 < set 1, rise2 < set2, and rise3 < set3.

This is a complicated solution to a seemingly simple problem of determining the overlap of a single time interval with respect to three other sequential time intervals. In this case, the “up/down” time span represents the rise and set of a celestial target. The other three rise/set’s are for sun (or moon) rise and set’s for three days before and after the “up/down” time. This produces 29 possible outcomes for the up and down occurrences relative to the three rise and set occurrences (0 is no solution):

* Up before rise1
  + and down before rise1 => 1
  + and down between rise1 and set1 => 2
  + and down between set1 and rise2 => 3
  + and down between rise2 and set2 => 4
  + and down between set2 and rise3 => 5
  + and down between rise3 and set3 => 6
  + and down after set3 => 7
* Up between rise1 and set1
  + and down between rise1 and set1 => 8
  + and down between set1 and rise2 => 9
  + and down between rise2 and set2 => 10
  + and down between set2 and rise3 => 11
  + and down between rise3 and set3 => 12
  + and down after set3 => 13
* Up between set1 and rise2
  + and down between set1 and rise2 => 14
  + and down between rise2 and set2 => 15
  + and down between set2 and rise3 => 16
  + and down between rise3 and set3 => 17
  + and down after set3 => 18
* Up between rise2 and set2
  + and down between rise2 and set2 => 19
  + and down between set2 and rise3 => 20
  + and down between rise3 and set3 => 21
  + and down after set3 => 22
* Up between set2 and rise3
  + and down between set2 and rise3 => 23
  + and down between rise3 and set3 => 24
  + and down after set3 => 25
* Up between rise3 and set3
  + and down between rise3 and set3 => 26
  + and down after set3 => 27
* Up after set3
  + and down after set3 => 28

<int> **LongestPeriod** (<TimeSpan> **a**, <TimeSpan> **b**, <TimeSpan> **c**)

Finds the parameter, 1, 2, or 3 that has the longest timespan

<TimeSpan> **LongestInterval (** <TimeSpan> **i1**, <TimeSpan> **i2**)

Finds the longest of two timespans and returns value

<double> **HourAngleToRA (**<double> **ha**, <DateTime> **ut**, <double> **longitude)**

Calculates Right Ascension (radians) for given hour angle (hours) from UT (date/time) at longitude (radians)

<RADec> **PostionFromBearingAndRange (**<RADec> **initialPosition**, <double> **bearing**, <double> **range)**

Calculates a new RA/Decimal position that is a specific arc distance (radians) and bearing (radians) from the initial position. The bearing is an astronomical position angle, CCW from North.

AstroMath.Planar Class

Class for handling quadratic and matrix equations used in astronomical calculations

**Planar.QuadRoot** Struct: Results array for quadratic solution

Constructors

**QuadRoot()** => all values set to zero

Properties

<int> **nz**

Number of roots within the interval [-1,1]

<double> **xe**

Extreme value of X in parabola solution

<double> **ye**

Extreme value of Y in parabola solution

<double> **zero1**

First root within [-1,1] for NZ = 1,2

<double> **zero2**

Second root within [-1,1] for NZ = 2

Methods

<QuadRoot> **Quad**(<double> **yminus**, <double> **yzero**, <double> **yplus**)

Finds a parabola through three points (-1,yminus), (0,yzero), (1,yplus) that do not lie on a straight line.

<double> **Frac** ( <double> **x**)

Calculates the positive fractional component of a number

<Point> **ThirdPoint(** <Point> **C**, <double> **circleradius**, <double> **Alpha**, <double> **ht**)

Calculates the coordinations (point) for the third point of a isocolese triangle with a height of ht and rotated to an angle (radians)

<double> **DotProduct**(<double[]> **a**, <double[]> **b**)

Computes the dot product of two vectors, a and b, of arbitrary length.

