AstroLib.jl Documentation

Release 0.0.4

Mose' Giordano

March 22, 2016

Contents

AstroLib.jl is a package of small generic routines useful above all in astronomical and astrophysical context, written in Julia.

Included are also translations of some IDL Astronomy User's Library procedures, which are released under terms of BSD-2-Clause License. `AstroLib.jl`'s functions are not drop-in replacement of those procedures, Julia standard data types are often used (e.g., `DateTime` type instead of generic string for dates) and the syntax may slightly differ.

An extensive error testing suite ensures old fixed bugs will not be brought back by future changes.

CHAPTER 1

Install

`AstroLib.jl` is available for Julia 0.4 and later versions, and can be installed with Julia built-in package manager. In a Julia session run the command

```
julia> Pkg.add("AstroLib")
```

You may need to update your package list with `Pkg.update()` in order to get the latest version of `AstroLib.jl`.

CHAPTER 2

Usage

After installing the package, you can start using `AstroLib.jl` with

```
using AstroLib
```

CHAPTER 3

Development

`AstroLib.jl` is developed on GitHub at https://github.com/giordano/AstroLib.jl. You can contribute by providing new functions, reporting bugs, and improving documentation.

CHAPTER 4

License

The `AstroLib.jl` package is licensed under the MIT "Expat" License. The original author is Mosè Giordano.

CHAPTER 5

Notes

This project is a work-in-progress, only few procedures have been translated so far. In addition, function syntax may change from time to time. Check TODO.md out to see how you can help. Volunteers are welcome!

CHAPTER 6

Documentation

Every function provided has detailed documentation that can be accessed at Julia REPL with

```
julia> ?FunctionName
```

or with

```
julia> @doc FunctionName
```

The following is the list of all functions provided to the users. Click on them to read their documentation.

6.1 Astronomical Utilities

6.1.1 adstring

adstring(*ra::Real*, *dec::Real*[, *precision::Int=2*, *truncate::Bool=true*]) → string
adstring([*ra*, *dec*]) → string
adstring(*dec*) → string
adstring([*ra*][, *dec*]) → ["string1", "string2", ...]

Purpose

Returns right ascension and declination as string(s) in sexagesimal format.

Explanation

Takes right ascension and declination expressed in decimal format, converts them to sexagesimal and return a formatted string. The precision of right ascension and declination can be specified.

Arguments

Arguments of this function are:

- `ra`: right ascension in decimal degrees. It is converted to hours before printing.
- `dec`: declination in decimal degrees.

The function can be called in different ways:

- Two numeric arguments: first is `ra`, the second is `dec`.
- A 2-tuple `(ra, dec)`.
- One 2-element numeric array: `[ra, dec]`. A single string is returned.
- One numeric argument: it is assumed only `dec` is provided.
- Two numeric arrays of the same length: `ra` and `dec` arrays. An array of strings is returned.
- An array of 2-tuples `(ra, dec)`.

Optional keywords affecting the output format are always available:

- `precision` (optional integer keyword): specifies the number of digits of declination seconds. The number of digits for righ ascension seconds is always assumed to be one more `precision`. If the function is called with only `dec` as input, `precision` default to 1, in any other case defaults to 0.
- `truncate` (optional boolean keyword): if true, then the last displayed digit in the output is truncated in precision rather than rounded. This option is useful if `adstring` is used to form an official IAU name (see http://vizier.u-strasbg.fr/Dic/iau-spec.htx) with coordinate specification.

Output

The function returns one string if the function was called with scalar `ra` and `dec` (or only `dec`) or a 2-element array `[ra, dec]`. If instead it was feeded with arrays of `ra` and `dec`, an array of strings will be returned. The format of strings can be specified with `precision` and `truncate` keywords, see above.

Example

```
julia> adstring(30.4, -1.23, truncate=true)
" 02 01 35.9  -01 13 48"

julia> adstring([30.4, -15.63], [-1.23, 48.41], precision=1)
2-element Array{AbstractString,1}:
 " 02 01 36.00  -01 13 48.0"
 "-22 57 28.80  +48 24 36.0"
```

6.1.2 airtovac

airtovac(*wave_air*) → wave_vacuum

Purpose

Converts air wavelengths to vacuum wavelengths.

Explanation

Wavelengths are corrected for the index of refraction of air under standard conditions. Wavelength values below 2000 will not be altered. Uses relation of Ciddor (1996).

Arguments

- `wave_air`: can be either a scalar or an array of numbers. Wavelengths are corrected for the index of refraction of air under standard conditions. Wavelength values below 2000 will *not* be altered, take care within $[1, 2000]$.

Output

Vacuum wavelength in angstroms, same number of elements as `wave_air`.

Method

Uses relation of Ciddor (1996), Applied Optics 62, 958.

Example

If the air wavelength is `w = 6056.125` (a Krypton line), then `airtovac(w)` yields a vacuum wavelength of `6057.8019`.

Notes

`vactoair` converts vacuum wavelengths to air wavelengths.

Code of this function is based on IDL Astronomy User's Library.

6.1.3 aitoff

`aitoff`$(l, b) \rightarrow$ x, y

Purpose

Convert longitude `l` and latitude `b` to `(x, y)` using an Aitoff projection.

Explanation

This function can be used to create an all-sky map in Galactic coordinates with an equal-area Aitoff projection. Output map coordinates are zero longitude centered.

