AstroLib.jl Documentation

Release 0.0.7

Mose' Giordano

May 05, 2017

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

Installation

AstroLib.jl is available for Julia 0.6 and later versions, and can be installed with Julia built-in package manager. In a Julia session run the command

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

Older versions are also available for Julia 0.4 and 0.5.

Note that, in order to work, a few functions require external files, which are automatically downloaded when building the package. Should these files be missing for some reason, you will be able to load the package but some functions may not work properly. You can manually build the package with

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

CHAPTER 2

Usage

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

```
using AstroLib
```

Many functions in `AstroLib.jl` are compatible with Measurements.jl package, which allows you to define quantities with uncertainty and propagate the error when performing calculations according to propagation of uncertainty rules. For example:

```
using AstroLib, Measurements
mag2flux(12.54 ± 0.03)
# => 3.499451670283562e-14 ± 9.669342299577655e-16
```

CHAPTER 3

New Types

Observatory

AstroLib.jl defines a new Observatory type. This can be used to define a new object holding information about an observing site. It is a composite type whose fields are

- name (AbstractString type): the name of the site
- latitude (Real type): North-ward latitude of the site in degrees
- longitude (Real type): East-ward longitude of the site in degrees
- altitude (Real type): altitude of the site in meters
- tz (Real type): the number of hours of offset from UTC

The type constructor Observatory can be used to create a new Observatory object. Its syntax is

```
Observatory(name, lat, long, alt, tz)
```

name should be a string; lat, long, and tz should be anything that can be converted to a floating number with ten function; alt should be a real number.

A predefined list of some observing sites is provided with AstroLib.observatories constant. It is a dictionary whose keys are the abbreviated names of the observatories. For example, you can access information of the European Southern Observatory with

```
julia> obs = AstroLib.observatories["eso"]
Observatory: European Southern Observatory
latitude:    -29.256666666666668°N
longitude:   -70.73°E
altitude:    2347.0 m
time zone:   UTC-4

julia> obs.longitude
-70.73
```

You can list all keys of the dictionary with

```
keys(AstroLib.observatories)
```

Feel free to contribute new sites or adjust information of already present ones.

Planet

The package provides `Planet` type to hold information about Solar System planets. Its fields are

- Designation:
  - `name`: the name
- Physical characteristics:
  - `radius`: mean radius in meters
  - `eqradius`: equatorial radius in meters
  - `polradius`: polar radius in meters
  - `mass`: mass in kilogram
- Orbital characteristics (epoch J2000):
  - `ecc`: eccentricity of the orbit
  - `axis`: semi-major axis of the orbit in meters
  - `period`: sidereal orbital period in seconds

The constructor has this syntax:

```
Planet(name, radius, eqradius, polradius, mass, ecc, axis, period)
```

The list of Solar System planets, from Mercury to Pluto, is available with `AstroLib.planets` dictionary. The keys of this dictionary are the lowercase names of the planets. For example:

```
julia> AstroLib.planets["mercury"]
Planet:             Mercury
mean radius:        2.4397e6 m
equatorial radius: 2.4397e6 m
polar radius:       2.4397e6 m
mass:               3.3011e23 kg
eccentricity:       0.20563069
semi-major axis:    5.790905e10 m
period:             5.790905e10 s

julia> AstroLib.planets["mars"].eqradius
3.3962e6

julia> AstroLib.planets["saturn"].mass
5.6834e25
```

CHAPTER 4

How Can I Help?

`AstroLib.jl` is developed on GitHub at https://github.com/giordano/AstroLib.jl. You can contribute to the project in a number of ways: by translating more routines from IDL Astronomy User's Library, or providing brand-new functions, or even improving existing ones (make them faster and more precise). Also bug reports are encouraged.

CHAPTER 5

License

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

CHAPTER 6

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 7

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.

Astronomical Utilities

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 right 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

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

---

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.

---

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 $(227.23, -8.890)$.

```
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.

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.

- `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.

The output coordinates are always floating points and have the same type (scalar or array) as the input coordinates.

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?

```
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

`hadec2altaz` converts Hour Angle and Declination to Horizon (Alt-Az) coordinates.

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

---

bprecess

bprecess(*ra*, *dec*[, *epoch*]) → ra1950, dec1950
bprecess(*ra*, *dec*, *muradec*[, *parallax=parallax*, *radvel=radvel*]) → ra1950, dec1950

Purpose

Precess positions from J2000.0 (FK5) to B1950.0 (FK4).

Explanation

Calculates the mean place of a star at B1950.0 on the FK4 system from the mean place at J2000.0 on the FK5 system.

`bprecess` function has two methods, one for each of the following cases:

- the proper motion is known and non-zero
- the proper motion is unknown or known to be exactly zero (i.e. extragalactic radio sources). Better precision can be achieved in this case by inputting the epoch of the original observations.

Arguments

The function has 2 methods. The common mandatory arguments are:

- `ra`: input J2000 right ascension, in degrees.
- `dec`: input J2000 declination, in degrees.

The two methods have a different third argument (see "Explanation" section for more details). It can be one of the following:

- `muradec`: 2-element vector containing the proper motion in seconds of arc per tropical *century* in right ascension and declination.
- `epoch`: scalar giving epoch of original observations.

If none of these two arguments is provided (so `bprecess` is fed only with right ascension and declination), it is assumed that proper motion is exactly zero and `epoch = 2000`.

If it is used the method involving `muradec` argument, the following keywords are available:

- `parallax` (optional numerical keyword): stellar parallax, in seconds of arc.
- `radvel` (optional numerical keyword): radial velocity in km/s.

Right ascension and declination can be passed as the 2-tuple `(ra, dec)`. You can also pass `ra`, `dec`, `parallax`, and `radvel` as arrays, all of the same length N. In that case, `muradec` should be a matrix 2×N.

Output

The 2-tuple of right ascension and declination in 1950, in degrees, of input coordinates is returned. If `ra` and `dec` (and other possible optional arguments) are arrays, the 2-tuple of arrays `(ra1950, dec1950)` of the same length as the input coordinates is returned.

Method

The algorithm is taken from the Explanatory Supplement to the Astronomical Almanac 1992, page 186. See also Aoki et al (1983), A&A, 128, 263. URL: http://adsabs.harvard.edu/abs/1983A%26A...128..263A.

Example

The SAO2000 catalogue gives the J2000 position and proper motion for the star HD 119288. Find the B1950 position.

- RA(2000) = 13h 42m 12.740s
- Dec(2000) = 8d 23' 17.69''
- Mu(RA) = -.0257 s/yr
- Mu(Dec) = -.090 ''/yr

```
muradec = 100*[-15*0.0257, -0.090]; # convert to century proper motion
ra = ten(13, 42, 12.74)*15;
decl = ten(8, 23, 17.69);
adstring(bprecess(ra, decl, muradec), precision=2)
# => " 13 39 44.526  +08 38 28.63"
```

Notes

"When transferring individual observations, as opposed to catalog mean place, the safest method is to transform the observations back to the epoch of the observation, on the FK4 system (or in the system that was used to to produce the observed mean place), convert to the FK5 system, and transform to the the epoch and equinox of J2000.0" – from the Explanatory Supplement (1992), p. 180

`jprecess` performs the precession to J2000 coordinates.

