Package 'geographiclib'

March 4, 2026

Title Access to 'GeographicLib'

Version 0.4.2

Description Bindings to the 'GeographicLib' C++ library
<https://geographiclib.sourceforge.io/> for precise geodetic calculations
including geodesic computations (distance, bearing, paths, intersections),
map projections (UTM/UPS, Transverse Mercator, Lambert Conformal Conic,
and more), grid reference systems (MGRS, Geohash, GARS, Georef),
coordinate conversions (geocentric, local Cartesian), and polygon area
on the WGS84 ellipsoid. All functions are fully vectorized.

License MIT + file LICENSE

Encoding UTF-8

RoxygenNote 7.3.3

LinkingTo cpp11

URL https://github.com/hypertidy/geographiclib

BugReports https://github.com/hypertidy/geographiclib/issues

Suggests knitr, rmarkdown, spelling, testthat (>= 3.0.0)

Config/testthat/edition 3

VignetteBuilder knitr

Language en-US

NeedsCompilation yes

Author Michael Sumner [cre, aut],
Charles Karney [cph, aut] (GeographicLib author and maintainer),
Mark Borgerding [cph] (kissfft library (BSD-3-Clause))

Maintainer Michael Sumner <mdsumner@gmail.com>

Repository CRAN

Date/Publication 2026-03-04 06:30:53 UTC

Contents

`albers_fwd` *Albers Equal Area projection*

Description

Convert geographic coordinates to/from Albers Equal Area conic projection. This is an equal-area projection commonly used for thematic maps of regions with greater east-west extent.

Usage

```
albers_fwd(
  x,
  lon0,
  stdlat = NULL,
  stdlat1 = NULL,
  stdlat2 = NULL,
  k0 = 1,
  k1 = 1
```

```
)

albers_rev(
  x,
  y,
  lon0,
  stdlat = NULL,
  stdlat1 = NULL,
  stdlat2 = NULL,
  k0 = 1,
  k1 = 1
)
```

Arguments

x	For forward conversion: a two-column matrix or data frame of coordinates (longitude, latitude) in decimal degrees. For reverse conversion: numeric vector of x (easting) coordinates in meters.
lon0	Central meridian in decimal degrees. Can be a vector to specify different central meridians for each point.
stdlat	Standard parallel for single standard parallel projections (e.g., Lambert Cylindrical Equal Area when stdlat = 0).
stdlat1, stdlat2	First and second standard parallels in decimal degrees for two standard parallel projections.
k0	Scale factor at the standard parallel(s). Default is 1.
k1	Scale factor at the first standard parallel for two standard parallel projections. Default is 1.
y	Numeric vector of y (northing) coordinates in meters (reverse only).

Details

The Albers Equal Area conic projection preserves area, making it ideal for:

- Thematic/choropleth maps where area comparison matters
- Continental-scale maps (e.g., USGS maps of CONUS)
- Statistical mapping and analysis

Common configurations:

- **CONUS**: stdlat1 = 29.5, stdlat2 = 45.5, lon0 = -96
- **Australia**: stdlat1 = -18, stdlat2 = -36, lon0 = 132
- **Europe**: stdlat1 = 43, stdlat2 = 62, lon0 = 10

Special cases:

- When stdlat1 = -stdlat2, the projection becomes Lambert Cylindrical Equal Area
- When stdlat1 = stdlat2 = 0, it becomes the cylindrical equal-area projection

The lon0 parameter is vectorized, allowing different central meridians for each point.

Value

Data frame with columns:

- For forward conversion:
  - `x`: Easting in meters
  - `y`: Northing in meters
  - `convergence`: Grid convergence in degrees
  - `scale`: Scale factor at the point
  - `lon`, `lat`: Input coordinates (echoed)
  - `lon0`: Central meridian (echoed)
- For reverse conversion:
  - `lon`: Longitude in decimal degrees
  - `lat`: Latitude in decimal degrees
  - `convergence`: Grid convergence in degrees
  - `scale`: Scale factor at the point
  - `x`, `y`: Input coordinates (echoed)
  - `lon0`: Central meridian (echoed)

See Also

`lcc_fwd()` for Lambert Conformal Conic (conformal, not equal-area).

Examples

```
# CONUS Albers Equal Area
pts <- cbind(lon = c(-122, -74, -90), lat = c(37, 41, 30))
albers_fwd(pts, lon0 = -96, stdlat1 = 29.5, stdlat2 = 45.5)

# Australia
aus <- cbind(lon = c(151.2, 115.9, 153.0), lat = c(-33.9, -32.0, -27.5))
albers_fwd(aus, lon0 = 132, stdlat1 = -18, stdlat2 = -36)

# Antarctic projection
ant <- cbind(lon = c(166.67, 77.97, -43.53), lat = c(-77.85, -67.60, -60.72))
albers_fwd(ant, lon0 = 0, stdlat1 = -72, stdlat2 = -60)

# Round-trip conversion
fwd <- albers_fwd(pts, lon0 = -96, stdlat1 = 29.5, stdlat2 = 45.5)
albers_rev(fwd$x, fwd$y, lon0 = -96, stdlat1 = 29.5, stdlat2 = 45.5)

# Single standard parallel (cylindrical-like)
albers_fwd(pts, lon0 = -96, stdlat = 37)
```

azeq_fwd *Azimuthal Equidistant projection*

Description

Convert geographic coordinates to/from Azimuthal Equidistant projection centered on a specified point. This projection preserves distances from the center point.

Usage

```
azeq_fwd(x, lon0, lat0)

azeq_rev(x, y, lon0, lat0)
```

Arguments

x	For forward conversion: a two-column matrix or data frame of coordinates (longitude, latitude) in decimal degrees. For reverse conversion: numeric vector of x coordinates in meters.
lon0	Longitude of projection center in decimal degrees. Can be a vector to specify different centers for each point.
lat0	Latitude of projection center in decimal degrees. Can be a vector to specify different centers for each point.
y	Numeric vector of y coordinates in meters (for reverse conversion).

Details

The Azimuthal Equidistant projection shows all points at their true distance and direction from the center point. It is commonly used for:

- Radio/telecommunications range maps
- Seismic wave propagation studies
- Air route distance calculations
- UN emblem (centered on North Pole)

The projection is neither conformal nor equal-area, but distances from the center are preserved exactly.

All parameters (x, lon0, lat0) are vectorized and recycled to a common length, allowing different projection centers for each point.

Value

Data frame with columns:

- For forward conversion:
  - x: Easting in meters from center

  - y: Northing in meters from center
  - azi: Azimuth from center to point (degrees)
  - scale: Scale factor at the point
  - lon, lat: Input coordinates (echoed)
  - lon0, lat0: Center coordinates (echoed)
- For reverse conversion:
  - lon: Longitude in decimal degrees
  - lat: Latitude in decimal degrees
  - azi: Azimuth from center to point (degrees)
  - scale: Scale factor at the point
  - x, y: Input coordinates (echoed)
  - lon0, lat0: Center coordinates (echoed)

Examples

```
# Project cities relative to Sydney
cities <- cbind(
  lon = c(-74, 139.7, 0),
  lat = c(40.7, 35.7, 51.5)
)
azeq_fwd(cities, lon0 = 151.2, lat0 = -33.9)

# Distance from Sydney = sqrt(x^2 + y^2)
result <- azeq_fwd(cities, lon0 = 151.2, lat0 = -33.9)
sqrt(result$x^2 + result$y^2) / 1000  # km

# Different center for each point (e.g., distance from home city)
homes <- cbind(lon = c(151.2, 139.7, -0.1), lat = c(-33.9, 35.7, 51.5))
destinations <- cbind(lon = c(-74, -74, -74), lat = c(40.7, 40.7, 40.7))
azeq_fwd(destinations, lon0 = homes[,1], lat0 = homes[,2])
```

---

cassini_fwd *Cassini-Soldner projection*

Description

Convert between geographic coordinates and the Cassini-Soldner projection. This is a transverse cylindrical equidistant projection historically used for large-scale mapping.