Arguments

- `l`: longitude, scalar or vector, in degrees.
- `b`: latitude, number of elements as `l`, in degrees.

Coordinates can be given also as a 2-tuple `(l, b)`.

Output

2-tuple `(x, y)`.

- `x`: x coordinate, same number of elements as `l`. `x` is normalized to be in $[-180, 180]$.
- `y`: y coordinate, same number of elements as `l`. `y` is normalized to be in $[-90, 90]$.

Example

Get (x, y) Aitoff coordinates of Sirius, whose Galactic coordinates are $(227.23, -8.890)$.

```
julia> x, y = aitoff(227.23, -8.890)
(-137.92196683723276,-11.772527357473054)
```

Notes

See AIPS memo No. 46 (ftp://ftp.aoc.nrao.edu/pub/software/aips/TEXT/PUBL/AIPSMEMO46.PS), page 4, for details of the algorithm. This version of `aitoff` assumes the projection is centered at b=0 degrees.

Code of this function is based on IDL Astronomy User's Library.

6.1.4 altaz2hadec

`altaz2hadec` (*alt*, *az*, *lat*) $\rightarrow$ ha, dec

Purpose

Convert Horizon (Alt-Az) coordinates to Hour Angle and Declination.

Explanation

Can deal with the NCP singularity. Intended mainly to be used by program `hor2eq`.

Arguments

Input coordinates may be either a scalar or an array, of the same dimension, the output coordinates are always floating points and have the same type (scalar or array) as the input coordinates.

- `alt`: local apparent altitude, in degrees, scalar or array.
- `az`: the local apparent azimuth, in degrees, scalar or vector, measured *east* of *north*!!! If you have measured azimuth west-of-south (like the book Meeus does), convert it to east of north via: `az = (az + 180) % 360`.
- `lat`: the local geodetic latitude, in degrees, scalar or array.

`alt` and `az` can be given as a 2-tuple `(alt, az)`.

Output

2-tuple `(ha, dec)`

- `ha`: the local apparent hour angle, in degrees. The hour angle is the time that right ascension of 0 hours crosses the local meridian. It is unambiguously defined.
- `dec`: the local apparent declination, in degrees.

Example

Arcturus is observed at an apparent altitude of 59d,05m,10s and an azimuth (measured east of north) of 133d,18m,29s while at the latitude of +43.07833 degrees. What are the local hour angle and declination of this object?

```
julia>  ha, dec = altaz2hadec(ten(59,05,10), ten(133,18,29), 43.07833)
(336.6828582472844,19.182450965120402)
```

The widely available XEPHEM code gets:

```
Hour Angle = 336.683
Declination = 19.1824
```

Notes

Code of this function is based on IDL Astronomy User's Library.

6.1.5 calz_unred

`calz_unred`(*wave*, *flux*, *ebv*[, *r_v*]) → deredden_wave

Purpose

Deredden a galaxy spectrum using the Calzetti et al. (2000) recipe.

Explanation

Calzetti et al. (2000, ApJ 533, 682; http://adsabs.harvard.edu/abs/2000ApJ...533..682C) developed a recipe for dereddening the spectra of galaxies where massive stars dominate the radiation output, valid between 0.12 to 2.2 microns. (`calz_unred` extrapolates between 0.12 and 0.0912 microns.)

Arguments

- `wave`: wavelength vector (Angstroms)
- `flux`: calibrated flux vector, same number of elements as `wave`.
- `ebv`: color excess E(B-V), scalar. If a negative `ebv` is supplied, then fluxes will be reddened rather than deredenned. Note that the supplied color excess should be that derived for the stellar continuum, EBV(stars), which is related to the reddening derived from the gas, EBV(gas), via the Balmer decrement by EBV(stars) = 0.44*EBV(gas).

- `r_v` (optional): scalar ratio of total to selective extinction, default is 4.05. Calzetti et al. (2000) estimate $r_v = 4.05 \pm 0.80$ from optical-IR observations of 4 starbursts.

Output

Unreddened flux vector, same units and number of elements as `flux`. Flux values will be left unchanged outside valid domain (0.0912 - 2.2 microns).

Example

Estimate how a flat galaxy spectrum (in wavelength) between 1200 and 3200 is altered by a reddening of E(B-V) = 0.1.

```
julia> wave = reshape(1200:50:3150,40);

julia> flux = ones(wave);

julia> calz_unred(wave, flux, -0.1);
```

Notes

Code of this function is based on IDL Astronomy User's Library.

6.1.6 ct2lst

`ct2lst`(*longitude*, *jd*) → local_sidereal_time
`ct2lst`(*longitude*, *tz*, *date*) → local_sidereal_time

Purpose

Convert from Local Civil Time to Local Mean Sidereal Time.

Arguments

The function can be called in two different ways. The only argument common to both methods is `longitude`:

- `longitude`: the longitude in degrees (east of Greenwich) of the place for which the local sidereal time is desired, scalar. The Greenwich mean sidereal time (GMST) can be found by setting longitude = `0`.

The civil date to be converted to mean sidereal time can be specified either by providing the Julian days:

- `jd` (optional numeric keyword): this is number of Julian days for the date to be converted. It can be a scalar or an array.

or the time zone and the date:

- `tz`: the time zone of the site in hours, positive East of the Greenwich meridian (ahead of GMT). Use this parameter to easily account for Daylight Savings time (e.g. -4=EDT, -5 = EST/CDT), scalar.
- `date`: this is the local civil time with type `DateTime`. It can be a scalar or an array.