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

---

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.

```
wave = reshape(1200:50:3150,40);
flux = ones(wave);
flux_new = calz_unred(wave, flux, -0.1);
```

Using a plotting tool you can visualize the unreddend flux. For example, with PyPlot.jl

```
using PyPlot
plot(wave, flux_new)
```

Notes

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

---

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`: 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

```
lst = ct2lst(-76.72, -4, DateTime(2008, 7, 30, 15, 53))
# => 11.356505172312609
sixty(lst)
# => 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.
longitude=ten(8, 43);
# Get number of Julian days.  Remember to subtract the time zone in
# order to convert local time to UTC.
jd = jdcnv(DateTime(2015, 11, 24, 13, 21) - Dates.Hour(1));
# Calculate Greenwich Mean Sidereal Time.
lst = ct2lst(longitude, jd)
# => 17.140685171005316
sixty(lst)
# => 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.

---

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

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

Notes

`jdcnv` is the inverse of this function.

---

deredd

`deredd`(*Eby*, *by*, *m1*, *c1*, *ub*) → by0, m0, c0, ub0

Purpose

Deredden stellar Stromgren parameters given for a value of E(b-y)

Arguments

- `Eby`: color index E(b-y), scalar (E(b-y) = 0.73*E(B-V))
- `by`: b-y color (observed)
- `m1`: Stromgren line blanketing parameter (observed)
- `c1`: Stromgren Balmer discontinuity parameter (observed)
- `ub`: u-b color (observed)

All arguments can be either scalars or arrays all of the same length.

Output

The 4-tuple `(by0, m0, c0, ub0)`.

- `by0`: b-y color (dereddened)
- `m0`: line blanketing index (dereddened)
- `c0`: Balmer discontinuity parameter (dereddened)
- `ub0`: u-b color (dereddened)

These are scalars or arrays of the same length as the input arguments.

Example

```
deredd(0.5, 0.2, 1.0, 1.0, 0.1)
# => (-0.3,1.165,0.905,-0.665)
```

Notes

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

eci2geo

eci2geo(*x*, *y*, *z*, *jd*) → latitude, longitude, altitude

Purpose

Convert Earth-centered inertial coordinates to geographic spherical coordinates.

Explanation

Converts from ECI (Earth-Centered Inertial) (x, y, z) rectangular coordinates to geographic spherical coordinates (latitude, longitude, altitude). Julian day is also needed as input.

ECI coordinates are in km from Earth center at the supplied time (True of Date). Geographic coordinates assume the Earth is a perfect sphere, with radius equal to its equatorial radius.

Arguments

- `x`: ECI x coordinate at `jd`, in kilometers.
- `y`: ECI y coordinate at `jd`, in kilometers.
- `z`: ECI z coordinate at `jd`, in kilometers.
- `jd`: Julian days.

The three coordinates can be passed as a 3-tuple `(x, y, z)`. In addition, `x`, `y`, `z`, and `jd` can be given as arrays of the same length.

Output

The 3-tuple of geographical coordinate (latitude, longitude, altitude).

- latitude: latitude, in degrees.
- longitude: longitude, in degrees.
- altitude: altitude, in kilometers.

If ECI coordinates are given as arrays, a 3-tuple of arrays of the same length is returned.

Example

Obtain the geographic direction of the vernal point on 2015-06-30T14:03:12.857, in geographic coordinates, at altitude 600 km. Note: equatorial radii of Solar System planets in meters are stored into `AstroLib.planets` dictionary.

```
x = AstroLib.planets["earth"].eqradius*1e-3 + 600;
lat, long, alt = eci2geo(x, 0, 0, jdcnv("2015-06-30T14:03:12.857"))
# => (0.0,230.87301833205856,600.0)
```

These coordinates can be further transformed into geodetic coordinates using `geo2geodetic` or into geomagnetic coordinates using `geo2mag`.

Notes

`geo2eci` converts geographic spherical coordinates to Earth-centered inertial coordinates.

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

---

eqpole

eqpole(l, b) $\to$ x, y

Purpose

Convert right ascension l and declination b to coordinate (x, y) using an equal-area polar projection.

Explanation

The output x and y coordinates are scaled to be in the range $[-90, 90]$ and to go from equator to pole to equator. Output map points can be centered on the north pole or south pole.

Arguments

- `l`: longitude, scalar or vector, in degrees
- `b`: latitude, same number of elements as right ascension, in degrees
- `southpole` (optional boolean keyword): keyword to indicate that the plot is to be centered on the south pole instead of the north pole. Default is `false`.

Output

The 2-tuple (x, y):

- x coordinate, same number of elements as right ascension, normalized to be in the range $[-90, 90]$.
- y coordinate, same number of elements as declination, normalized to be in the range $[-90, 90]$.

Example

```
eqpole(100, 35, southpole=true)
# => (-111.18287262822456,-19.604540237028665)
eqpole(80, 19)
# => (72.78853915267848,12.83458333897169)
```

Notes

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

---

flux2mag

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

Purpose

Convert from flux expressed in erg/(s cm^2 Å) to magnitudes.

Explanation

This is the reverse of `mag2flux`.

Arguments

- `flux`: the flux to be converted in magnitude, expressed in erg/(s cm^2 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

```
flux2mag(5.2e-15)
# => 14.609991640913002
flux2mag(5.2e-15, 15)
# => 20.709991640913003
flux2mag(5.2e-15, ABwave=15)
# => 27.423535345634598
```

Notes

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

---

gal_uvw

gal_uvw(*ra*, *dec*, *pmra*, *pmdec*, *vrad*, *plx*[, *lsr=true*]) → u, v, w

Purpose

Calculate the Galactic space velocity (u, v, w) of a star.

Explanation

Calculates the Galactic space velocity (u, v, w) of a star given its (1) coordinates, (2) proper motion, (3) parallax, and (4) radial velocity.

Arguments

User must supply a position, proper motion, radial velocity and parallax. Either scalars or arrays all of the same length can be supplied.

1. Position:

- `ra`: right ascension, in degrees
- `dec`: declination, in degrees

2. Proper Motion

- `pmra`: proper motion in right ascension in arc units (typically milli-arcseconds/yr). If given μ_α – proper motion in seconds of time/year – then this is equal to $15\mu_\alpha\cos(\text{dec})$.
- `pmdec`: proper motion in declination (typically mas/yr).

3. Radial Velocity

- `vrad`: velocity in km/s

4. Parallax

- `plx`: parallax with same distance units as proper motion measurements typically milliarcseconds (mas)

If you know the distance in parsecs, then set `plx` to $1000/\text{distance}$, if proper motion measurements are given in milli-arcseconds/yr.

There is an additional optional keyword:

- `lsr` (optional boolean keyword): if this keyword is set to `true`, then the output velocities will be corrected for the solar motion $(u, v, w)_\odot = (-8.5, 13.38, 6.49)$ (Coşkunoğlu et al. 2011 MNRAS, 412, 1237; DOI:10.1111/j.1365-2966.2010.17983.x) to the local standard of rest (LSR). Note that the value of the solar motion through the LSR remains poorly determined.

Output

The 3-tuple (u, v, w)

- u: velocity (km/s) positive toward the Galactic *anti*center
- v: velocity (km/s) positive in the direction of Galactic rotation
- w: velocity (km/s) positive toward the North Galactic Pole

Method

Follows the general outline of Johnson & Soderblom (1987, AJ, 93, 864; DOI:10.1086/114370) except that u is positive outward toward the Galactic *anti*center, and the J2000 transformation matrix to Galactic coordinates is taken from the introduction to the Hipparcos catalog.

Example

Compute the U,V,W coordinates for the halo star HD 6755. Use values from Hipparcos catalog, and correct to the LSR.