Usage

```
cassini_fwd(x, lon0, lat0)

cassini_rev(x, y, lon0, lat0)
```

Arguments

x	For forward conversion: a two-column matrix or data frame of coordinates (longitude, latitude) in decimal degrees. For reverse conversion: numeric vector of x (easting) coordinates in meters.
lon0	Longitude of the central meridian in decimal degrees.
lat0	Latitude of the origin in decimal degrees.
y	Numeric vector of y (northing) coordinates in meters.

Details

The Cassini-Soldner projection was historically used for large-scale topographic mapping before UTM became standard. It is still used in some countries and for historical map analysis.

Key properties:

- Distances along the central meridian are preserved
- Transverse cylindrical equidistant projection
- Not conformal (angles are not preserved)

Value

Data frame with columns:

- For forward conversion:
  - x: Easting in meters
  - y: Northing in meters
  - azi: Azimuth of the geodesic from the central point (degrees)
  - rk: Reciprocal of the azimuthal scale
  - lon, lat: Input coordinates (echoed)
- For reverse conversion:
  - lon: Longitude in decimal degrees
  - lat: Latitude in decimal degrees
  - azi: Azimuth of the geodesic from the central point (degrees)
  - rk: Reciprocal of the azimuthal scale
  - x, y: Input coordinates (echoed)

See Also

utmups_fwd() for UTM projection, lcc_fwd() for Lambert Conformal Conic

Examples

```
# Project relative to a central meridian
pts <- cbind(lon = c(-100, -99, -101), lat = c(40, 41, 39))
cassini_fwd(pts, lon0 = -100, lat0 = 40)

# Round-trip
fwd <- cassini_fwd(pts, lon0 = -100, lat0 = 40)
cassini_rev(fwd$x, fwd$y, lon0 = -100, lat0 = 40)
```

`dms_decode` *Convert between degrees and DMS (degrees, minutes, seconds) representation*

Description

Parse strings representing degrees, minutes, and seconds and return the angle in degrees. Format an angle in degrees as degrees, minutes, and seconds strings.

Usage

```
dms_decode(x)

dms_decode_latlon(dmsa, dmsb, longfirst = FALSE)

dms_decode_angle(x)

dms_decode_azimuth(x)

dms_encode(x, prec = 5L, component = NULL, indicator = "none", sep = "")

dms_split(x, seconds = FALSE)

dms_combine(d, m = 0, s = 0)
```

Arguments

`x`	Character vector of DMS strings to parse, or numeric vector of angles in degrees to encode.
`dmsa, dmsb`	Character vectors of DMS strings for latitude/longitude parsing.
`longfirst`	Logical; if TRUE, assume longitude is given before latitude when no hemisphere designators are present.
`prec`	Integer precision for output strings. For `dms_encode()` this is the number of digits after the decimal point for the trailing component. For automatic encoding, prec < 2 gives degrees, prec 2-3 gives minutes, prec >= 4 gives seconds.
`component`	Character indicating the trailing unit: `"degree"`, `"minute"`, or `"second"`.
`indicator`	Character indicating formatting: `"none"` (signed result), `"latitude"` (trailing N/S), `"longitude"` (trailing E/W), `"azimuth"` (0-360, no sign), or `"number"` (plain number).
`sep`	Character to use as DMS separator instead of d, ', ". Use `":"` for colon-separated output.
`seconds`	Logical; if TRUE, split into degrees, minutes, and seconds. If FALSE (default), split into degrees and minutes only.
`d, m, s`	Numeric vectors of degrees, minutes, and seconds.

Details

Input Formats:

The `dms_decode()` function accepts various input formats:

- Degrees, minutes, seconds: `"40d26'47\"N"`, `"40°26'47\"N"`
- Degrees and minutes: `"40d26.783'N"`, `"40:26.783N"`
- Decimal degrees: `"40.446N"`, `"-40.446"`
- Colon-separated: `"40:26:47"`, `"-74:0:21.5"`

Hemisphere designators (N, S, E, W) can appear at the beginning or end. Many Unicode symbols are supported for degrees, minutes, and seconds. See the GeographicLib DMS documentation for the full list of accepted symbols.

Precision and Components:

For `dms_encode()`, the `prec` parameter controls decimal places in the trailing component:

- `prec = 0`: whole degrees/minutes/seconds
- `prec = 1`: one decimal place
- `prec = 2`: two decimal places, etc.

For automatic component selection:

- `prec < 2`: output in degrees
- `prec = 2, 3`: output in degrees and minutes
- `prec >= 4`: output in degrees, minutes, and seconds

Value

- `dms_decode()`: Data frame with columns `angle` (degrees) and `indicator` (0=NONE, 1=LATITUDE, 2=LONGITUDE)
- `dms_decode_latlon()`: Data frame with columns `lat` and `lon` (degrees)
- `dms_decode_angle()`: Numeric vector of angles in degrees
- `dms_decode_azimuth()`: Numeric vector of azimuths in degrees (range -180 to 180)
- `dms_encode()`: Character vector of DMS strings
- `dms_split()`: Data frame with columns `d`, `m`, and optionally `s`
- `dms_combine()`: Numeric vector of angles in degrees

See Also

`geocoords_parse()` for parsing complete coordinate strings

Examples

```
# Parse DMS strings
dms_decode("40d26'47\"N")
dms_decode(c("40:26:47", "-74:0:21.5", "51d30'N"))

# Parse latitude/longitude pairs
dms_decode_latlon("40d26'47\"N", "74d0'21.5\"W")
```

```
# Parse angles (no hemisphere designator)
dms_decode_angle(c("45:30:0", "123d45'6\""))

# Parse azimuths (E/W allowed)
dms_decode_azimuth(c("45:30:0", "90W", "45E"))

# Encode to DMS strings
dms_encode(40.446, indicator = "latitude")
dms_encode(-74.006, indicator = "longitude")
dms_encode(c(40.446, -74.006), prec = 2)

# With colon separator
dms_encode(40.446, sep = ":")

# Split angle into components
dms_split(40.446)
dms_split(c(40.446, -74.006), seconds = TRUE)

# Combine components to decimal degrees
dms_combine(40, 26, 47)
dms_combine(d = c(40, -74), m = c(26, 0), s = c(47, 21.5))
```

`ellipsoid_params` *WGS84 Ellipsoid parameters and calculations*

Description

Access WGS84 ellipsoid parameters and perform ellipsoid-related calculations including auxiliary latitudes, radii of curvature, and meridian distances.

Usage

```
ellipsoid_params()

ellipsoid_circle(lat)

ellipsoid_latitudes(lat)

ellipsoid_latitudes_inv(lat, type)

ellipsoid_curvature(lat)
```

Arguments

`lat`	Numeric vector of geographic (geodetic) latitudes in decimal degrees.
`type`	Character string specifying the type of auxiliary latitude for inverse conversion. One of: "parametric", "geocentric", "rectifying", "authalic", "conformal", "isometric".