Output

The local sidereal time for the date/time specified in hours. This is a scalar or an array of the same length as `jd` or `date`.

Method

The Julian days of the day and time is question is used to determine the number of days to have passed since 2000-01-01. This is used in conjunction with the GST of that date to extrapolate to the current GST; this is then used to get the LST. See Astronomical Algorithms by Jean Meeus, p. 84 (Eq. 11-4) for the constants used.

Example

Find the Greenwich mean sidereal time (GMST) on 2008-07-30 at 15:53 in Baltimore, Maryland (longitude=-76.72 degrees). The timezone is EDT or tz=-4

```
julia> ct2lst(-76.72, -4, DateTime(2008, 7, 30, 15, 53))
11.356505172312609

julia> sixty(ans)
3-element Array{Float64,1}:
 11.0    # Hours
 21.0    # Minutes
 23.4186 # Seconds
```

Find the Greenwich mean sidereal time (GMST) on 2015-11-24 at 13:21 in Heidelberg, Germany (longitude=08° 43' E). The timezone is CET or tz=1. Provide `ct2lst` only with the longitude of the place and the number of Julian days.

```
# Convert longitude to decimals.
julia> longitude=ten(8, 43)
8.716666666666667

# Get number of Julian days.  Remember to subtract the time zone in
# order to convert local time to UTC.
julia> jd = jdcnv(DateTime(2015, 11, 24, 13, 21) - Dates.Hour(1))
2.4573510145833334e6

# Calculate Greenwich Mean Sidereal Time.
julia> ct2lst(longitude, jd)
17.140685171005316

julia> sixty(ans)
3-element Array{Float64,1}:
 17.0    # Hours
  8.0    # Minutes
 26.4666 # Seconds
```

Notes

Code of this function is based on IDL Astronomy User's Library.

6.1.7 daycnv

daycnv (*julian_days*) → DateTime

Purpose

Converts Julian days number to Gregorian calendar dates.

Explanation

Takes the number of Julian calendar days since epoch `-4713-11-24T12:00:00` and returns the corresponding proleptic Gregorian Calendar date.

Argument

- `julian_days`: Julian days number, scalar or array.

Output

Proleptic Gregorian Calendar date, of type `DateTime`, corresponding to the given Julian days number.

Example

```
julia> daycnv(2440000)
1968-05-23T12:00:00
```

Notes

`jdcnv` is the inverse of this function.

6.1.8 flux2mag

flux2mag (*flux*[, *zero_point*, *ABwave=number*]) → magnitude

Purpose

Convert from flux expressed in erg/(s cm² Å) to magnitudes.

Explanation

This is the reverse of `mag2flux`.

Arguments

- `flux`: the flux to be converted in magnitude, expressed in erg/(s cm² Å). It can be either a scalar or an array.
- `zero_point`: scalar giving the zero point level of the magnitude. If not supplied then defaults to 21.1 (Code et al 1976). Ignored if the `ABwave` keyword is supplied
- `ABwave` (optional numeric keyword): wavelength scalar or vector in Angstroms. If supplied, then returns Oke AB magnitudes (Oke & Gunn 1983, ApJ, 266, 713; http://adsabs.harvard.edu/abs/1983ApJ...266..713O).

Output

The magnitude. It is of the same type, scalar or array, as `flux`.

If the `ABwave` keyword is set then magnitude is given by the expression

$$\text{ABmag} = -2.5\log_{10}(f) - 5\log_{10}(\text{ABwave}) - 2.406$$

Otherwise, magnitude is given by the expression

$$\text{mag} = -2.5\log_{10}(\text{flux}) - \text{zero point}$$

Example

```
julia> flux2mag(5.2e-15)
14.609991640913002
```

Notes

Code of this function is based on IDL Astronomy User's Library.

6.1.9 get_date

get_date ([*date::DateTime*]) → string
get_date (*[date::DateTime;] old=true*) → string
get_date (*[date::DateTime;] timetag=true*) → string

Purpose

Returns the UTC date in `"CCYY-MM-DD"` format for FITS headers.

Explanation

This is the format required by the `DATE` and `DATE-OBS` keywords in a FITS header.

Argument

- `date` (optional): the date in UTC standard, of `DateTime` type. If omitted, defaults to the current UTC time.
- `old` (optional boolean keyword): see below.
- `timetag` (optional boolean keyword): see below.

Output

A string with the date formatted according to the given optional keywords.

- When no optional keywords (`timetag` and `old`) are supplied, the format of the output string is `"CCYY-MM-DD"` (year-month-day part of the date), where represents a 4-digit calendar year, the 2-digit ordinal number of a calendar month within the calendar year, and

  the 2-digit ordinal number of a day within the calendar month.
- If the boolean keyword `old` is true (default: false), the year-month-day part of date has `"DD/MM/YY"` format. This is the formerly (pre-1997) recommended for FITS. Note that this format is now deprecated because it uses only a 2-digit representation of the year.
- If the boolean keyword `timetag` is true (default: false), `"Thh:mm:ss"` is appended to the year-month-day part of the date, where represents the hour in the day, the minutes, the seconds, and the literal 'T' the ISO 8601 time designator.

Note that `old` and `timetag` keywords can be used together, so that the output string will have `"DD/MM/YYThh:mm:ss"` format.

