AstroMath Reference

**Class Celestial**

Mathematical Constants

<double> FIRST\_GREGORIAN\_YEAR = 1583.0;

<double> JULIAN\_BIAS = 2200000.0;

<double> SIDEREAL\_A = 0.0657098;

<double> PI = Math.PI;

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

<double> EPOCH2000 = 2451545.0;

<double> EPOCHMJD = 2400000.5;

<double> COSEPS = 0.91748;

<double> SINEPS = 0.39778;

<double> ARC = 206264.8062;

Date and Time Methods

*<double>* DateToJulian(*<DateTime>* datein) -> Julian days

Converts civil date/time to the Julian Days

*<double>* DateToMJD(*<DateTime>* thisdate) -> Modified Julian days

Converts civil date/time to Modified Julian Date

*<double>* DateToJ2kD(*<DateTime>* thisdate) -> J2000 days

Converts civil current date/time to J2000

*<double>* DateToJ2kC(*<DateTime>* thisdate) > J2000 in centuries

Converts civil date/time to J2000 centuries

*<DateTime>* JulianToDate(*<double>* jd) -> Civil date/time

Converts Julian Days To a civil date

*<double>* JulianToJ2kD(*<double>* jd) -> Julian days since 2000

Converts Julian Days To J2000 days

*<double>* JulianToJ2KC(*<double>* jd) -> Julian centuries since 2000

Converts Julian Days To J2000 centuries

*<double>* JulianToMJD(*<double>* jd) -> Modified Julian days

Converts Modified Julian days to Julian days

*<DateTime>* J2kDToDate(*<double>* j2k) -> Civil date/time

Converts J2000 Days To a civil date

*<double>* J2kDToJulian(*<double>* j2k) -> Julian days

Converts J2000 days To Julian Days

*<DateTime>* J2kCToDate(*<double>* j2kc) -> Civil date/time

Converts J2000 centuries to date/time

*<double>* J2kCToJulian(*<double>* j2kc) -> Julian days

Converts J2000 centuries To Julian days

*<DateTime>* MJDToDate(*<double>* mjd) -> Civil date/time

Converts Modified Julian Days To a civil date

*<double>* MJDToJulian(*<double>* mjd) -> Julian days

Converts Modified Julian Days to Julian days

*<double>* MJDToJ2kD(*<double>* mjd) -> J2000 days

Converts Modified Julian Days to Julian 2000 days

*<double>* J2kDToLMST(*<double>* j2k, *<double>* longR) -> LMST hours

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

*<double>* MJDtoLMST(*<double>* mjd, *<double>* longR) -> LMST hours

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

*<double>* DateUTCToGST(*<DateTime>* userdate) -> Greenich Sidereal Time

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

*<double>* LSTToLocalTime(*<double>* lst, *<double>* longitudeD) -> Civil date/time

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

*<double>* GSTToLST(*<double>* gst, *<double>* longitude) -> LST hours

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

*<double>* LSTToGST(*<double>* lst, *<LatLon>* location)-> GST hours

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

Time Comparison Methods

*<DateTime>* DayPlusHours(*<DateTime>* thisdate, *<double>* someHours) -> Composite date/time

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

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

Determines if (this time is later than the earliestTime but sooner than the latestTime, ignoring the date

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

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

<int> TimeMachine( *<DateTime>* u, *<DateTime>* d,

*<DateTime>* r1, *<DateTime>* s1,

*<DateTime>* r2, *<DateTime>* s2,

*<DateTime>* r3, *<DateTime>* s3)

Calculates intersection types for target and sun intersections and returns a value for each type of intersection, returns 0 if something doesn’t match up

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

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

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

Finds the longest of two timespans and returns it

Astronomical Coordinate Operations Methods

*<double>* HourAngleToRA(*<double>* ha, *<DateTime>* ut, *<double>* longitude) -> Hour Angle (radians)

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

*<RADec>* PostionFromBearingAndRange(Celestial.RADec initialPosition, *<double>* bearingRadians, *<double>* rangeRadians,)

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.

Celestial: Astronomical Data Classes

LatLon Class

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”

RADec Class

Represents a sky location in radians of equatorial coordinates

Constructors:

RADec()

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

Properties:

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

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

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 (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

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

Calculate the transit time for current RADEC in UTC

AltAz Class

Represents a sky location in horizon coordinates:,altitude/azimuth

Constructors

AltAz()

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

Properties

*<double>* Alt

*<double>* Azm

Methods

*<RADec>* MakeRaDec(*<double>* haR, *<LatLon>* loc) -> Right Ascension/Declination (radians)

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

*<double>* RightAscension(*<double>* ha, *<LatLon>* loc) -> Right Ascension (radians)

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

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

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

**Class Formatters**

Methods

<string> HourString(double dvalue) -> “hours:minutes” (string)

Converts a value (double) to a string looking like an hour:minutes

**Class Planar**

QuadRoot

Structure for returning quadratic root results

<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

Constructors

QuadRoot() -> all values set to zero

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) -> System.Drawing.Point

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.

Class SkyView

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

Constructors

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

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

Point3 Class

Constructors

Point3(float ex, float ey, float ez, float ew)

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) -> Sine as double:

<double> COSD(<double> degrees) -> Cosine as double:

<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