Details

The WGS84 ellipsoid is the reference surface used by GPS and most modern mapping systems. It is defined by:

- Equatorial radius: 6,378,137 m
- Flattening: 1/298.257223563

Auxiliary latitudes are different ways of measuring latitude that are useful in various map projections:

- **Parametric**: Used in ellipsoid parameterization
- **Geocentric**: Angle from center of ellipsoid
- **Rectifying**: Preserves distances along meridians
- **Authalic**: Preserves areas
- **Conformal**: Preserves angles/shapes
- **Isometric**: Used in Mercator projection

Value

- `ellipsoid_params()`: Named list with WGS84 parameters:
  - `a`: Equatorial radius (semi-major axis) in meters
  - `f`: Flattening
  - `b`: Polar radius (semi-minor axis) in meters
  - `e2`: First eccentricity squared
  - `ep2`: Second eccentricity squared
  - `n`: Third flattening
  - `area`: Surface area in square meters
  - `volume`: Volume in cubic meters
- `ellipsoid_circle()`: Data frame with columns:
  - `lat`: Input latitude
  - `radius`: Radius of the circle of latitude in meters
  - `quarter_meridian`: Distance from equator to pole along a meridian
  - `meridian_distance`: Distance from equator to the given latitude
- `ellipsoid_latitudes()`: Data frame with auxiliary latitudes:
  - `lat`: Input geographic latitude
  - `parametric`: Parametric (reduced) latitude
  - `geocentric`: Geocentric latitude
  - `rectifying`: Rectifying latitude
  - `authalic`: Authalic latitude
  - `conformal`: Conformal latitude
  - `isometric`: Isometric latitude
- `ellipsoid_latitudes_inv()`: Data frame with:
  - `input`: Input auxiliary latitude

  - `geographic`: Corresponding geographic latitude
- `ellipsoid_curvature()`: Data frame with radii of curvature:
  - `lat`: Input latitude
  - `meridional`: Meridional radius of curvature (M)
  - `transverse`: Transverse radius of curvature (N)

Examples

```
# WGS84 parameters
ellipsoid_params()

# Radius at different latitudes
ellipsoid_circle(c(0, 30, 45, 60, 90))

# Compare auxiliary latitudes
ellipsoid_latitudes(c(0, 30, 45, 60, 90))

# Radii of curvature
ellipsoid_curvature(c(0, 45, 90))
```

`gars_fwd` *Global Area Reference System (GARS)*

Description

Convert geographic coordinates (longitude/latitude) to GARS codes, or convert GARS codes back to coordinates.

Usage

```
gars_fwd(x, precision = 2L)

gars_rev(gars)
```

Arguments

`x`	A two-column matrix or data frame of coordinates (longitude, latitude) in decimal degrees, or a list with longitude and latitude components. Can also be a length-2 numeric vector for a single point.
`precision`	Integer specifying the precision level (0, 1, or 2): • 0: 30-minute cells (5-character code) • 1: 15-minute quadrants (6-character code) • 2: 5-minute keypads (7-character code, maximum precision)
`gars`	Character vector of GARS codes to convert back to coordinates.

Details

GARS (Global Area Reference System) is a standardized geospatial reference system used by the US military. It divides the Earth into cells using a hierarchical grid:

- **30-minute cells**: The base grid (720 × 360 cells globally)
- **15-minute quadrants**: Each 30-minute cell divided into 4 quadrants (1-4)
- **5-minute keypads**: Each quadrant divided into 9 keypads (1-9, like a phone keypad)

A GARS code consists of:

- 3-digit longitude band (001-720)
- 2-letter latitude band (AA-QZ)
- Optional 1-digit quadrant (1-4)
- Optional 1-digit keypad (1-9)

Example: "006AG39" = 5-minute cell at approximately (-177°, -89.5°)

Value

- `gars_fwd()`: Character vector of GARS codes.
- `gars_rev()`: Data frame with columns:
  - `lon`: Longitude of cell center in decimal degrees
  - `lat`: Latitude of cell center in decimal degrees
  - `precision`: Precision level (0, 1, or 2)
  - `lat_resolution`: Cell half-height in degrees
  - `lon_resolution`: Cell half-width in degrees

See Also

`mgrs_fwd()` for Military Grid Reference System, another military grid system.

Examples

```
# Basic conversion
gars_fwd(c(-74, 40.7))

# Different precision levels
gars_fwd(c(-74, 40.7), precision = 0)  # 30-minute
gars_fwd(c(-74, 40.7), precision = 1)  # 15-minute
gars_fwd(c(-74, 40.7), precision = 2)  # 5-minute

# Multiple points
pts <- cbind(lon = c(-74, 139.7, 0), lat = c(40.7, 35.7, 51.5))
gars_fwd(pts, precision = 2)

# Reverse conversion
gars_rev(c("213LR29", "498MH18", "361NS47"))
```

`geocentric_fwd` *Convert between geodetic and geocentric (ECEF) coordinates*

Description

Convert geographic coordinates (longitude/latitude/height) to geocentric Earth-Centered Earth-Fixed (ECEF) Cartesian coordinates (X/Y/Z), or convert ECEF coordinates back to geographic coordinates.

Usage

```
geocentric_fwd(x, h = 0)

geocentric_rev(x, y, z)
```

Arguments

`x`	For forward conversion: a two or three-column matrix or data frame of coordinates (longitude, latitude) or (longitude, latitude, height) in decimal degrees and meters. Can also be a list with lon, lat, and optionally h components. For reverse conversion: numeric vector of X coordinates in meters.
`h`	Numeric vector of heights above the ellipsoid in meters. Default is 0.
`y`	Numeric vector of Y coordinates in meters for reverse conversion.
`z`	Numeric vector of Z coordinates in meters for reverse conversion.

Details

The geocentric coordinate system (also called ECEF - Earth-Centered Earth-Fixed) is a Cartesian coordinate system with:

- Origin at the Earth's center of mass
- X-axis pointing to the intersection of the equator and prime meridian
- Y-axis pointing to the intersection of the equator and 90°E
- Z-axis pointing to the North Pole

This coordinate system is commonly used in GPS and satellite applications. All calculations use the WGS84 ellipsoid.

Value

Data frame with columns:

- For forward conversion:
  - `X`, `Y`, `Z`: Geocentric ECEF coordinates in meters
  - `lon`, `lat`, `h`: Input coordinates (echoed)
- For reverse conversion:

- `lon`: Longitude in decimal degrees
- `lat`: Latitude in decimal degrees
- `h`: Height above ellipsoid in meters
- `X`, `Y`, `Z`: Input coordinates (echoed)

See Also

`utmups_fwd()` for projected coordinates.

Examples

```
# Convert London to ECEF
geocentric_fwd(c(-0.1, 51.5))

# With height
geocentric_fwd(c(-0.1, 51.5), h = 100)

# Multiple points
pts <- cbind(lon = c(0, 90, -90), lat = c(0, 0, 0))
geocentric_fwd(pts)

# Round-trip
fwd <- geocentric_fwd(c(-0.1, 51.5, 100))
geocentric_rev(fwd$X, fwd$Y, fwd$Z)
```

`geocoords_parse` *Parse Geographic Coordinate Strings*

Description

Parse coordinate strings in various formats (MGRS, UTM/UPS, DMS, decimal) and return latitude/longitude.

Usage

```
geocoords_parse(x)
```

Arguments

`x`	Character vector of coordinate strings to parse

Details

Accepts coordinates in multiple formats:

- MGRS: `"33TWN0500049000"`
- UTM/UPS: `"33N 505000 4900000"`
- DMS: `"40d26'47\"N 74d0'21\"W"`
- Decimal: `"40.446 -74.006"`

Value

Data frame with columns:

- `lat` - Latitude in degrees
- `lon` - Longitude in degrees
- `zone` - UTM/UPS zone number
- `northp` - Logical, TRUE if in northern hemisphere
- `easting` - UTM/UPS easting in meters
- `northing` - UTM/UPS northing in meters

See Also

`mgrs_fwd()`, `mgrs_rev()`, `utmups_fwd()`, `utmups_rev()`, `dms_decode()`

Examples

```
# Parse MGRS
geocoords_parse("33TWN0500049000")

# Parse UTM
geocoords_parse("33N 505000 4900000")

# Parse DMS
geocoords_parse("40d26'47\"N 74d0'21\"W")

# Parse decimal
geocoords_parse("40.446 -74.006")

# Vectorized
geocoords_parse(c("33TWN0500049000", "40.446 -74.006"))
```

`geodesic_direct` *Geodesic calculations on the WGS84 ellipsoid*

Description

Solve geodesic problems on the WGS84 ellipsoid using exact algorithms. These functions provide high-precision solutions for:

- Finding destination points given start, azimuth, and distance (direct problem)
- Finding distance and azimuths between two points (inverse problem)
- Generating points along geodesic paths
- Computing distance matrices

Usage

```
geodesic_direct(x, azi, s)

geodesic_inverse(x, y)

geodesic_path(x, y, n = 100L)

geodesic_line(x, azi, distances)

geodesic_distance(x, y)

geodesic_distance_matrix(x, y = NULL)
```

Arguments

x	A two-column matrix or data frame of starting coordinates (longitude, latitude) in decimal degrees.
azi	Numeric vector of azimuths (bearings) in degrees, measured clockwise from north.
s	Numeric vector of distances in meters.
y	A two-column matrix or data frame of ending coordinates (longitude, latitude) in decimal degrees.
n	Integer number of points to generate along the path (including start and end points).
distances	Numeric vector of distances from the starting point in meters.