Example

```
julia> get_date(timetag=true)
"2016-03-14T11:26:23"
```

Notes

1. A discussion of the DATExxx syntax in FITS headers can be found in http://www.cv.nrao.edu/fits/documents/standards/year2000.txt
2. Those who wish to use need further flexibility in their date formats (e.g. to use TAI time) should look at Bill Thompson's time routines in http://sohowww.nascom.nasa.gov/solarsoft/gen/idl/time

---

6.1.10 get_juldate

`get_juldate``()` → julian_days

Purpose

Return the number of Julian days for current time.

Explanation

Return for current time the number of Julian calendar days since epoch `-4713-11-24T12:00:00` as a floating point.

Example

```
julia> get_juldate()
2.4574620222685183e6

julia> daycnv(get_juldate())
2016-03-14T12:32:13
```

Notes

Use `jdcnv` to get the number of Julian days for a different date.

6.1.11 gcirc

`gcirc`(*units*, *ra1*, *dec1*, *ra2*, *dec2*) → angular_distance

Purpose

Computes rigorous great circle arc distances.

Explanation

Input position can be either radians, sexagesimal right ascension and declination, or degrees.

Arguments

- `units`: integer, can be either 0, or 1, or 2. Describes units of inputs and output:
  - 0: everything (input right ascensions and declinations, and output distance) is radians
  - 1: right ascensions are in decimal hours, declinations in decimal degrees, output distance in arc seconds
  - 2: right ascensions and declinations are in degrees, output distance in arc seconds
- `ra1`: right ascension or longitude of point 1
- `dec1`: declination or latitude of point 1
- `ra2`: right ascension or longitude of point 2
- `dec2`: declination or latitude of point 2

Both `ra1` and `dec1`, and `ra2` and `dec2` can be given as 2-tuples `(ra1, dec1)` and `(ra2, dec2)`.

Output

Angular distance on the sky between points 1 and 2, as a `AbstractFloat`. See `units` argument above for the units.

Method

"Haversine formula" see http://en.wikipedia.org/wiki/Great-circle_distance.

Example

```
julia> gcirc(0, 120, -43, 175, +22)
1.590442261600714
```

Notes

- If `ra1`, `dec1` are scalars, and `ra2`, `dec2` are vectors, then the output is a vector giving the distance of each element of `ra2`, `dec2` to `ra1`, `dec1`. Similarly, if `ra1`,`de1` are vectors, and `ra2`,`dec2` are scalars, then the output is a vector giving the distance of each element of `ra1`, `dec1` to `ra2`, `dec2`. If both `ra1`, `dec1` and `ra2`, `dec2` are vectors then the output is a vector giving the distance of each element of `ra1`, `dec1` to the corresponding element of `ra2`, `dec2`.
- The function `sphdist` provides an alternate method of computing a spherical distance.
- The Haversine formula can give rounding errors for antipodal points.

Code of this function is based on IDL Astronomy User's Library.

6.1.12 jdcnv

`jdcnv` (*date::DateTime*) → julian_days

Purpose

Convert proleptic Gregorian Calendar date in UTC standard to number of Julian days.

Explanation

Takes the given proleptic Gregorian date in UTC standard and returns the number of Julian calendar days since epoch `-4713-11-24T12:00:00`.

Argument

- `date`: date of `DateTime` type, in proleptic Gregorian Calendar.

Output

Number of Julian days, as a floating point.

Example

Find the Julian days number at 2009 August 23, 03:39:06.

```
julia> jdcnv(DateTime(2009, 08, 23, 03, 39, 06))
2.4550666521527776e6
```

Notes

This is the inverse of `daycnv`.

`get_juldate` returns the number of Julian days for current time. It is equivalent to `jdcnv(Dates.now())`.

For the conversion of Julian date to number of Julian days, use `juldate`.

6.1.13 juldate

`juldate`(*date::DateTime*) → reduced_julia_days

Purpose

Convert from calendar to Reduced Julian Days.

Explanation

Julian Day Number is a count of days elapsed since Greenwich mean noon on 1 January 4713 B.C. Julian Days are the number of Julian days followed by the fraction of the day elapsed since the preceding noon.

This function takes the given `date` and returns the number of Julian calendar days since epoch `1858-11-16T12:00:00` (Reduced Julian Days = Julian Days - 2400000).

Argument

- `date`: date of `DateTime` type, in Julian Calendar, UTC standard.

Example

Get number of Reduced Julian Days at 2016-03-20T15:24:00.

```
julia> juldate(DateTime(2016, 03, 20, 15, 24))
57468.14166666667
```

Notes

Julian Calendar is assumed, thus before `1582-10-15T00:00:00` this function is *not* the inverse of `daycnv`. For the conversion proleptic Gregorian date to number of Julian days, use `jdcnv`, which is the inverse of `daycnv`.

Code of this function is based on IDL Astronomy User's Library.

6.1.14 mag2flux

mag2flux (*mag*[, *zero_point*, *ABwave=number*]) → flux

Purpose

Convert from magnitudes to flux expressed in erg/(s cm² Å).

Explanation

This is the reverse of `flux2mag`.