```
ra=ten(1,9,42.3)*15.; dec = ten(61,32,49.5);
pmra = 627.89;  pmdec = 77.84; # mas/yr
vrad = -321.4; dis = 129; # distance in parsecs
u, v, w = gal_uvw(ra, dec, pmra, pmdec, vrad, 1e3/dis, lsr=true)
# => (118.2110474553902,-466.4828898385057,88.16573278565097)
```

Notes

This function does not take distance as input. See “Arguments” section above for how to provide it using parallax argument.

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

---

geo2eci

`geo2eci`(*latitude*, *longitude*, *altitude*, *jd*) → x, y, z

Purpose

Convert geographic spherical coordinates to Earth-centered inertial coordinates.

Explanation

Converts from geographic spherical coordinates (latitude, longitude, altitude) to ECI (Earth-Centered Inertial) (x, y, z) rectangular coordinates. Julian days is also needed.

Geographic coordinates assume the Earth is a perfect sphere, with radius equal to its equatorial radius. ECI coordinates are in km from Earth center at epoch TOD (True of Date).

Arguments

- `latitude`: geographic latitude, in degrees.
- `longitude`: geographic longitude, in degrees.
- `altitude`: geographic altitude, in kilometers.
- `jd`: Julian days.

The three coordinates can be passed as a 3-tuple `(latitude, longitude, altitude)`. In addition, `latitude`, `longitude`, `altitude`, and `jd` can be given as arrays of the same length.

Output

The 3-tuple of ECI (x, y, z) coordinates, in kilometers. The TOD epoch is the supplied `jd` time.

If geographical coordinates are given as arrays, a 3-tuple of arrays of the same length is returned.

Example

Obtain the ECI coordinates of the intersection of the equator and Greenwich's meridian on 2015-06-30T14:03:12.857

```
geo2eci(0, 0, 0, jdcnv("2015-06-30T14:03:12.857"))
# => (-4024.8671780315185,4947.835465127513,0.0)
```

Notes

`eci2geo` converts Earth-centered inertial coordinates to geographic spherical coordinates.

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

---

geo2geodetic

`geo2geodetic`(*latitude*, *longitude*, *altitude*) → latitude, longitude, altitude
`geo2geodetic`(*latitude*, *longitude*, *altitude*, *planet*) → latitude, longitude, altitude
`geo2geodetic`(*latitude*, *longitude*, *altitude*, *equatorial_radius*, *polar_radius*) → latitude, longitude, altitude

Purpose

Convert from geographic (or planetographic) to geodetic coordinates.

Explanation

Converts from geographic (latitude, longitude, altitude) to geodetic (latitude, longitude, altitude). In geographic coordinates, the Earth is assumed a perfect sphere with a radius equal to its equatorial radius. The geodetic (or ellipsoidal) coordinate system takes into account the Earth's oblateness.

Geographic and geodetic longitudes are identical. Geodetic latitude is the angle between local zenith and the equatorial plane. Geographic and geodetic altitudes are both the closest distance between the satellite and the ground.

Arguments

The function has two base methods. The arguments common to all methods and always mandatory are `latitude`, `longitude`, and `altitude`:

- `latitude`: geographic latitude, in degrees.
- `longitude`: geographic longitude, in degrees.
- `altitude`: geographic altitude, in kilometers.

In order to convert to geodetic coordinates, you can either provide custom equatorial and polar radii of the planet or use the values of one of the planets of Solar System (Pluto included).

If you want to use the method with explicit equatorial and polar radii the additional mandatory arguments are:

- `equatorial_radius`: value of the equatorial radius of the body, in kilometers.
- `polar_radius`: value of the polar radius of the body, in kilometers.

Instead, if you want to use the method with the selection of a planet, the only additional argument is the planet name:

- `planet` (optional string argument): string with the name of the Solar System planet, from "Mercury" to "Pluto". If omitted (so, when only `latitude`, `longitude`, and `altitude` are provided), the default is "Earth".

In all cases, the three coordinates can be passed as a 3-tuple `(latitude, longitude, altitude)`. In addition, geographical `latitude`, `longitude`, and `altitude` can be given as arrays of the same length.

Output

The 3-tuple `(latitude, longitude, altitude)` in geodetic coordinates, for the body with specified equatorial and polar radii (Earth by default).

If geographical coordinates are given as arrays, a 3-tuple of arrays of the same length is returned.

Method

Stephen P. Keeler and Yves Nievergelt, "Computing geodetic coordinates", SIAM Rev. Vol. 40, No. 2, pp. 300-309, June 1998 (DOI:10.1137/S0036144597323921).

Planetary constants are from Planetary Fact Sheet (http://nssdc.gsfc.nasa.gov/planetary/factsheet/index.html).

Example

Locate the Earth geographic North pole (latitude: 90°, longitude: 0°, altitude 0 km), in geodetic coordinates:

```
geo2geodetic(90, 0, 0)
# => (90.0,0.0,21.38499999999931)
```

The same for Jupiter:

```
geo2geodetic(90, 0, 0, "Jupiter")
# => (90.0,0.0,4355.443799999994)
```

Find geodetic coordinates for point of geographic coordinates (latitude, longitude, altitude) = (43.16°, -24.32°, 3.87 km) on a planet with equatorial radius 8724.32 km and polar radius 8619.19 km:

```
geo2geodetic(43.16, -24.32, 3.87, 8724.32, 8619.19)
# => (43.849399515234516,-24.32,53.53354478670836)
```

Notes

Whereas the conversion from geodetic to geographic coordinates is given by an exact, analytical formula, the conversion from geographic to geodetic isn't. Approximative iterations (as used here) exist, but tend to become less good with increasing eccentricity and altitude. The formula used in this routine should give correct results within six digits for all spatial locations, for an ellipsoid (planet) with an eccentricity similar to or less than Earth's. More accurate results can be obtained via calculus, needing a non-determined amount of iterations.

In any case, the function `geodetic2geo`, which converts from geodetic (or planetodetic) to geographic coordinates, can be used to estimate the accuracy of `geo2geodetic`.

```
collect(geodetic2geo(geo2geodetic(67.2, 13.4, 1.2))) - [67.2, 13.4, 1.2]
# => 3-element Array{Float64,1}:
#     -3.56724e-9
#      0.0
#      9.47512e-10
```

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

---

geo2mag

`geo2mag`(*latitude*, *longitude*[, *year*]) → geomagnetic_latitude, geomagnetic_longitude

Purpose

Convert from geographic to geomagnetic coordinates.

Explanation

Converts from geographic (latitude, longitude) to geomagnetic (latitude, longitude). Altitude is not involved in this function.

Arguments

- `latitude`: geographic latitude (North), in degrees.
- `longitude`: geographic longitude (East), in degrees.
- `year` (optional numerical argument): the year in which to perform conversion. If omitted, defaults to current year.

The coordinates can be passed as arrays of the same length.

Output

The 2-tuple of magnetic (latitude, longitude) coordinates, in degrees.

If geographical coordinates are given as arrays, a 2-tuple of arrays of the same length is returned.

Example

Kyoto has geographic coordinates 35° 00’ 42” N, 135° 46’ 06” E, find its geomagnetic coordinates in 2016:

```
geo2mag(ten(35,0,42), ten(135,46,6), 2016)
# => (36.86579228937769,-60.184060536651614)
```

Notes

This function uses list of North Magnetic Pole positions provided by World Magnetic Model (https://www.ngdc.noaa.gov/geomag/data/poles/NP.xy).

`mag2geo` converts geomagnetical coordinates to geographic coordinates.

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

---

geodetic2geo

geodetic2geo(*latitude*, *longitude*, *altitude*) → latitude, longitude, altitude
geodetic2geo(*latitude*, *longitude*, *altitude*, *planet*) → latitude, longitude, altitude
geodetic2geo(*latitude*, *longitude*, *altitude*, *equatorial_radius*, *polar_radius*) → latitude, longitude, altitude

Purpose

Convert from geodetic (or planetodetic) to geographic coordinates.

Explanation

Converts from geodetic (latitude, longitude, altitude) to geographic (latitude, longitude, altitude). In geographic coordinates, the Earth is assumed a perfect sphere with a radius equal to its equatorial radius. The geodetic (or ellipsoidal) coordinate system takes into account the Earth’s oblateness.

Geographic and geodetic longitudes are identical. Geodetic latitude is the angle between local zenith and the equatorial plane. Geographic and geodetic altitudes are both the closest distance between the satellite and the ground.

Arguments

The function has two base methods. The arguments common to all methods and always mandatory are `latitude`, `longitude`, and `altitude`:

- `latitude`: geodetic latitude, in degrees.
- `longitude`: geodetic longitude, in degrees.
- `altitude`: geodetic altitude, in kilometers.

In order to convert to geographic coordinates, you can either provide custom equatorial and polar radii of the planet or use the values of one of the planets of Solar System (Pluto included).

If you want to use the method with explicit equatorial and polar radii the additional mandatory arguments are:

- `equatorial_radius`: value of the equatorial radius of the body, in kilometers.
- `polar_radius`: value of the polar radius of the body, in kilometers.

Instead, if you want to use the method with the selection of a planet, the only additional argument is the planet name:

- `planet` (optional string argument): string with the name of the Solar System planet, from "Mercury" to "Pluto". If omitted (so, when only `latitude`, `longitude`, and `altitude` are provided), the default is "Earth".

In all cases, the three coordinates can be passed as a 3-tuple `(latitude, longitude, altitude)`. In addition, geodetic `latitude`, `longitude`, and `altitude` can be given as arrays of the same length.

Output

The 3-tuple `(latitude, longitude, altitude)` in geographic coordinates, for the body with specified equatorial and polar radii (Earth by default).

If geodetic coordinates are given as arrays, a 3-tuple of arrays of the same length is returned.

Method

Stephen P. Keeler and Yves Nievergelt, "Computing geodetic coordinates", SIAM Rev. Vol. 40, No. 2, pp. 300-309, June 1998 (DOI:10.1137/S0036144597323921).

Planetary constants from "Allen's Astrophysical Quantities", Fourth Ed., (2000).

Example

Find geographic coordinates of geodetic North pole (latitude: 90°, longitude: 0°, altitude 0 km) of the Earth:

```
geodetic2geo(90, 0, 0)
# => (90.0,0.0,-21.38499999999931)
```

The same for Jupiter:

```
geodetic2geo(90, 0, 0, "Jupiter")
# => (90.0,0.0,-4355.443799999994)
```

Find geographic coordinates for point of geodetic coordinates (latitude, longitude, altitude) = (43.16°, -24.32°, 3.87 km) on a planet with equatorial radius 8724.32 km and polar radius 8619.19 km:

```
geodetic2geo(43.16, -24.32, 3.87, 8724.32, 8619.19)
# => (42.46772711708433,-24.32,-44.52902080669082)
```

Notes

`geo2geodetic` converts from geographic (or planetographic) to geodetic coordinates.

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

---

get_date

get_date([*date*, *old=true*, *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. If omitted, defaults to the current UTC time. It can be either a single date or an array of dates. Each element can be either a `DateTime` type or anything that can be converted to that type. In the case of vectorial input, each element is considered as a date, so you cannot provide a date by parts.
- `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 `CCYY` represents a 4-digit calendar year, `MM` the 2-digit ordinal number of a calendar month within the calendar year, and `DD` 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

```
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

---

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

```
get_juldate()
daycnv(get_juldate())
```

Notes

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

---

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

```
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.

hadec2altaz

hadec2altaz (*ha*, *dec*, *lat*[, *ws=true*]) → alt, az

Purpose

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

Explanation

Can deal with the NCP singularity. Intended mainly to be used by program `eq2hor`.

Arguments

Input coordinates may be either a scalar or an array, of the same dimension.

- `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.
- `lat`: the local geodetic latitude, in degrees, scalar or array.
- `ws` (optional boolean keyword): if true, the output azimuth is measured West from South. The default is to measure azimuth East from North.

`ha` and `dec` can be given as a 2-tuple `(ha, dec)`.

Output

2-tuple `(alt, az)`

- `alt`: local apparent altitude, in degrees.
- `az`: the local apparent azimuth, in degrees.

The output coordinates are always floating points and have the same type (scalar or array) as the input coordinates.

Example

Arcturus is observed at an apparent hour angle of 336.6829 and a declination of 19.1825 while at the latitude of +43° 4' 42''. What are the local altitude and azimuth of this object?

```
alt, az = hadec2altaz(336.6829, 19.1825, ten(43, 4, 42))
# => (59.08617155005683,133.3080693440254)
```

Notes

`altaz2hadec` converts Horizon (Alt-Az) coordinates to Hour Angle and Declination.

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

helio_jd

helio_jd(*date*, *ra*, *dec*[, *B1950=true*, *diff=false*]) → jd_helio
helio_jd(*date*, *ra*, *dec*[, *B1950=true*, *diff=true*]) → time_diff

Purpose

Convert geocentric (reduced) Julian date to heliocentric Julian date.

Explanation

This procedure corrects for the extra light travel time between the Earth and the Sun.

An online calculator for this quantity is available at http://www.physics.sfasu.edu/astro/javascript/hjd.html

Users requiring more precise calculations and documentation should look at the IDL code available at http://astroutils.astronomy.ohio-state.edu/time/

Arguments

- `date`: reduced Julian date (= JD - 2400000), it can be either a scalar or vector. You can use `juldate()` to calculate the reduced Julian date.
- `ra` and `dec`: scalars giving right ascension and declination in degrees. Default equinox is J2000.
- `B1950` (optional boolean keyword): if set to `true`, then input coordinates are assumed to be in equinox B1950 coordinates. Default is `false`.
- `diff` (optional boolean keyword): if set to `true`, the function returns the time difference (heliocentric JD - geocentric JD) in seconds. Default is `false`.

Output

The return value depends on the value of `diff` optional keywords:

- if `diff` is `false` (default), then the heliocentric reduced Julian date is returned.
- if `diff` is `true`, then the time difference in seconds between the geocentric and heliocentric Julian date is returned.

Example

What is the heliocentric Julian date of an observation of V402 Cygni (J2000: RA = 20 9 7.8, Dec = 37 09 07) taken on June 15, 2016 at 11:40 UT?

```
jd = juldate(2016, 6, 15, 11, 40);
helio_jd(jd, ten(20, 9, 7.8)*15, ten(37, 9, 7))
# => 57554.98808289718
```

Notes

Wayne Warren (Raytheon ITSS) has compared the results of this algorithm with the FORTRAN subroutines in the STARLINK SLALIB library (see http://star-www.rl.ac.uk/).