Details

These functions use the exact geodesic algorithms from GeographicLib, which provide full double-precision accuracy for all points on the WGS84 ellipsoid.

The **direct problem** finds the destination given a starting point, initial azimuth (bearing), and distance. This is useful for navigation and creating buffers.

The **inverse problem** finds the shortest path (geodesic) between two points and returns the distance and azimuths at both endpoints.

The azimuth is measured in degrees from north, with positive values clockwise (east) and negative values counter-clockwise (west). The range is -180° to 180° (e.g., 90° = east, -90° = west, 180° or -180° = south).

Value

- `geodesic_direct()`: Data frame with columns:
  - `lon1`, `lat1`: Starting coordinates
  - `azi1`: Starting azimuth (degrees)
  - `s12`: Distance (meters)
  - `lon2`, `lat2`: Destination coordinates

  - azi2: Azimuth at destination (degrees)
  - m12: Reduced length (meters)
  - M12, M21: Geodesic scale factors
  - S12: Area under geodesic (square meters)
- geodesic_inverse(): Data frame with columns:
  - lon1, lat1: Starting coordinates
  - lon2, lat2: Ending coordinates
  - s12: Distance (meters)
  - azi1: Azimuth at start (degrees)
  - azi2: Azimuth at end (degrees)
  - m12: Reduced length (meters)
  - M12, M21: Geodesic scale factors
  - S12: Area under geodesic (square meters)
- geodesic_path(): Data frame with columns:
  - lon, lat: Coordinates along the path
  - azi: Azimuth at each point (degrees)
  - s: Distance from start (meters)
- geodesic_line(): Data frame with columns:
  - lon, lat: Coordinates at specified distances
  - azi: Azimuth at each point (degrees)
  - s: Distance from start (meters)
- geodesic_distance(): Numeric vector of distances in meters (pairwise).
- geodesic_distance_matrix(): Matrix of distances in meters.

Examples

```
# Direct problem: Where do you end up starting from London,
# heading east for 1000 km?
geodesic_direct(c(-0.1, 51.5), azi = 90, s = 1000000)

# Inverse problem: Distance from London to New York
geodesic_inverse(c(-0.1, 51.5), c(-74, 40.7))

# Generate a great circle path
path <- geodesic_path(c(-0.1, 51.5), c(-74, 40.7), n = 100)
head(path)

# Multiple distances along a bearing
geodesic_line(c(-0.1, 51.5), azi = 45, distances = c(100, 500, 1000) * 1000)
```

`geodesic_direct_fast` *Fast geodesic calculations (series approximation)*

Description

These functions provide the same geodesic calculations as `geodesic_direct()`, `geodesic_inverse()`, etc., but use a series approximation that is slightly faster at the cost of reduced precision (accurate to ~15 nanometers vs full double precision for the exact versions).

For most applications, the difference is negligible and these faster versions are recommended.

Usage

```
geodesic_direct_fast(x, azi, s)

geodesic_inverse_fast(x, y)

geodesic_path_fast(x, y, n = 100L)

geodesic_distance_fast(x, y)

geodesic_distance_matrix_fast(x, y = NULL)
```

Arguments

`x`	A two-column matrix or data frame of starting coordinates (longitude, latitude) in decimal degrees.
`azi`	Numeric vector of azimuths (bearings) in degrees, measured clockwise from north.
`s`	Numeric vector of distances in meters.
`y`	A two-column matrix or data frame of ending coordinates (longitude, latitude) in decimal degrees.
`n`	Integer number of points to generate along the path (including start and end points).

Value

Same as the corresponding exact geodesic functions.

See Also

`geodesic_direct()`, `geodesic_inverse()` for exact versions

Examples

```
# Fast inverse: London to New York
geodesic_inverse_fast(c(-0.1, 51.5), c(-74, 40.7))

# Compare to exact version
geodesic_inverse(c(-0.1, 51.5), c(-74, 40.7))$s12
geodesic_inverse_fast(c(-0.1, 51.5), c(-74, 40.7))$s12
```

---

geodesic_intersect *Geodesic intersections*

Description

Find the intersection of two geodesics on the WGS84 ellipsoid. Several methods are available:

- `geodesic_intersect()` - Find the closest intersection of two geodesics defined by starting points and azimuths
- `geodesic_intersect_segment()` - Find the intersection of two geodesic segments defined by their endpoints
- `geodesic_intersect_next()` - Find the next closest intersection from a known intersection point
- `geodesic_intersect_all()` - Find all intersections within a maximum distance

Usage

```
geodesic_intersect(x, azi_x, y, azi_y)

geodesic_intersect_segment(x1, x2, y1, y2)

geodesic_intersect_next(x, azi_x, azi_y)

geodesic_intersect_all(x, azi_x, y, azi_y, maxdist)
```

Arguments

`x`	Coordinates for geodesic X: a vector of `c(lon, lat)`, a matrix with columns `[lon, lat]`, or a list with `lon` and `lat` components.
`azi_x`	Azimuth(s) for geodesic X in degrees.
`y`	Coordinates for geodesic Y (same format as `x`).
`azi_y`	Azimuth(s) for geodesic Y in degrees.
`x1`, `x2`	Start and end coordinates for segment X.
`y1`, `y2`	Start and end coordinates for segment Y.
`maxdist`	Maximum distance (in meters) for finding all intersections.

Details

The intersection point is found using the algorithm described in:

C. F. F. Karney, "Geodesic intersections", J. Surveying Eng. 150(3), 04024005:1-9 (2024). doi:10.1061/JSUED2.SUENG1483

The "closest" intersection minimizes the L1 distance, defined as `|x| + |y|` where `x` and `y` are the displacements along the two geodesics.

For segment intersection, `segmode` encodes whether the intersection lies within the segments:

- `segmode = 0` means the intersection lies within both segments
- Non-zero values indicate the intersection lies outside one or both segments

The coincidence indicator is useful for detecting when geodesics are parallel or antiparallel at their intersection.

Value

A data frame with columns:

- `x` - Displacement along geodesic X from its starting point (meters)
- `y` - Displacement along geodesic Y from its starting point (meters)
- `coincidence` - Indicator: 0 = normal intersection, +1 = geodesics are parallel and coincident, -1 = geodesics are antiparallel and coincident
- `lat` - Latitude of intersection point (degrees)
- `lon` - Longitude of intersection point (degrees)

For `geodesic_intersect_segment()`, an additional column `segmode` indicates whether the intersection lies within both segments (0), or which segment(s) the intersection lies outside of.

For `geodesic_intersect_all()`, returns a list of data frames (one per input pair of geodesics).