Arguments

- `mag`: the magnitude to be converted in flux. It can be either a scalar or an array.
- `zero_point`: scalar giving the zero point level of the magnitude. If not supplied then defaults to 21.1 (Code et al 1976). Ignored if the `ABwave` keyword is supplied
- `ABwave` (optional numeric keyword): wavelength, scalar or array, in Angstroms. If supplied, then the input `mag` is assumed to contain Oke AB magnitudes (Oke & Gunn 1983, ApJ, 266, 713; http://adsabs.harvard.edu/abs/1983ApJ...266..713O).

Output

The flux. It is of the same type, scalar or array, as `mag`.

If the `ABwave` keyword is set, then the flux is given by the expression

$$\text{flux} = 10^{-0.4(\text{mag}+2.406+4\log_{10}(\text{ABwave}))}$$

Otherwise the flux is given by

$$\text{flux} = 10^{-0.4(\text{mag}+\text{zero point})}$$

Example

```
julia> mag2flux(8.3)
1.7378008287493692e-12
```

Notes

Code of this function is based on IDL Astronomy User's Library.

6.1.15 polrec

polrec (*radius*, *angle*[, *degrees=true*]) → x, y

Purpose

Convert 2D polar coordinates to rectangular coordinates.

Explanation

This is the partial inverse function of `recpol`.

Arguments

- `radius`: radial coordinate of the point. It may be a scalar or an array.
- `angle`: the angular coordinate of the point. It may be a scalar or an array of the same lenth as `radius`.
- `degrees` (optional boolean keyword): if `true`, the `angle` is assumed to be in degrees, otherwise in radians. It defaults to `false`.

Mandatory arguments can also be passed as the 2-tuple `(radius, angle)`, so that it is possible to execute `recpol(polrec(radius, angle))`.

Output

A 2-tuple `(x, y)` with the rectangular coordinate of the input. If `radius` and `angle` are arrays, `x` and `y` are arrays of the same length as `radius` and `angle`.

Example

Get rectangular coordinates (x, y) of the point with polar coordinates $(r,) = (1.7, 227)$, with angle expressed in degrees.

```
julia> x, y = polrec(1.7, 227, degrees=true)
(-1.1593972121062475,-1.2433012927525897)
```

6.1.16 precess

`precess`(*ra*, *dec*, *equinox1*, *equinox2*[, *FK4=true*, *radians=true*]) → prec_ra, prec_dec

Purpose

Precess coordinates from `equinox1` to `equinox2`.

Explanation

The default `(ra, dec)` system is FK5 based on epoch J2000.0 but FK4 based on B1950.0 is available via the `FK4` boolean keyword.

Arguments

- `ra`: input right ascension, scalar or vector, in degrees, unless the `radians` keyword is set to `true`
- `dec`: input declination, scalar or vector, in degrees, unless the `radians` keyword is set to `true`
- `equinox1`: original equinox of coordinates, numeric scalar.
- `equinox2`: equinox of precessed coordinates.
- `FK4` (optional boolean keyword): if this keyword is set to `true`, the FK4 (B1950.0) system precession angles are used to compute the precession matrix. When it is `false`, the default, use FK5 (J2000.0) precession angles.
- `radians` (optional boolean keyword): if this keyword is set to `true`, then the input and output right ascension and declination vectors are in radians rather than degrees.

Output

The 2-tuple `(ra, dec)` of coordinates modified by precession.

Example

The Pole Star has J2000.0 coordinates (2h, 31m, 46.3s, 89d 15' 50.6"); compute its coordinates at J1985.0

```
julia> ra, dec = ten(2,31,46.3)*15, ten(89,15,50.6)
(37.94291666666666,89.26405555555556)

julia> adstring(precess(ra, dec, 2000, 1985), precision=1)
" 02 16 22.73  +89 11 47.3"
```

Precess the B1950 coordinates of Eps Ind (RA = 21h 59m,33.053s, DEC = (-56d, 59', 33.053") to equinox B1975.

```
julia> ra, dec = ten(21, 59, 33.053)*15, ten(-56, 59, 33.053)
(329.88772083333333,-56.992514722222225)

julia> adstring(precess(ra, dec, 1950, 1975, FK4=true), precision=1)
" 22 01 15.46  -56 52 18.7"
```

Method

Algorithm from "Computational Spherical Astronomy" by Taff (1983), p. 24. (FK4). FK5 constants from "Explanatory Supplement To The Astronomical Almanac" 1992, page 104 Table 3.211.1 (https://archive.org/details/131123ExplanatorySupplementAstronomicalAlmanac).

Notes

Accuracy of precession decreases for declination values near 90 degrees. `precess` should not be used more than 2.5 centuries from 2000 on the FK5 system (1950.0 on the FK4 system).

Code of this function is based on IDL Astronomy User's Library.

6.1.17 precess_xyz

`precess_xyz` (*x*, *y*, *z*, *equinox1*, *equinox2*) → prec_x, prec_y, prec_z

Purpose

Precess equatorial geocentric rectangular coordinates.

Arguments

- `x`, `y`, `z`: scalars or vectors giving heliocentric rectangular coordinates.
- `equinox1`: original equinox of coordinates, numeric scalar.
- `equinox2`: equinox of precessed coordinates, numeric scalar.

Input coordinates can be given also a 3-tuple `(x, y, z)`.

Output

The 3-tuple `(x, y, z)` of coordinates modified by precession.

Example

Precess 2000 equinox coordinates `(1, 1, 1)` to 2050.

```
julia> precess_xyz(1, 1, 1, 2000, 2050)
(0.9838854500981734,1.0110925876508692,1.0048189888146941)
```

Method

The equatorial geocentric rectangular coordinates are converted to right ascension and declination, precessed in the normal way, then changed back to `x`, `y` and `z` using unit vectors.