```
                                                  Time Diff (sec)
     Date                   RA(2000)   Dec(2000)  STARLINK      IDL

1999-10-29T00:00:00.0  21 08 25.  -67 22 00.  -59.0         -59.0
1999-10-29T00:00:00.0  02 56 33.4 +00 26 55.  474.1         474.1
1940-12-11T06:55:00.0  07 34 41.9 -00 30 42.  366.3         370.2
1992-02-29T03:15:56.2  12 56 27.4 +42 10 17.  350.8         350.9
2000-03-01T10:26:31.8  14 28 36.7 -20 42 11.  243.7         243.7
2100-02-26T09:18:24.2  08 26 51.7 +85 47 28.  104.0         108.8
```

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

---

helio_rv

helio_rv (*jd*, *T*, *P*, *V_0*, *K*[, *e*, *ω*]) → rv

Purpose

Return the heliocentric radial velocity of a spectroscopic binary.

Explanation

This function will return the heliocentric radial velocity of a spectroscopic binary star at a given heliocentric date given its orbit.

Arguments

- `jd`: time of observation, as number of Julian days. It can be either a scalar or an array.
- `T`: time of periastron passage (max. +ve velocity for circular orbits), same time system as `jd`
- `P`: the orbital period in same units as `jd`
- `V_0`: systemic velocity
- `K`: velocity semi-amplitude in the same units as `V_0`
- `e`: eccentricity of the orbit. It defaults to 0 if omitted
- `ω`: longitude of periastron in degrees. It defaults to 0 if omitted

Output

The predicted heliocentric radial velocity in the same units as Gamma for the date(s) specified by `jd`. It is a scalar or an array depending on the type of`jd`. ##### Example ####

1. What was the heliocentric radial velocity of the primary component of HU Tau at 1730 UT 25 Oct 1994?

```
jd = juldate(94, 10, 25, 17, 30); # Obtain Geocentric Julian days
hjd = helio_jd(jd, ten(04, 38, 16)*15, ten(20, 41, 05)); # Convert to HJD
helio_rv(hjd, 46487.5303, 2.0563056, -6, 59.3)
# => -62.965570109145034
```

NB: the functions `juldate` and `helio_jd` return a reduced HJD (HJD - 2400000) and so T and P must be specified in the same fashion.

2. Plot two cycles of an eccentric orbit, $e = 0.6$, $\omega = 45$ for both components of a binary star. Use PyPlot.jl for plotting.

```
using PyPlot
ϕ = linspace(0, 2, 1000); # Generate 1000 phase points
plot(ϕ ,helio_rv(ϕ, 0, 1, 0, 100, 0.6, 45)) # Plot 1st component
plot(ϕ ,helio_rv(ϕ, 0, 1, 0, 100, 0.6, 45+180)) # Plot 2nd component
```

Notes

The user should ensure consistency with all time systems being used (i.e. `jd` and `t` should be in the same units and time system). Generally, users should reduce large time values by subtracting a large constant offset, which may improve numerical accuracy.

If using the the function `juldate` and `helio_jd`, the reduced HJD time system must be used throughtout.

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

---

jdcnv

jdcnv(*date*) → 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 in proleptic Gregorian Calendar. Can be either a single date or an array of dates. Each element can be either a `DateTime` type or anything that can be converted directly to `DateTime`. In the case of vectorial input, each element is considered as a date, so you cannot provide a date by parts.

Output

Number of Julian days, as a floating point.

Example

Find the Julian days number at 2016 August 23, 03:39:06.

```
jdcnv(DateTime(2016, 08, 23, 03, 39, 06))
# => 2.4576236521527776e6
jdcnv(2016, 08, 23, 03, 39, 06)
# => 2.4576236521527776e6
jdcnv("2016-08-23T03:39:06")
# => 2.4576236521527776e6
```

Notes

This is the inverse of `daycnv`.

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

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

---

jprecess

jprecess (*ra*, *dec*[, *epoch*]) → ra2000, dec2000
jprecess (*ra*, *dec*, *muradec*[, *parallax=parallax*, *radvel=radvel*]) → ra2000, dec2000

Purpose

Precess positions from B1950.0 (FK4) to J2000.0 (FK5).

Explanation

Calculate the mean place of a star at J2000.0 on the FK5 system from the mean place at B1950.0 on the FK4 system.

`jprecess` function has two methods, one for each of the following cases:

- the proper motion is known and non-zero
- the proper motion is unknown or known to be exactly zero (i.e. extragalactic radio sources). Better precision can be achieved in this case by inputting the epoch of the original observations.

Arguments

The function has 2 methods. The common mandatory arguments are:

- `ra`: input B1950 right ascension, in degrees.
- `dec`: input B1950 declination, in degrees.

The two methods have a different third argument (see "Explanation" section for more details). It can be one of the following:

- `muradec`: 2-element vector containing the proper motion in seconds of arc per tropical *century* in right ascension and declination.
- `epoch`: scalar giving epoch of original observations.

If none of these two arguments is provided (so `jprecess` is fed only with right ascension and declination), it is assumed that proper motion is exactly zero and `epoch = 1950`.

If it is used the method involving `muradec` argument, the following keywords are available:

- `parallax` (optional numerical keyword): stellar parallax, in seconds of arc.
- `radvel` (optional numerical keyword): radial velocity in km/s.

Right ascension and declination can be passed as the 2-tuple `(ra, dec)`. You can also pass `ra`, `dec`, `parallax`, and `radvel` as arrays, all of the same length N. In that case, `muradec` should be a matrix 2×N.

Output

The 2-tuple of right ascension and declination in 2000, in degrees, of input coordinates is returned. If `ra` and `dec` (and other possible optional arguments) are arrays, the 2-tuple of arrays `(ra2000, dec2000)` of the same length as the input coordinates is returned.

Method

The algorithm is taken from the Explanatory Supplement to the Astronomical Almanac 1992, page 184. See also Aoki et al (1983), A&A, 128, 263. URL: http://adsabs.harvard.edu/abs/1983A%26A...128..263A.

Example

The SAO catalogue gives the B1950 position and proper motion for the star HD 119288. Find the J2000 position.

- RA(1950) = 13h 39m 44.526s
- Dec(1950) = 8d 38’ 28.63’‘
- Mu(RA) = -.0259 s/yr
- Mu(Dec) = -.093 ‘’/yr

```
muradec = 100*[-15*0.0259, -0.093]; # convert to century proper motion
ra = ten(13, 39, 44.526)*15;
decl = ten(8, 38, 28.63);
adstring(jprecess(ra, decl, muradec), precision=2)
# => " 13 42 12.740  +08 23 17.69"
```

Notes

“When transferring individual observations, as opposed to catalog mean place, the safest method is to tranform the observations back to the epoch of the observation, on the FK4 system (or in the system that was used to to produce the observed mean place), convert to the FK5 system, and transform to the the epoch and equinox of J2000.0” – from the Explanatory Supplement (1992), p. 180

`bprecess` performs the precession to B1950 coordinates.

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

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 in Julian Calendar, UTC standard. It can be either e single date or an array of dates. Each element can be given in `DateTime` type or anything that can be converted to that type. In the case of vectorial input, each element is considered as a date, so you cannot provide a date by parts.

Output

The number of Reduced Julian Days is returned. If `date` is an array, an array of the same length as `date` is returned.

Example

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

```
juldate(DateTime(2016, 03, 20, 15, 24))
# => 57468.14166666667
juldate(2016, 03, 20, 15, 24)
# => 57468.14166666667
juldate("2016-03-20T15:24")
# => 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.

---

kepler_solver

kepler_solver(*M*, *e*) → E

Purpose

Solve Kepler's equation in the elliptic motion regime ($0 \leq e \leq 1$) and return eccentric anomaly E.

Explanation

In order to find the position of a body in elliptic motion (e.g., in the two-body problem) at a given time t, one has to solve the Kepler's equation

$$M(t) = E(t) - e \sin E(t)$$

where $M(t) = (t - t_0)/P$ is the mean anomaly, $E(t)$ the eccentric anomaly, e the eccentricity of the orbit, t_0 is the time of periapsis passage, and P is the period of the orbit. Usually the eccentricity is given and one wants to find the eccentric anomaly $E(t)$ at a specific time t, so that also the mean anomaly $M(t)$ is known.