See Also

`geodesic_inverse()` for computing azimuths between points

Examples

```
# Two geodesics from different starting points
# Geodesic X: starts at (0, 0), azimuth 45 degrees
# Geodesic Y: starts at (1, 0), azimuth 315 degrees
geodesic_intersect(c(0, 0), 45, c(1, 0), 315)

# Vectorized: multiple pairs of geodesics
geodesic_intersect(
  cbind(c(0, 0, 0), c(0, 0, 0)),
  c(30, 45, 60),
  cbind(c(1, 1, 1), c(0, 0, 0)),
  c(330, 315, 300)
)
# Intersection of two geodesic segments
```

```
# Segment X: (0, -1) to (0, 1)
# Segment Y: (-1, 0) to (1, 0)
geodesic_intersect_segment(
  c(0, -1), c(0, 1),
  c(-1, 0), c(1, 0)
)
# Find the next intersection from a known intersection point
# Two geodesics crossing at (0, 0) with azimuths 45 and 315
geodesic_intersect_next(c(0, 0), 45, 315)
# Find all intersections within 1,000,000 meters
geodesic_intersect_all(c(0, 0), 45, c(1, 0), 315, maxdist = 1e6)
```

`geodesic_nn` *Nearest Neighbor Search Using Geodesic Distance*

Description

Find nearest neighbors on the WGS84 ellipsoid using geodesic distance. These functions build an efficient vantage-point tree index for fast repeated queries.

Usage

```
geodesic_nn(dataset, query, k = 1L)

geodesic_nn_radius(dataset, query, radius)
```

Arguments

`dataset`	A matrix or vector of coordinates (lon, lat) for the dataset points. For a matrix, each row is a point. For a vector, it should be `c(lon, lat)` for a single point.
`query`	A matrix or vector of coordinates (lon, lat) for the query points. Same format as `dataset`.
`k`	Integer. The number of nearest neighbors to find.
`radius`	Numeric. The search radius in meters.

Details

These functions use the GeographicLib NearestNeighbor class, which implements a vantage-point tree optimized for geodesic distance calculations on the WGS84 ellipsoid.

The vantage-point tree provides O(log n) search complexity after O(n log n) construction time. For repeated queries against the same dataset, this is much more efficient than computing all pairwise distances.

Distances are computed using the exact geodesic inverse formula, not approximations like Haversine or Vincenty.

Value

For `geodesic_nn()`: A list with two matrices:

- `index`: Integer matrix (k x n_queries) of 1-based indices into `dataset`
- `distance`: Numeric matrix (k x n_queries) of geodesic distances in meters

For `geodesic_nn_radius()`: A list of data frames, one per query point, each containing:

- `index`: Integer vector of 1-based indices into `dataset`
- `distance`: Numeric vector of geodesic distances in meters

Examples

```
# Create a dataset of cities
cities <- cbind(
  lon = c(151.21, 144.96, 153.03, 115.86, 138.60),
  lat = c(-33.87, -37.81, -27.47, -31.95, -34.93)
)
rownames(cities) <- c("Sydney", "Melbourne", "Brisbane", "Perth", "Adelaide")

# Find 2 nearest neighbors for each city (including itself)
result <- geodesic_nn(cities, cities, k = 2)
result$index
result$distance

# Query points not in the dataset
queries <- cbind(
  lon = c(149.13, 147.32),
  lat = c(-35.28, -42.88)
)
rownames(queries) <- c("Canberra", "Hobart")

geodesic_nn(cities, queries, k = 3)

# Find all cities within 1000 km
geodesic_nn_radius(cities, queries, radius = 1e6)
```

---

geohash_fwd *Convert coordinates to/from Geohash*

Description

Convert geographic coordinates (longitude/latitude) to Geohash strings, or convert Geohash strings back to coordinates.

Usage

```
geohash_fwd(x, len = 12L)

geohash_rev(geohash)

geohash_resolution(len)

geohash_length(resolution = NULL, lat_resolution = NULL, lon_resolution = NULL)
```

Arguments

x	A two-column matrix or data frame of coordinates (longitude, latitude) in decimal degrees, or a list with longitude and latitude components. Can also be a length-2 numeric vector for a single point.
len	Integer specifying the length of the Geohash string (1-18). Default is 12, which gives approximately 19mm precision. Can be a vector to specify different lengths for each point.
geohash	Character vector of Geohash strings to convert back to coordinates.
resolution	Numeric. Desired resolution in degrees for geohash_length().
lat_resolution	Numeric. Desired latitude resolution in degrees.
lon_resolution	Numeric. Desired longitude resolution in degrees.

Details

Geohash is a geocoding system that encodes geographic coordinates into a short string of letters and digits. It has a useful property: truncating a Geohash reduces precision but the truncated code still refers to a location containing the original point.

The Geohash length determines precision:

- Length 1: ~5000 km
- Length 4: ~20 km
- Length 6: ~610 m
- Length 8: ~19 m
- Length 10: ~0.6 m
- Length 12: ~19 mm (default)
- Length 18: ~0.0001 mm (maximum)

Both geohash_fwd() and geohash_rev() are fully vectorized.

Value

- geohash_fwd(): Character vector of Geohash strings.
- geohash_rev(): Data frame with columns:
  - lon: Longitude in decimal degrees (center of cell)
  - lat: Latitude in decimal degrees (center of cell)

  - `len`: Length of the Geohash string
  - `lat_resolution`: Latitude resolution in degrees (half-height of cell)
  - `lon_resolution`: Longitude resolution in degrees (half-width of cell)
- `geohash_resolution()`: Data frame with columns:
  - `len`: Geohash length
  - `lat_resolution`: Latitude resolution in degrees
  - `lon_resolution`: Longitude resolution in degrees
- `geohash_length()`: Integer, minimum Geohash length to achieve the specified resolution.

See Also

`mgrs_fwd()` for Military Grid Reference System encoding, which provides a different grid-based coordinate system.

Examples

```
# Single point conversion
(gh <- geohash_fwd(c(147.325, -42.881)))
geohash_rev(gh)

# Multiple points with varying precision
pts <- cbind(
  lon = c(147, -74, 0),
  lat = c(-42, 40.7, 51.5)
)
geohash_fwd(pts, len = c(6, 8, 12))

# Truncation preserves containment
gh <- geohash_fwd(c(147.325, -42.881), len = 12)
substr(gh, 1, 6)  # Lower precision, but still contains original point

# Resolution for different lengths
geohash_resolution(1:12)

# Find length needed for ~1km precision
geohash_length(1/111)  # ~1 degree / 111 km
```

---

georef_fwd *World Geographic Reference System (Georef)*

Description

Convert geographic coordinates (longitude/latitude) to World Geographic Reference System (Georef) codes, or convert Georef codes back to coordinates.

Usage

```
georef_fwd(x, precision = 2L)

georef_rev(georef)
```

Arguments

x	A two-column matrix or data frame of coordinates (longitude, latitude) in decimal degrees, or a list with longitude and latitude components. Can also be a length-2 numeric vector for a single point.
precision	Integer specifying the precision level (-1 to 11): • -1: 15-degree cells (2-character code) • 0: 1-degree cells (4-character code) • 1: 1-minute cells (6-character code) • 2: 0.1-minute cells (8-character code) • Higher values give progressively finer precision
georef	Character vector of Georef codes to convert back to coordinates.

Details

The World Geographic Reference System (Georef) is a grid-based geocode system used primarily for air navigation. It was developed by the US and adopted by ICAO (International Civil Aviation Organization).

The Georef code structure:

- First letter: 15° longitude band (A-Z, omitting I and O)
- Second letter: 15° latitude band (A-M, omitting I)
- Third letter: 1° longitude within band (A-Q, omitting I and O)
- Fourth letter: 1° latitude within band (A-Q, omitting I and O)
- Remaining digits: minutes (and fractions) of longitude and latitude

Example: "GJPJ3217" represents approximately (0.54°, 51.28°)

Value

- `georef_fwd()`: Character vector of Georef codes.
- `georef_rev()`: Data frame with columns:
  - `lon`: Longitude of cell center in decimal degrees
  - `lat`: Latitude of cell center in decimal degrees
  - `precision`: Precision level
  - `lat_resolution`: Cell half-height in degrees
  - `lon_resolution`: Cell half-width in degrees

See Also

`gars_fwd()` for Global Area Reference System, `mgrs_fwd()` for Military Grid Reference System.