Notes

Code of this function is based on IDL Astronomy User's Library.

6.1.18 premat

premat(*equinox1*, *equinox2*[, *FK4=true*]) → precession_matrix

Purpose

Return the precession matrix needed to go from `equinox1` to `equinox2`.

Explanation

This matrix is used by `precess` and `baryvel` to precess astronomical coordinates.

Arguments

- `equinox1`: original equinox of coordinates, numeric scalar.
- `equinox2`: equinox of precessed coordinates.
- `FK4` (optional boolean keyword): if this keyword is set to `true`, the FK4 (B1950.0) system precession angles are used to compute the precession matrix. When it is `false`, the default, use FK5 (J2000.0) precession angles.

Output

A 3×3 `AbstractFloat` matrix, used to precess equatorial rectangular coordinates.

Example

Return the precession matrix from 1950.0 to 1975.0 in the FK4 system

```
julia> premat(1950,1975,FK4=true)
3x3 Array{Float64,2}:
 0.999981    -0.00558775  -0.00242909
 0.00558775   0.999984    -6.78691e-6
 0.00242909  -6.78633e-6   0.999997
```

Method

FK4 constants from "Computational Spherical Astronomy" by Taff (1983), p. 24. (FK4). FK5 constants from "Explanatory Supplement To The Astronomical Almanac" 1992, page 104 Table 3.211.1 (https://archive.org/details/131123ExplanatorySupplementAstronomicalAlmanac).

Notes

Code of this function is based on IDL Astronomy User's Library.

6.1.19 radec

radec(*ra::Number*, *dec::Number*[, *hours=true*]) → ra_hours, ra_minutes, ra_seconds, dec_degrees, dec_minutes, dec_seconds

Purpose

Convert right ascension and declination from decimal to sexagesimal units.

Explanation

The conversion is to sexagesimal hours for right ascension, and sexagesimal degrees for declination.

Arguments

- `ra`: decimal right ascension, scalar or array. It is expressed in degrees, unless the optional keyword `hours` is set to `true`.
- `dec`: declination in decimal degrees, scalar or array, same number of elements as `ra`.
- `hours` (optional boolean keyword): if `false` (the default), `ra` is assumed to be given in degrees, otherwise `ra` is assumed to be expressed in hours.

Output

A 6-tuple of `AbstractFloat`:

```
(ra_hours, ra_minutes, ra_seconds, dec_degrees, dec_minutes, dec_seconds)
```

If `ra` and `dec` are arrays, also each element of the output 6-tuple are arrays of the same dimension.

Example

Position of Sirius in the sky is (ra, dec) = (6.7525, -16.7161), with right ascension expressed in hours. Its sexagesimal representation is given by

```
julia> radec(6.7525, -16.7161, hours=true)
(6.0,45.0,9.0,-16.0,42.0,57.9600000000064)
```

6.1.20 recpol

`recpol` (x, y[, *degrees*=*true*]) $\rightarrow$ radius, angle

Purpose

Convert 2D rectangular coordinates to polar coordinates.

Explanation

This is the partial inverse function of `polrec`.

Arguments

- `x`: the abscissa coordinate of the point. It may be a scalar or an array.
- `y`: the ordinate coordinate of the point. It may be a scalar or an array of the same lenth as `x`.
- `degrees` (optional boolean keyword): if `true`, the output `angle` is given in degrees, otherwise in radians. It defaults to `false`.

Mandatory arguments may also be passed as the 2-tuple `(x, y)`, so that it is possible to execute `polrec(recpol(x, y))`.

Output

A 2-tuple `(radius, angle)` with the polar coordinates of the input. The coordinate `angle` coordinate lies in the range $[-,]$ if `degrees=false`, or $[-180, 180]$ when `degrees=true`.

If `x` and `y` are arrays, `radius` and `angle` are arrays of the same length as `radius` and `angle`.

Example

Calculate polar coordinates $(r,)$ of point with rectangular coordinates $(x, y) = (2.24, -1.87)$.

```
julia> r, phi = recpol(2.24, -1.87)
(2.9179616172938263,-0.6956158538564537)
```

Angle is given in radians.

6.1.21 rhotheta

`rhotheta` (*period*, *periastron*, *eccentricity*, *semimajor_axis*, *inclination*, *omega*, *omega2*, *epoch*) → rho, theta

Purpose

Calculate the separation and position angle of a binary star.

Explanation

This function will return the separation and position angle of a visual binary star derived from its orbital elements. The algorithms described in the following book will be used: Meeus J., 1992, Astronomische Algorithmen, Barth. Compared to the examples given at page 400 and no discrepancy found.

Arguments

- `period`: period [year]
- `periastro`: time of periastron passage [year]
- `eccentricity`: eccentricity of the orbit
- `semimajor_axis`: semi-major axis [arc second]
- `inclination`: inclination angle [degree]
- `omega`: node [degree]
- `omega2`: longitude of periastron [degree]
- `epoch`: epoch of observation [year]

All input parameters have to be scalars.