Arguments

- `M`: mean anomaly. This can be either a scalar or an array
- `e`: eccentricity, in the elliptic motion regime ($0 \leq e \leq 1$)

Output

The eccentric anomaly E, restricted to the range $[-\pi, \pi]$. If an array of mean anomalies is provided in input, an array of the same length as `M` is returned.

Method

Many different numerical methods exist to solve Kepler's equation. This function implements the algorithm proposed in Markley (1995) Celestial Mechanics and Dynamical Astronomy, 63, 101 (DOI:10.1007/BF00691917). This method is not iterative, requires only four transcendental function evaluations, and has been proved to be fast and efficient over the entire range of elliptic motion $0 \leq e \leq 1$.

Example

1. Find the eccentric anomaly for an orbit with eccentricity $e = 0.7$ and for $M(t) = 8\pi/3$.

```
ecc = 0.7;
E = kepler_solver(8pi/3, ecc)
# => 2.5085279492864223
```

2. Plot the eccentric anomaly as a function of mean anomaly for eccentricity $e = 0, 0.5, 0.9$. Recall that `kepler_solver` gives $E \in [-\pi, \pi]$, use `cirrange` to have it in $[0, 2\pi]$. Use PyPlot.jl for plotting.

```
using PyPlot
M=linspace(0, 2pi, 1001)[1:end-1];
for ecc in (0, 0.5, 0.9); plot(M, cirrange(kepler_solver(M, ecc), 2pi)); end
```

Notes

The true anomaly can be calculated with `trueanom` function.

---

lsf_rotate

`lsf_rotate` (*delta_v*, *v_sin_i*[, *epsilon = 0.3*]) → velocity_grid, lsf

Purpose

Create a 1-d convolution kernel to broaden a spectrum from a rotating star.

Explanation

Can be used to derive the broadening effect (LSF, line spread function) due to rotation on a synthetic stellar spectrum. Assumes constant limb darkening across the disk.

Arguments

- `delta_v`: numeric scalar giving the step increment (in km/s) in the output rotation kernel
- `v_sin_i`: the rotational velocity projected along the line of sight (km/s)
- `epsilon` (optional numeric argument): numeric scalar giving the limb-darkening coefficient, default = 0.6 which is typical for photospheric lines. The specific intensity I at any angle θ from the specific intensity I_{cen} at the center of the disk is given by:

$$I = I_{\text{cen}} \cdot (1 - \varepsilon \cdot (1 - \cos(\theta)))$$

Output

The 2-tuple (`velocity_grid`, `lsf`):

- `velocity_grid`: vector of velocity grid with the same number of elements as `lsf` (see below)
- `lsf`: the convolution kernel vector for the specified rotational velocity. The number of points in `lsf` will be always be odd (the kernel is symmetric) and equal to either `ceil(2*v_sin_i/delta_v)` or `ceil(2*v_sin_i/delta_v) + 1`, whichever number is odd. Elements of `lsf` will always be of type `AbstractFloat`. To actually compute the broadening, the spectrum should be convolved with the rotational `lsf`

Example

Plot the line spread function for a star rotating at 90 km/s in velocity space every 3 km/s. Use PyPlot.jl for plotting.

```
using PyPlot
plot(lsf_rotate(3, 90)...)
```

Notes

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

---

mag2flux

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

Purpose

Convert from magnitudes to flux expressed in erg/(s cm^2 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

```
mag2flux(8.3)
# => 1.7378008287493692e-12
mag2flux(8.3, 12)
# => 7.58577575029182e-9
mag2flux(8.3, ABwave=12)
# => 3.6244115683017193e-7
```

Notes

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

---

mag2geo

mag2geo (*latitude*, *longitude*[, *year*]) → geographic_latitude, geographic_longitude

Purpose

Convert from geomagnetic to geographic coordinates.

Explanation

Converts from geomagnetic (latitude, longitude) to geographic (latitude, longitude). Altitude is not involved in this function.

Arguments

- `latitude`: geomagnetic latitude (North), in degrees.
- `longitude`: geomagnetic longitude (East), in degrees.
- `year` (optional numerical argument): the year in which to perform conversion. If omitted, defaults to current year.

The coordinates can be passed as arrays of the same length.

Output

The 2-tuple of geographic (latitude, longitude) coordinates, in degrees.

If geomagnetic coordinates are given as arrays, a 2-tuple of arrays of the same length is returned.

Example

Find position of North Magnetic Pole in 2016

```
mag2geo(90, 0, 2016)
# => (86.395,-166.29000000000002)
```

Notes

This function uses list of North Magnetic Pole positions provided by World Magnetic Model (https://www.ngdc.noaa.gov/geomag/data/poles/NP.xy).

`geo2mag` converts geographic coordinates to geomagnetic coordinates.

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

---

month_cnv

month_cnv(*number*[, *shor=true*, *up=true*, *low=true*]) → month_name
month_cnv(*name*) → number

Purpose

Convert between a month English name and the equivalent number.

Explanation

For example, converts from "January" to 1 or vice-versa.

Arguments

The functions has two methods, one with numeric input (and three possible boolean keywords) and the other one with string input.

Numeric input arguments:

- `number`: the number of the month to be converted to month name.
- `short` (optional boolean keyword): if true, the abbreviated (3-character) name of the month will be returned, e.g. "Apr" or "Oct". Default is false.
- `up` (optional boolean keyword): if true, the name of the month will be all in upper case, e.g. "APRIL" or "OCTOBER". Default is false.
- `low` (optional boolean keyword): if true, the name of the month will be all in lower case, e.g. "april" or "october". Default is false.

String input argument:

- `name`: month name to be converted to month number.

All mandatory arguments can be provided either as a single element or as an array.

Output

The month name or month number, depending on the input. For numeric input, the format of the month name is influenced by the optional keywords.

Example

```
month_cnv(["janua", "SEP", "aUgUsT"])
# => 3-element Array{Integer,1}:
#      1
#      9
#      8
month_cnv([2, 12, 6], short=true, low=true)
# => 3-element Array{UTF8String,1}:
#      "feb"
#      "dec"
#      "jun"
```

moonpos

moonpos (*jd*[, *radians=true*]) → ra, dec, dis, geolong, geolat

Purpose

Compute the right ascension and declination of the Moon at specified Julian date.

Arguments

- `jd`: the Julian ephemeris date. It can be either a scalar or an array
- `radians` (optional boolean keyword): if set to `true`, then all output angular quantities are given in radians rather than degrees. The default is `false`

Output

The 5-tuple `(ra, dec, dis, geolong, geolat)`:

- `ra`: apparent right ascension of the Moon in degrees, referred to the true equator of the specified date(s)
- `dec`: the declination of the Moon in degrees
- `dis`: the distance between the centre of the Earth and the centre of the Moon in kilometers
- `geolong`: apparent longitude of the moon in degrees, referred to the ecliptic of the specified date(s)
- `geolat`: apparent longitude of the moon in degrees, referred to the ecliptic of the specified date(s)

If `jd` is an array, then all output quantities are arrays of the same length as `jd`.

Method

Derived from the Chapront ELP2000/82 Lunar Theory (Chapront-Touze' and Chapront, 1983, 124, 50), as described by Jean Meeus in Chapter 47 of ''Astronomical Algorithms'' (Willmann-Bell, Richmond), 2nd edition, 1998. Meeus quotes an approximate accuracy of 10" in longitude and 4" in latitude, but he does not give the time range for this accuracy.

Comparison of the IDL procedure with the example in ''Astronomical Algorithms'' reveals a very small discrepancy (~1 km) in the distance computation, but no difference in the position calculation.