Examples

```
# Basic conversion
georef_fwd(c(-0.1, 51.5))

# Different precision levels
georef_fwd(c(-0.1, 51.5), precision = -1)  # 15-degree
georef_fwd(c(-0.1, 51.5), precision = 0)   # 1-degree
georef_fwd(c(-0.1, 51.5), precision = 1)   # 1-minute
georef_fwd(c(-0.1, 51.5), precision = 2)   # 0.1-minute

# Multiple points
pts <- cbind(lon = c(-74, 139.7, 0), lat = c(40.7, 35.7, 51.5))
georef_fwd(pts)

# Reverse conversion
georef_rev(c("GJPJ3217", "SKNA2342", "FJBL0630"))
```

---

gnomonic_fwd *Gnomonic projection*

Description

Convert between geographic coordinates and the gnomonic projection. In this projection, geodesics (shortest paths) appear as straight lines, making it useful for navigation and great circle route planning.

Usage

```
gnomonic_fwd(x, lon0, lat0)

gnomonic_rev(x, y, lon0, lat0)
```

Arguments

x	For forward conversion: a two-column matrix or data frame of coordinates (longitude, latitude) in decimal degrees. For reverse conversion: numeric vector of x coordinates in meters.
lon0	Longitude of the projection center in decimal degrees.
lat0	Latitude of the projection center in decimal degrees.
y	Numeric vector of y coordinates in meters.

Details

The gnomonic projection has a unique property: all geodesics (great circles on a sphere, shortest paths on an ellipsoid) appear as straight lines. This makes it invaluable for:

- Planning great circle routes in aviation and shipping

- Seismic ray path analysis
- Radio wave propagation studies

Limitations:

- Can only show less than a hemisphere
- Extreme distortion away from the center
- Neither conformal nor equal-area

Value

Data frame with columns:

- For forward conversion:
  - `x`: X coordinate in meters
  - `y`: Y coordinate in meters
  - `azi`: Azimuth of the geodesic at the center (degrees)
  - `rk`: Reciprocal of the azimuthal scale
  - `lon`, `lat`: Input coordinates (echoed)
- For reverse conversion:
  - `lon`: Longitude in decimal degrees
  - `lat`: Latitude in decimal degrees
  - `azi`: Azimuth of the geodesic at the center (degrees)
  - `rk`: Reciprocal of the azimuthal scale
  - `x`, `y`: Input coordinates (echoed)

See Also

`azeq_fwd()` for azimuthal equidistant projection

Examples

```
# Project cities relative to London
cities <- cbind(
  lon = c(-74, 139.7, 151.2, 2.3),
  lat = c(40.7, 35.7, -33.9, 48.9)
)
gnomonic_fwd(cities, lon0 = -0.1, lat0 = 51.5)

# Great circle route appears as straight line
# London to NYC path
path <- geodesic_path(c(-0.1, 51.5), c(-74, 40.7), n = 10)
projected <- gnomonic_fwd(cbind(path$lon, path$lat), lon0 = -37, lat0 = 46)
# x and y should be approximately linear
plot(projected$x, projected$y, type = "l")
```

lcc_fwd *Lambert Conformal Conic projection*

Description

Convert geographic coordinates (longitude/latitude) to Lambert Conformal Conic (LCC) projected coordinates, or convert projected coordinates back to geographic coordinates.

Usage

```
lcc_fwd(
  x,
  lon0,
  lat0 = NULL,
  stdlat = NULL,
  stdlat1 = NULL,
  stdlat2 = NULL,
  k0 = 1,
  k1 = 1
)

lcc_rev(
  x,
  y,
  lon0,
  lat0 = NULL,
  stdlat = NULL,
  stdlat1 = NULL,
  stdlat2 = NULL,
  k0 = 1,
  k1 = 1
)
```

Arguments

x	For forward conversion: a two-column matrix or data frame of coordinates (longitude, latitude) in decimal degrees, or a list with longitude and latitude components. Can also be a length-2 numeric vector for a single point. For reverse conversion: numeric vector of easting values (x coordinates) in meters.
lon0	Central meridian (longitude of origin) in decimal degrees.
lat0	Latitude of origin in decimal degrees (used for documentation, not in the projection calculation itself).
stdlat	Standard parallel in decimal degrees for single standard parallel (tangent cone) projections.
stdlat1, stdlat2	First and second standard parallels in decimal degrees for two standard parallel (secant cone) projections.

k0	Scale factor at the standard parallel. Default is 1.
k1	Scale factor at the first standard parallel for two standard parallel projections. Default is 1.
y	Numeric vector of northing values (y coordinates) in meters for reverse conversion.

Details

The Lambert Conformal Conic projection is a conic map projection commonly used for aeronautical charts, state plane coordinate systems, and many national/regional coordinate systems.

Two forms are supported:

- **Single standard parallel** (tangent cone): The cone is tangent to the ellipsoid at one latitude. Use `lcc_fwd()` and `lcc_rev()` with `stdlat`.
- **Two standard parallels** (secant cone): The cone intersects the ellipsoid at two latitudes. Use `lcc_fwd()` and `lcc_rev()` with `stdlat1` and `stdlat2`.

The projection is conformal (preserves local angles/shapes) and is best suited for mid-latitude regions with greater east-west extent.

All functions use the WGS84 ellipsoid and are fully vectorized on coordinate inputs.

Value

Data frame with columns:

- For forward conversion:
  - `x`: Easting in meters
  - `y`: Northing in meters
  - `convergence`: Meridian convergence in degrees
  - `scale`: Scale factor at the point
  - `lon`: Longitude (echoed from input)
  - `lat`: Latitude (echoed from input)
- For reverse conversion:
  - `lon`: Longitude in decimal degrees
  - `lat`: Latitude in decimal degrees
  - `convergence`: Meridian convergence in degrees
  - `scale`: Scale factor at the point
  - `x`: Easting (echoed from input)
  - `y`: Northing (echoed from input)

See Also

`utmups_fwd()` for UTM/UPS projections which are also conformal.

Examples

```
# Single standard parallel (e.g., for a state plane zone)
pts <- cbind(lon = c(-100, -99, -98), lat = c(40, 41, 42))
lcc_fwd(pts, lon0 = -100, stdlat = 40)

# Two standard parallels (e.g., for continental US)
# CONUS Albers-like setup
lcc_fwd(pts, lon0 = -96, stdlat1 = 33, stdlat2 = 45)

# Round-trip conversion
fwd <- lcc_fwd(pts, lon0 = -100, stdlat = 40)
lcc_rev(fwd$x, fwd$y, lon0 = -100, stdlat = 40)
```

---

localcartesian_fwd *Local Cartesian (ENU) coordinate system*

Description

Convert between geographic coordinates and a local Cartesian coordinate system centered at a specified origin. The local system uses East-North-Up (ENU) axes.

Usage

```
localcartesian_fwd(x, lon0, lat0, h = 0, h0 = 0)

localcartesian_rev(x, y, z, lon0, lat0, h0 = 0)
```

Arguments

x	For forward conversion: a two-column matrix or data frame of coordinates (longitude, latitude) in decimal degrees, or a list with longitude and latitude components. Can also be a length-2 numeric vector for a single point. For reverse conversion: numeric vector of x (east) coordinates in meters.
lon0	Longitude of the origin in decimal degrees.
lat0	Latitude of the origin in decimal degrees.
h	Numeric vector of heights above the ellipsoid in meters. Default is 0.
h0	Height of the origin above the ellipsoid in meters. Default is 0.
y	Numeric vector of y (north) coordinates in meters.
z	Numeric vector of z (up) coordinates in meters.

Details

The local Cartesian coordinate system is useful for:

- Local surveys where a flat Earth approximation is valid
- Converting GPS positions to a local reference frame

- Robotics and navigation applications

The coordinate system is:

- **x**: positive east
- **y**: positive north
- **z**: positive up (away from Earth's center)

This is also known as an ENU (East-North-Up) coordinate system.