Output

The 2-tuple $(,)$, where

- : separation [arc second]
- : position angle [degree]

Example

Find the position of Eta Coronae Borealis at the epoch 2016

```
julia> ρ, θ = rhotheta(41.623, 1934.008, 0.2763, 0.907, 59.025, 23.717, 219.907, 2016)
(0.6351167848228113,214.42513388052114)
```

Notes

Code of this function is based on IDL Astronomy User's Library.

6.1.22 sixty

`sixty`(*number*) $\rightarrow$ [deg, min, sec]

Purpose

Converts a decimal number to sexagesimal.

Explanation

The reverse of `ten` function.

Argument

- `number`: decimal number to be converted to sexagesimal.

Output

An array of three `AbstractFloat`, that are the sexagesimal counterpart (degrees, minutes, seconds) of `number`.

Example

```
julia> sixty(-0.615)
3-element Array{Float64,1}:
 -0.0
 36.0
 54.0
```

Notes

Code of this function is based on IDL Astronomy User's Library.

6.1.23 sphdist

sphdist (*long1*, *lat1*, *long2*, *lat2*[, *degrees=true*]) → angular_distance

Purpose

Angular distance between points on a sphere.

Arguments

- `long1`: longitude of point 1
- `lat1`: latitude of point 1
- `long2`: longitude of point 2
- `lat2`: latitude of point 2
- `degrees` (optional boolean keyword): if `true`, all angles, including the output distance, are assumed to be in degrees, otherwise they are all in radians. It defaults to `false`.

Output

Angular distance on a sphere between points 1 and 2, as an `AbstractFloat`. It is expressed in radians unless `degrees` keyword is set to `true`.

Example

```
julia> sphdist(120, -43, 175, +22)
1.5904422616007134
```

Notes

- `gcirc` function is similar to `sphdist`, but may be more suitable for astronomical applications.
- If `long1`, `lat1` are scalars, and `long2`, `lat2` are vectors, then the output is a vector giving the distance of each element of `long2`, `lat2` to `long1`, `lat1`. Similarly, if `long1`,`de1` are vectors, and `long2`,`lat2` are scalars, then the output is a vector giving the distance of each element of `long1`, `lat1` to `long2`, `lat2`. If both `long1`, `lat1` and `long2`, `lat2` are vectors then the output is a vector giving the distance of each element of `long1`, `lat1` to the corresponding element of `long2`, `lat2`.

Code of this function is based on IDL Astronomy User's Library.

6.1.24 ten

ten (*deg*[, *min*, *sec*]) → decimal
ten (*"deg:min:sec"*) → decimal
tenv ([*deg*][, *min*][, *sec*]) → decimal
tenv ([*"deg:min:sec"*]) → decimal

Purpose

Converts a sexagesimal number or string to decimal.

Explanation

`ten` is the inverse of the `sixty` function. `tenv` is the vectorial version of `ten`.

Arguments

`ten` takes as argument either three scalars (`deg`, `min`, `sec`) or a string. The string should have the form `"deg:min:sec"` or `"deg min sec"`. Also a one dimensional array `[deg, min, sec]` is accepted as argument.

If minutes and seconds are not specified they default to zero.

`tenv` takes as input three numerical arrays of numbers (minutes and seconds arrays default to null arrays if omitted) or one array of strings.

Output

The decimal conversion of the sexagesimal numbers provided is returned. The output has the same dimension as the input.

Method

The formula used for the conversion is

$$\text{sign}(\text{deg})\left(|\text{deg}| + \frac{\text{min}}{60} + \frac{\text{sec}}{3600}\right)$$

Example

```
julia> ten(-0.0, 19, 47)
-0.3297222222222222

julia> ten("+5:14:58")
5.249444444444444

julia> ten("-10 26")
-10.433333333333334
```

Notes

These functions cannot deal with `-0` (negative integer zero) in numeric input. If it is important to give sense to negative zero, you can either make sure to pass a floating point negative zero `-0.0` (this is the best option), or use negative minutes and seconds, or non-integer negative degrees and minutes.

6.1.25 vactoair

`vactoair`(*wave_vacuum*) → wave_air

Purpose

Converts vacuum wavelengths to air wavelengths.

Explanation

Corrects for the index of refraction of air under standard conditions. Wavelength values below 2000 will not be altered. Uses relation of Ciddor (1996).

Arguments

- `wave_vacuum`: vacuum wavelength in angstroms. Can be either a scalar or an array of numbers. Wavelengths are corrected for the index of refraction of air under standard conditions. Wavelength values below 2000 will *not* be altered, take care within $[1, 2000]$.

Output

Air wavelength in angstroms, same number of elements as `wave_vacuum`.

Method

Uses relation of Ciddor (1996), Applied Optics 35, 1566 (http://adsabs.harvard.edu/abs/1996ApOpt..35.1566C).

Example

If the vacuum wavelength is `w = 2000`, then `vactoair(w)` yields an air wavelength of `1999.353`.

Notes

`airtovac` converts air wavelengths to vacuum wavelengths.

Code of this function is based on IDL Astronomy User's Library.

6.1.26 xyz

xyz (*jd*[, *equinox=2000*]) → x, y, z, v_x, v_y, v_z

Purpose

Calculate geocentric x, y, and z and velocity coordinates of the Sun.

Explanation

Calculates geocentric x, y, and z vectors and velocity coordinates (dx, dy and dz) of the Sun. (The positive x axis is directed towards the equinox, the y-axis, towards the point on the equator at right ascension 6h, and the z axis toward the north pole of the equator). Typical position accuracy is $< 10^{-4}$ AU (15000 km).