Example

1. Find the position of the moon on April 12, 1992

```
jd = jdcnv(1992, 4, 12);
adstring(moonpos(jd)[1:2],precision=1)
# => " 08 58 45.23  +13 46 06.1"
```

This is within 1" from the position given in the Astronomical Almanac.

2. Plot the Earth-moon distance during 2016 with sampling of 6 hours. Use PyPlot.jl for plotting

```
using PyPlot
points = DateTime(2016):Dates.Hour(6):DateTime(2017);
plot(points, moonpos(jdcnv(points))[3])
```

Notes

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

mphase

mphase(*jd*) $\rightarrow$ k

Purpose

Return the illuminated fraction of the Moon at given Julian date(s).

Arguments

- `jd`: the Julian ephemeris date. It can be either a scalar or an array.

Output

The illuminated fraction k of Moon's disk, with $0 \leq k \leq 1$. $k = 0$ indicates a new moon, while $k = 1$ stands for a full moon. If `jd` is given as an array, an array of the same number of elements as `jd` is returned.

Method

Algorithm from Chapter 46 of "Astronomical Algorithms" by Jean Meeus (Willmann-Bell, Richmond) 1991. `sunpos` and `moonpos` are used to get positions of the Sun and the Moon, and the Moon distance. The selenocentric elongation of the Earth from the Sun (phase angle) is then computed, and used to determine the illuminated fraction.

Example

Plot the illuminated fraction of the Moon for every day in January 2018 with a hourly sampling. Use PyPlot.jl for plotting

```
using PyPlot
points = DateTime(2018,01,01):Dates.Hour(1):DateTime(2018,01,31,23,59,59);
plot(points, mphase(jdcnv(points)))
```

Note that in this calendar month there are two full moons, this event is called blue moon.

Notes

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

nutate

nutate (*jd*) → long, obliq

Purpose

Return the nutation in longitude and obliquity for a given Julian date.

Arguments

- `jd`: Julian ephemeris date, it can be either a scalar or a vector

Output

The 2-tuple `(long, obliq)`, where

- `long`: the nutation in longitude
- `obl`: the nutation in latitude

If `jd` is an array, `long` and `obl` are arrays of the same length.

Method

Uses the formula in Chapter 22 of ‘‘Astronomical Algorithms” by Jean Meeus (1998, 2nd ed.) which is based on the 1980 IAU Theory of Nutation and includes all terms larger than 0.0003”.

Example

1. Find the nutation in longitude and obliquity 1987 on Apr 10 at Oh. This is example 22.a from Meeus

```
jd = jdcnv(1987, 4, 10);
nutate(jd)
# => (-3.787931077110755,9.442520698644401)
```

2. Plot the daily nutation in longitude and obliquity during the 21st century. Use PyPlot.jl for plotting.

```
using PyPlot
years = DateTime(2000):DateTime(2100);
long, obl = nutate(jdcnv(years));
plot(years, long); plot(years, obl)
```

You can see both the dominant large scale period of nutation, of 18.6 years, and smaller oscillations with shorter periods.

Notes

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

paczynski

paczynski(u) $\rightarrow$ amplification

Purpose

Calculate gravitational microlensing amplification of a point-like source by a single point-like lens.

Explanation

Return the gravitational microlensing amplification of a point-like source by a single point-like lens, using Paczyński formula

$$A(u) = \frac{u^2 + 2}{u\sqrt{u^2 + 4}}$$

where u is the projected distance between the lens and the source in units of Einstein radii.

In order to speed up calculations for extreme values of u, the following asyntotic expressions for $A(u)$ are used:

$$A(u) = \begin{cases} 1/u & |u| \ll 1 \\ \operatorname{sgn}(u) & |u| \gg 1 \end{cases}$$

Arguments

- `u`: projected distance between the lens and the source, in units of Einstein radii

The distance can be either a scalar or an array.

Output

The microlensing amplification for the given distance. If `u` is passed as an array, an array of the same length is returned.

Example

Calculate the microlensing amplification for $u = 10^{-10}, 10^{-1}, 1, 10, 10^{10}$:

```
paczynski([1e-10, 1e-1, 1, 10, 1e10])
# => 5-element Array{Float64,1}:
#      1.0e10
#     10.0375
#      1.34164
#      1.00019
#      1.0
```

Notes

The expression of $A(u)$ of microlensing amplification has been given by Bohdan Paczyński in

- Paczynski, B. 1986, ApJ, 304, 1. DOI:10.1086/164140, Bibcode:1986ApJ...304....1P

The same expression was actually found by Albert Einstein half a century earlier:

- Einstein, A. 1936, Science, 84, 506. DOI:10.1126/science.84.2188.506, Bibcode:1936Sci....84..506E

planck_freq

planck_freq (*frequency*, *temperature*) $\rightarrow$ black_body_flux

Purpose

Calculate the flux of a black body per unit frequency.

Explanation

Return the spectral radiance of a black body per unit frequency using Planck's law

$$B_\nu(\nu, T) = \frac{2h\nu^3}{c^2}\frac{1}{e^{\frac{h\nu}{k_\mathrm{B}T}} - 1}$$

Arguments

- `frequency`: frequency at which the flux is to be calculated, in Hertz.
- `temperature`: the equilibrium temperature of the black body, in Kelvin.

Both arguments can be either scalar or arrays of the same length.

Output

The spectral radiance of the black body, in units of W/(sr·m^2·Hz).

Example

Plot the spectrum of a black body in $[10^{12}, 10^{15.4}]$ Hz at 8000 K. Use PyPlot.jl for plotting.

```
using PyPlot
frequency=logspace(12, 15.4, 1000);
temperature=ones(frequency)*8000;
flux=planck_freq(frequency, temperature);
plot(frequency, flux)
```

Notes

`planck_wave` calculates the flux of a black body per unit wavelength.

planck_wave

planck_wave (*wavelength*, *temperature*) $\rightarrow$ black_body_flux

Purpose

Calculate the flux of a black body per unit wavelength.

Explanation

Return the spectral radiance of a black body per unit wavelength using Planck's law

$$B_\lambda(\lambda, T) = \frac{2hc^2}{\lambda^5}\frac{1}{e^{\frac{hc}{\lambda k_\mathrm{B}T}} - 1}$$

Arguments

- `wavelength`: wavelength at which the flux is to be calculated, in meters.
- `temperature`: the equilibrium temperature of the black body, in Kelvin.

Both arguments can be either scalar or arrays of the same length.

Output

The spectral radiance of the black body, in units of W/(sr·m^3).

Example

Plot the spectrum of a black body in $[0, 3]$ µm at 5000 K. Use PyPlot.jl for plotting.

```
using PyPlot
wavelength=linspace(0, 3e-6, 1000);
temperature=ones(wavelength)*5000;
flux=planck_wave(wavelength, temperature);
plot(wavelength, flux)
```

Notes

`planck_freq` calculates the flux of a black body per unit frequency.

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

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 $(r, \varphi) = (1.7, 227)$, with angle φ expressed in degrees.

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

---

posang

posang(*units*, *ra1*, *dec1*, *ra2*, *dec2*) → angular_distance

Purpose

Compute rigorous position angle of point 2 relative to point 1.

Explanation

Computes the rigorous position angle of point 2 (with given right ascension and declination) using point 1 (with given right ascension and declination) as the center.

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 degrees

  - 2: right ascensions and declinations are in degrees, output distance in degrees

- `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

Angle of the great circle containing `[ra2, dec2]` from the meridian containing `[ra1, dec1]`, in the sense north through east rotating about `[ra1, dec1]`. See `units` argument above for units.

Method

The "four-parts formula" from spherical trigonometry (p. 12 of Smart's Spherical Astronomy or p. 12 of Green' Spherical Astronomy).