Value

- `localcartesian_fwd()`: Data frame with columns:
  - `x`: East coordinate in meters
  - `y`: North coordinate in meters
  - `z`: Up coordinate in meters
  - `lon`, `lat`, `h`: Input coordinates (echoed)
- `localcartesian_rev()`: Data frame with columns:
  - `lon`: Longitude in decimal degrees
  - `lat`: Latitude in decimal degrees
  - `h`: Height above ellipsoid in meters
  - `x`, `y`, `z`: Input coordinates (echoed)

See Also

`geocentric_fwd()` for Earth-Centered Earth-Fixed (ECEF) coordinates

Examples

```
# Set up local system centered on London
london <- c(-0.1, 51.5)

# Convert nearby points to local coordinates
pts <- cbind(
  lon = c(-0.1, -0.2, 0.0),
  lat = c(51.5, 51.6, 51.4)
)
localcartesian_fwd(pts, lon0 = london[1], lat0 = london[2])

# Round-trip conversion
fwd <- localcartesian_fwd(pts, lon0 = -0.1, lat0 = 51.5)
localcartesian_rev(fwd$x, fwd$y, fwd$z, lon0 = -0.1, lat0 = 51.5)
```

mgrs_fwd *Convert coordinates to/from Military Grid Reference System (MGRS)*

Description

Convert geographic coordinates (longitude/latitude) to MGRS grid reference strings, or convert MGRS strings back to coordinates.

Usage

```
mgrs_fwd(x, precision = 5L)

mgrs_rev(code)
```

Arguments

x	A two-column matrix or data frame of coordinates (longitude, latitude) in decimal degrees, or a list with longitude and latitude components. Can also be a length-2 numeric vector for a single point.
precision	Integer between 0 and 5 (default 5) specifying the precision of the MGRS grid reference: • 0: 100 km precision • 1: 10 km precision • 2: 1 km precision • 3: 100 m precision • 4: 10 m precision • 5: 1 m precision (full precision) Can be a vector to specify different precisions for each point.
code	Character vector of MGRS grid reference strings to convert back to coordinates.

Details

The Military Grid Reference System (MGRS) is a geocoordinate standard used by NATO militaries for locating points on Earth. It is an alternative to latitude/longitude that uses a hierarchical grid system.

Both functions are fully vectorized. Missing values (NA) are not currently supported.

For polar regions (latitude > 84°N or < 80°S), the Universal Polar Stereographic (UPS) system is used instead of UTM, indicated by zone = 0.

Value

- `mgrs_fwd()`: Character vector of MGRS grid reference strings
- `mgrs_rev()`: Data frame with columns:
  - `lon`: Longitude in decimal degrees

- `lat`: Latitude in decimal degrees
- `x`: Easting in meters (UTM/UPS projection)
- `y`: Northing in meters (UTM/UPS projection)
- `zone`: UTM zone number (0 for polar UPS regions)
- `northp`: Logical, TRUE for northern hemisphere, FALSE for southern
- `precision`: Integer precision level (0-5) encoded in the MGRS string
- `convergence`: Meridian convergence in degrees (angle between true north and grid north)
- `scale`: Scale factor at the point (dimensionless, typically near 1.0)
- `grid_zone`: Grid zone designator (e.g., "51P", "04L")
- `square_100km`: 100km square identifier (e.g., "SM", "GH")
- `crs`: EPSG code string for the appropriate UTM/UPS projection (e.g., "EPSG:32755" for UTM zone 55S, "EPSG:32661" for UPS North)

Examples

```
# Single point conversion
(code <- mgrs_fwd(cbind(147.325, -42.881)))
mgrs_rev(code)

# Multiple points with varying precision
x <- cbind(lon = c(-63.22, 34.02, 49.45, 45.67, 47.4),
           lat = c(17.62, -1.9, 37.47, 39.84, 33.15))
codes <- mgrs_fwd(x, precision = c(5, 4, 3, 2, 1))
codes

# Reverse conversion returns detailed coordinate information
result <- mgrs_rev(codes)
result

# Polar regions use UPS (zone 0)
polar_codes <- mgrs_fwd(cbind(c(147, -100), c(88, -88)))
mgrs_rev(polar_codes)
```

---

osgb_fwd *Ordnance Survey National Grid (Great Britain)*

Description

Convert between geographic coordinates and the Ordnance Survey National Grid used in Great Britain.

Important: These functions expect coordinates on the OSGB36 datum, not WGS84. For WGS84 coordinates (e.g., from GPS), you need to perform a datum transformation first using another package such as sf.

Usage

```
osgb_fwd(x)

osgb_rev(easting, northing)

osgb_gridref(x, precision = 2L)

osgb_gridref_rev(gridref)
```

Arguments

x	For `osgb_fwd()` and `osgb_gridref()`: a two-column matrix or data frame of OSGB36 coordinates (longitude, latitude) in decimal degrees.
easting	Numeric vector of OSGB eastings in meters.
northing	Numeric vector of OSGB northings in meters.
precision	Integer specifying the precision of grid references: • -1: 500 km squares (first letter only) • 0: 100 km squares (two letters) • 1: 10 km (2 digits) • 2: 1 km (4 digits) • 3: 100 m (6 digits) • 4: 10 m (8 digits) • 5: 1 m (10 digits)
gridref	Character vector of OSGB grid reference strings.

Details

The Ordnance Survey National Grid is a geographic grid reference system used in Great Britain. It uses the OSGB36 datum and a Transverse Mercator projection.

Grid references are alphanumeric codes like "TQ3080" for central London. The format is two letters (100 km square) followed by an even number of digits.

Datum note: The difference between WGS84 and OSGB36 can be up to ~100m. For precise work, transform WGS84 coordinates to OSGB36 first.

Value

- `osgb_fwd()`: Data frame with columns:
  - `easting`: OSGB easting in meters
  - `northing`: OSGB northing in meters
  - `convergence`: Grid convergence in degrees
  - `scale`: Scale factor
  - `lon`, `lat`: Input OSGB36 coordinates (echoed)
- `osgb_rev()`: Data frame with columns:
  - `lon`: OSGB36 longitude in decimal degrees

  - `lat`: OSGB36 latitude in decimal degrees
  - `convergence`: Grid convergence in degrees
  - `scale`: Scale factor
  - `easting`, `northing`: Input coordinates (echoed)
- `osgb_gridref()`: Character vector of grid reference strings.
- `osgb_gridref_rev()`: Data frame with columns:
  - `lon`: OSGB36 longitude in decimal degrees
  - `lat`: OSGB36 latitude in decimal degrees
  - `easting`: OSGB easting in meters
  - `northing`: OSGB northing in meters
  - `precision`: Precision level of the grid reference

Examples

```
# OSGB36 coordinates for central London (not WGS84!)
# In practice, you would transform from WGS84 first
london_osgb36 <- c(-0.1270, 51.5072)

# Convert to OSGB grid
osgb_fwd(london_osgb36)

# Get grid reference at various precisions
osgb_gridref(london_osgb36, precision = 2)  # 1 km
osgb_gridref(london_osgb36, precision = 3)  # 100 m
osgb_gridref(london_osgb36, precision = 4)  # 10 m

# Parse a grid reference
osgb_gridref_rev("TQ3080")

# Round-trip conversion
fwd <- osgb_fwd(london_osgb36)
osgb_rev(fwd$easting, fwd$northing)
```

---

polarstereo_fwd *Polar Stereographic projection*

Description

Convert geographic coordinates to/from Polar Stereographic projection. This conformal projection is used for polar regions and is the basis for the Universal Polar Stereographic (UPS) system.

Usage

```
polarstereo_fwd(x, northp, k0 = 0.994)

polarstereo_rev(x, y, northp, k0 = 0.994)
```

Arguments

x	For forward conversion: a two-column matrix or data frame of coordinates (longitude, latitude) in decimal degrees. For reverse conversion: numeric vector of x (easting) coordinates in meters.
northp	Logical indicating hemisphere: TRUE for north polar, FALSE for south polar. Can be a vector for different hemispheres per point.
k0	Scale factor at the pole. Default is 0.994 (UPS standard). Use k0 = 1 for true stereographic.
y	Numeric vector of y (northing) coordinates in meters (reverse only).