AstroMath.SkyView Class

Class of objects and methods for drawing celestial maps. All these methods create graphics on in cartesian coordinates, then map the points on to a spherical projection

Methods

**SkyView**(<Graphics> **fcntl**, <Point> **centerPoint**, <float> **radius**, <float> **observersLatitude**)

Creates a dark blue circle that is used as a background of a celestial projection. Upper left corner and size for a rectangle that defines a circle centered on centerpoint with a radius of radius. Points for north pole, south pole, east equatorial and west equatorial are calculated for subsequent mappings.

<Point[]> **HourLine**( <double> **hourAngleH**)

Creates a set of points that define a line of longitude over the skyviewmap

<Point[]> **DecLine**(<double> **decAngleD**)

Creates a set of points that define a line of longitude over the skyviewmap

<Point[]> **TrackLine**(<double> **startHourAngleH**, <double> **endHourAngleH**, <double> **declinationD**)

Produces a line arc of integer X,Y points that are equidistant from the center

<Point> **LocationOffset**(<Point> **Center**, <double> **Diameter**)

Determines the upper left corner offset of a circle drawing from its center, based on angle

Class Spherical

Class of objects and methods for drawing celestial maps in spherical coordinates

**Spherical.Polar3** Struct**:**

Constructors:

**Polar3()**

**Polar3**(<float> **rhoVal**, <float> **thetaVal**, <float> **phiVal**)

**Polar3**(<Point3> **cpt**)

Converts cartesian to spherical coordinates.

Properties

<float> **Rho**

spherical radius

<float> **Theta**

spherical latitude

<float> **Phi**

spherical longitude

Methods

<Polar3> **RotateX**(<float> **rotationR**)

Rotation of cartesian coordinates around X axis

**Spherical.Point3** Struct

Constructors

**Point3**(<float> **ex**, <float> **ey**, <float> **ez**)

ex => X coordinate

ey => Y coordinate

ez => Z coordinate

**Point3()** => Point3(0,0,0)

**Point3(**<Polar3> **sph**)

Converts spherical to cartesian coordinates

Properties

<float> **X**

<float> **Y**

<float`> **Z**

Methods

<Point3> **RotateX**(<float> **thetaRot**)

Rotate X,Y,Z coordinates around origin along spherical angle theta

<Point[]> **ProjectXY**(<Polar3[]> **spts**)

Projects set of spherical coordinates into an XY plane, where rho is the radius or 1/2 X and 1/2 Y dimension, theta is the Altitude, phi is the Azimuth

Class Transform

Methods

<double> **SIND**(<double> **degrees**)

Computes Sine in degrees

<double> **COSD**(<double> **degrees**)

Computes Cosine in degrees

<double> **RadiansToDegrees**(<double> **rad**)

Converts radians to degrees

<double> **DegreesToRadians**(<double> **deg**)

Converts degrees to radians (-2pi,+2pi)

<double> **HoursToRadians**(<double> **hours**)

Converts hours (0-24) to radians at 2Pi per 24 hours (15 degrees per hour) (-2pi, +2pi)

<double> **RadiansToHours**(<double> **radians**)

Converts radians at 2pi per 24 hours (15 degrees per hour) to hours (-24,+24)

<double> **HoursToDegrees**(<double> **hours**)

Converts hours(0 - 24 hour clock) to degrees (-360,+360)

<double> **DegreesToHours**(<double> **degrees**)

Converts degrees(0 - 360.0) To hours (-24,+24)

<double> **HourAngleToPolarAngle**(<double> **haH**)

Converts HourAngle (0 hour at 6 oclock) in hours to Polar Coordinate (0 degrees at 3 oclock) in radians (-2pi,+2pi)

<double> **NormalizeDegreeRange**(<double> **angleD**)

Converts angle in degrees to 0-360 range

<double> **NormalizeRadianRange**(<double> **angleR**)

Converts angle in radians to 0-2Pi range

<double> **NormalizeHours**(<TimeSpan> **hours**)

Converts time span hours to range 0-24

<double> **NormalizeHours**(<double> **hours**)

Converts hours to range 0-24