Example

Mizar has coordinates (ra, dec) = (13h 23m 55.5s, +54° 55' 31''). Its companion, Alcor, has coordinates (ra, dec) = (13h 25m 13.5s, +54° 59' 17''). Find the position angle of Alcor with respect to Mizar.

```
posang(1, ten(13, 25, 13.5), ten(54, 59, 17), ten(13, 23, 55.5), ten(54, 55, 31))
# => -108.46011246802047
```

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.
- Note that `posang` is not commutative: the position angle between A and B is θ, then the position angle between B and A is $180 + \theta$.

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

---

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

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

```
ra, dec = ten(21, 59, 33.053)*15, ten(-56, 59, 33.053)
# => (329.88772083333333,-56.992514722222225)
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). If you need better accuracy, use `bprecess` or `jprecess` as needed.

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

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.

```
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.

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

```
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.

---

radec

`radec`(*ra::Real*, *dec::Real*[, *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

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

---

recpol

recpol (*x*, *y*[, *degrees=true*]) → 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 $[-\pi, \pi]$ 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, φ) of point with rectangular coordinates $(x, y) = (2.24, -1.87)$.

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

Angle φ is given in radians.

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 ρ and position angle θ 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 (ρ, θ), where

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

Example

Find the position of Eta Coronae Borealis at the epoch 2016

```
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.

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

```
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.

---

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

```
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.

sunpos

`sunpos` (*jd*[, *radians=true*]) → ra, dec, elong, obliquity

Purpose

Compute the right ascension and declination of the Sun at a given date.

Arguments

- `jd`: the Julian date of when you want to calculate Sun position. It can be either a scalar or a vector. Use `jdcnv` to get the Julian date for a given date and time.
- `radians` (optional boolean keyword): if set to `true`, all output quantities are given in radians. The default is `false`, so all quantities are given in degrees.

Output

The 4-tuple `(ra, dec, elong, obliquity)`:

- `ra`: the right ascension of the Sun at that date
- `dec`: the declination of the Sun at that date
- `elong`: ecliptic longitude of the Sun at that date
- `obliquity`: the obliquity of the ecliptic

All quantities are given in degrees, unless `radians` keyword is set to `true` (see "Arguments" section). If `jd` is an array, arrays of the same given as `jd` are returned.

Method

Uses a truncated version of Newcomb's Sun. Adapted from the IDL routine SUN_POS by CD Pike, which was adapted from a FORTRAN routine by B. Emerson (RGO).

Example

1. Find the apparent right ascension and declination of the Sun on May 1, 1982

```
adstring(sunpos(jdcnv(1982, 5, 1))[1:2], precision=2)
# => " 02 31 32.614  +14 54 34.92"
```

The Astronomical Almanac gives `02 31 32.58 +14 54 34.9` so the error for this case is < 0.5”.

2. Plot the apparent right ascension, in hours, and declination of the Sun, in degrees, for every day in 2016. Use PyPlot.jl for plotting.

```
using PyPlot
days = DateTime(2016):DateTime(2016, 12, 31);
ra, declin = sunpos(jdcnv(days));
plot(days, ra/15); plot(days, declin)
```

Notes

Patrick Wallace (Rutherford Appleton Laboratory, UK) has tested the accuracy of a C adaptation of the present algorithm and found the following results. From 1900-2100 `sunpos` gave 7.3 arcsec maximum error, 2.6 arcsec RMS. Over the shorter interval 1950-2050 the figures were 6.4 arcsec max, 2.2 arcsec RMS.

The returned `ra` and `dec` are in the given date’s equinox.

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

---

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 any iterable like `(deg, min, sec)` or `[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 or iterables.

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

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

Example

```
ten(-0.0, 19, 47)
# => -0.3297222222222222
ten("+5:14:58")
# => 5.249444444444444
ten("-10 26")
# => -10.433333333333334
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.

---

trueanom

trueanom(E, e) → true anomaly

Purpose

Calculate true anomaly for a particle in elliptic orbit with eccentric anomaly E and eccentricity e.

Explanation

In the two-body problem, once that the Kepler's equation is solved and $E(t)$ is determined, the polar coordinates $(r(t), \theta(t))$ of the body at time t in the elliptic orbit are given by

$$\theta(t) = 2\arctan\left(\sqrt{\frac{1+e}{1-e}}\tan\frac{E(t)}{2}\right)$$

$$r(t) = \frac{a(1-e^2)}{1 + e\cos(\theta(t) - \theta_0)}$$

in which a is the semi-major axis of the orbit, and θ_0 the value of angular coordinate at time $t = t_0$.

Arguments

- `E`: eccentric anomaly. This can be either a scalar or an array
- `e`: eccentricity, in the elliptic motion regime ($0 \leq e \leq 1$)

Output

The true anomaly. If an array of eccentric anomalies is provided in input, an array of the same length as `E` is returned.

Example

Plot the true anomaly as a function of mean anomaly for eccentricity $e = 0, 0.5, 0.9$. Use PyPlot.jl for plotting.

```
using PyPlot
M=linspace(0, 2pi, 1001)[1:end-1];
for ecc in (0, 0.5, 0.9)
    E = kepler_solver(M, ecc);
    plot(M, cirrange(trueanom(E, ecc), 2pi))
end
```

Notes

The eccentric anomaly can be calculated with `kepler_solver` function.

---

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.

---

xyz

`xyz` (*jd*[, *equinox*]) → 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 argument): 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.

```
jd = juldate(DateTime(1999, 1, 22))
# => 51200.5
xyz(jd + 64./86400., 2000)
# => (0.5145687092402946,-0.7696326261820777,-0.33376880143026394,0.
↪014947267514081075,0.008314838205475709,0.0036068576075747784)
```

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.

---

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.

```
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.

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).

```
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.

Miscellaneous (Non-Astronomy) Utilities

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 $[0, 2\pi)$ as if they are angles expressed in radians:

```
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 `2.0*pi` as second argument to restrict a number to the same interval.

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

---

rad2sec

rad2sec(*rad*) → seconds

Purpose

Convert from radians to seconds.

Argument

- `rad`: number of radians. It can be either a scalar or a vector.

Output

The number of seconds corresponding to `rad`. If `rad` is an array, an array of the same length is returned.

Example

```
rad2sec(1)
# => 206264.80624709636
```

Notes

Use `sec2rad` to convert seconds to radians.

---

sec2rad

`sec2rad``(`*sec*`)` → radians

Purpose

Convert from seconds to radians.

Argument

- `sec`: number of seconds. It can be either a scalar or a vector.

Output

The number of radians corresponding to `sec`. If `sec` is an array, an array of the same length is returned.

Example

```
sec2rad(3600*30)
# => 0.5235987755982988
```

Notes

Use `rad2sec` to convert radians to seconds.

CHAPTER 8

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 here.

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

- `aper`, see AperturePhotometry.jl package
- `cosmo_param`, see Cosmology.jl package
- `galage`, see Cosmology.jl package
- `glactc_pm`, see SkyCoords.jl package
- `glactc`, see SkyCoords.jl package
- `jplephinterp`, see JPLEphemeris.jl package
- `jplephread`, see JPLEphemeris.jl package
- `jplephtest`, see JPLEphemeris.jl package
- `lumdist`, see Cosmology.jl package
- `readcol`, use readdlm, part of Julia `Base.DataFmt` module. This is not a complete replacement for `readcol` but most of the time it does-the-right-thing even without using any option (it automatically identifies string and numerical columns) and you do not need to manually specify a variable for each column

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

CHAPTER 9

Indices and tables

- genindex

Index