Arguments

- `jd`: number of Reduced Julian Days for the wanted date. It can be either a scalar or a vector.
- `equinox` (optional numeric keyword): equinox of output. Default is 1950.

You can use `juldate` to get the number of Reduced Julian Days for the selected dates.

Output

The 6-tuple (x, y, z, v_x, v_y, v_z), where

- x, y, z: scalars or vectors giving heliocentric rectangular coordinates (in AU) for each date supplied. Note that $\sqrt{x^2 + y^2 + z^2}$ gives the Earth-Sun distance for the given date.
- v_x, v_y, v_z: velocity vectors corresponding to x, y, and z.

Example

What were the rectangular coordinates and velocities of the Sun on 1999-01-22T00:00:00 (= JD 2451200.5) in J2000 coords? Note: Astronomical Almanac (AA) is in TDT, so add 64 seconds to UT to convert.

```
julia> jd = juldate(DateTime(1999, 1, 22))
51200.5

julia> xyz(jd + 64./86400., equinox=2000)
(0.5145687092402946,-0.7696326261820777,-0.33376880143026394,0.014947267514081075,0.008314838205475
```

Compare to Astronomical Almanac (1999 page C20)

```
             x  (AU)         y  (AU)      z (AU)
xyz:       0.51456871   -0.76963263  -0.33376880
AA:        0.51453130   -0.7697110   -0.3337152
abs(err): 0.00003739    0.00007839   0.00005360
abs(err)
    (km):   5609          11759         8040
```

NOTE: Velocities in AA are for Earth/Moon barycenter (a very minor offset) see AA 1999 page E3

```
           x vel (AU/day) y vel (AU/day)   z vel (AU/day)
xyz:      -0.014947268   -0.0083148382    -0.0036068576
AA:       -0.01494574    -0.00831185      -0.00360365
abs(err):  0.000001583    0.0000029886     0.0000032076
abs(err)
 (km/sec): 0.00265        0.00519          0.00557
```

Notes

Code of this function is based on IDL Astronomy User's Library.

6.1.27 ydn2md

ydn2md (*year*, *day*) → date

Purpose

Convert from year and day number of year to a date.

Explanation

Returns the date corresponding to the `day` of `year`.

Arguments

- `year`: the year, as a scalar integer.
- `day`: the day of `year`, as an integer. It is can be either a scalar or array of integers.

Output

The date, of `Date` type, of day − 1 days after January 1st of `year`.

Example

Find the date of the 60th and 234th days of the year 2016.

```
julia> ydn2md(2016, [60, 234])
2-element Array{Date,1}:
 2016-02-29
 2016-08-21
```

Note

`ymd2dn` converts from a date to day of the year.

6.1.28 ymd2dn

ymd2dn (*date*) → number_of_days

Purpose

Convert from a date to day of the year.

Explanation

Returns the day of the year for `date` with January 1st being day 1.

Arguments

- `date`: the date with `Date` type. Can be a single date or an array of dates.

Output

The day of the year for the given `date`. If `date` is an array, returns an array of days.

Example

Find the days of the year for March 5 in the years 2015 and 2016 (this is a leap year).

```
julia> ymd2dn([Date(2015, 3, 5), Date(2016, 3, 5)])
2-element Array{Int64,1}:
 64
 65
```

Note

`ydn2md` converts from year and day number of year to a date.

6.2 Miscellaneous (Non-Astronomy) Utilities

6.2.1 cirrange

cirrange (*number*[, *max*]) → restricted_number

Purpose

Force a number into a given range $[0, \max)$.

Argument

- `number`: the number to modify. Can be a scalar or an array.
- `max` (optional numerical argument): specify the extremum of the range $[0, \max)$ into which the number should be restricted. If omitted, defaults to `360.0`.

Output

The converted number or array of numbers, as `AbstractFloat`.

Example

Restrict an array of numbers in the range $[0, 2)$ as if they are angles expressed in radians:

```
julia> cirrange([4pi, 10, -5.23], 2.0*pi)
3-element Array{Float64,1}:
 0.0
 3.71681
 1.05319
```

Notes

This function does not support the `radians` keyword like IDL implementation. Use `max=2.0*pi` to restrict a number to the same interval.

Code of this function is based on IDL Astronomy User's Library.

CHAPTER 7

Related Projects

This is not the only effort to bundle astronomical functions written in Julia language. Other packages useful for more specific purposes are available at https://juliaastro.github.io/. A list of other packages is available at https://github.com/svaksha/Julia.jl/blob/master/Astronomy.md.

Because of this, some of IDL AstroLib's utilities are not provided in `AstroLib.jl` because already present in other Julia packages. Here is a list of such utilities:

- `aper`, see https://github.com/kbarbary/AperturePhotometry.jl
- `cosmo_param`, see `Cosmology` package (https://github.com/JuliaAstro/Cosmology.jl)
- `galage`, see `Cosmology` package (https://github.com/JuliaAstro/Cosmology.jl)
- `glactc_pm`, see `SkyCoords` package (https://github.com/kbarbary/SkyCoords.jl)
- `glactc`, see `SkyCoords` package (https://github.com/kbarbary/SkyCoords.jl)
- `lumdist`, see `Cosmology` package (https://github.com/JuliaAstro/Cosmology.jl)

In addition, there are similar projects for Python (Python AstroLib) and R (Astronomy Users Library).