Details

The Polar Stereographic projection is a conformal azimuthal projection centered on either pole. It is ideal for mapping polar regions because:

- It preserves local angles and shapes
- Directions from the pole are true
- Scale distortion is minimal near the pole

UPS (Universal Polar Stereographic) The default k0 = 0.994 corresponds to the UPS system used by:

- NATO military mapping
- EPSG:32661 (UPS North) and EPSG:32761 (UPS South)
- High-latitude extensions of UTM

UPS is used for latitudes poleward of 84°N and 80°S.

Common scale factors:

- k0 = 0.994: UPS standard
- k0 = 1.0: True stereographic (scale = 1 at pole)
- k0 = 0.97276901289: NSIDC Sea Ice Polar Stereographic

Value

Data frame with columns:

- For forward conversion:
  - x: Easting in meters from pole
  - y: Northing in meters from pole
  - convergence: Grid convergence in degrees
  - scale: Scale factor at the point
  - lon, lat: Input coordinates (echoed)
  - northp: Hemisphere indicator (echoed)
- For reverse conversion:
  - lon: Longitude in decimal degrees

- `lat`: Latitude in decimal degrees
- `convergence`: Grid convergence in degrees
- `scale`: Scale factor at the point
- `x`, `y`: Input coordinates (echoed)
- `northp`: Hemisphere indicator (echoed)

See Also

`utmups_fwd()` for automatic UTM/UPS selection based on latitude.

Examples

```
# Antarctic stations
stations <- cbind(
  lon = c(166.67, 77.97, -43.53, 0),
  lat = c(-77.85, -67.60, -60.72, -90)
)
polarstereo_fwd(stations, northp = FALSE)

# Arctic points
arctic <- cbind(lon = c(0, 90, 180, -90), lat = c(85, 85, 85, 85))
polarstereo_fwd(arctic, northp = TRUE)

# True stereographic (k0 = 1)
polarstereo_fwd(stations, northp = FALSE, k0 = 1.0)

# NSIDC Sea Ice projection
polarstereo_fwd(stations, northp = FALSE, k0 = 0.97276901289)

# South Pole is at origin
sp <- polarstereo_fwd(c(0, -90), northp = FALSE)
sp$x  # 0
sp$y  # 0

# Round-trip conversion
fwd <- polarstereo_fwd(stations, northp = FALSE)
polarstereo_rev(fwd$x, fwd$y, northp = FALSE)
```

---

polygon_area *Compute geodesic polygon area and perimeter*

Description

Compute the area and perimeter of a polygon on the WGS84 ellipsoid using geodesic calculations. This gives accurate results for polygons of any size, including those spanning large portions of the globe.

Usage

```
polygon_area(x, id = NULL, polyline = FALSE)
```

Arguments

x	A two-column matrix or data frame of coordinates (longitude, latitude) in decimal degrees defining polygon vertices, or a list with longitude and latitude components.
id	Optional integer vector identifying separate polygons. Points with the same id are treated as vertices of the same polygon. If NULL (default), all points are treated as a single polygon.
polyline	Logical. If FALSE (default), compute area and perimeter of a closed polygon. If TRUE, compute only the length of a polyline (area will be meaningless).

Details

The polygon area is computed using the geodesic method which accounts for the ellipsoidal shape of the Earth. This is more accurate than spherical approximations, especially for large polygons.

The area is signed: counter-clockwise polygons have positive area, clockwise polygons have negative area. The absolute value gives the actual area.

For very large polygons (more than half the Earth's surface), the sign convention may seem counterintuitive - the "inside" is the smaller region.

The computation uses the WGS84 ellipsoid (the same as GPS).

Value

- For a single polygon (id = NULL): A list with components:
  - area: Signed area in square meters. Positive for counter-clockwise polygons, negative for clockwise.
  - perimeter: Perimeter in meters.
  - n: Number of vertices.
- For multiple polygons (id specified): A data frame with columns:
  - id: Polygon identifier
  - area: Signed area in square meters
  - perimeter: Perimeter in meters
  - n: Number of vertices

Examples

```
# Triangle: London - New York - Rio de Janeiro
pts <- cbind(
  lon = c(0, -74, -43),
  lat = c(52, 41, -23)
)
polygon_area(pts)

# Multiple polygons using id
pts <- cbind(
  lon = c(0, -74, -43, 100, 110, 105),
  lat = c(52, 41, -23, 10, 10, 20)
```

```
)
polygon_area(pts, id = c(1, 1, 1, 2, 2, 2))

# Polyline length (not a closed polygon)
polygon_area(pts[1:3, ], polyline = TRUE)

# Area of Australia (approximate boundary)
australia <- cbind(
  lon = c(113, 153, 153, 142, 129, 113),
  lat = c(-26, -26, -10, -10, -15, -26)
)
result <- polygon_area(australia)
# Area in square kilometers
abs(result$area) / 1e6
```

`polygon_area_cumulative`

Compute cumulative polygon area and perimeter

Description

Compute the area and perimeter of a polygon at each vertex, showing how the measurements accumulate as vertices are added.

Usage

```
polygon_area_cumulative(x, polyline = FALSE)
```

Arguments

`x`	A two-column matrix or data frame of coordinates (longitude, latitude) in decimal degrees defining polygon vertices, or a list with longitude and latitude components.
`polyline`	Logical. If FALSE (default), compute area and perimeter of a closed polygon. If TRUE, compute only the length of a polyline (area will be meaningless).

Details

This function is useful for understanding how polygon area accumulates and for debugging polygon vertex order issues.

Value

A data frame with columns:

- `lon`: Longitude of vertex
- `lat`: Latitude of vertex
- `area`: Cumulative area in square meters up to this vertex
- `perimeter`: Cumulative perimeter in meters up to this vertex

Examples

```
# Watch area accumulate as vertices are added
pts <- cbind(
  lon = c(0, -74, -43, 28),
  lat = c(52, 41, -23, -26)
)
polygon_area_cumulative(pts)
```

`rhumb_direct` *Rhumb line (loxodrome) calculations on the WGS84 ellipsoid*

Description

Solve rhumb line problems on the WGS84 ellipsoid. A rhumb line (or loxodrome) is a path of constant bearing, which appears as a straight line on a Mercator projection. Unlike geodesics, rhumb lines are not the shortest path between two points, but they are easier to navigate as they maintain a constant compass heading.

Usage

```
rhumb_direct(x, azi, s)

rhumb_inverse(x, y)

rhumb_path(x, y, n = 100L)

rhumb_line(x, azi, distances)

rhumb_distance(x, y)

rhumb_distance_matrix(x, y = NULL)
```

Arguments

`x`	A two-column matrix or data frame of starting coordinates (longitude, latitude) in decimal degrees.
`azi`	Numeric vector of azimuths (bearings) in degrees, measured clockwise from north.
`s`	Numeric vector of distances in meters.
`y`	A two-column matrix or data frame of ending coordinates (longitude, latitude) in decimal degrees.
`n`	Integer number of points to generate along the path (including start and end points).
`distances`	Numeric vector of distances from the starting point in meters.

Details

Rhumb lines are paths of constant azimuth (bearing). They are longer than geodesics (up to 50% longer for long distances) but are useful for navigation because they can be followed with a constant compass heading.

The azimuth is measured in degrees from north, with positive values clockwise (east) and negative values counter-clockwise (west). The range is -180 to 180 degrees.

The area `S12` represents the area under the rhumb line quadrilateral with corners at (lat1, lon1), (0, lon1), (0, lon2), and (lat2, lon2).

Value

- `rhumb_direct()`: Data frame with columns:
  - `lon1`, `lat1`: Starting coordinates
  - `azi12`: Azimuth (constant along rhumb line, degrees)
  - `s12`: Distance (meters)
  - `lon2`, `lat2`: Destination coordinates
  - `S12`: Area under rhumb line (square meters)
- `rhumb_inverse()`: Data frame with columns:
  - `lon1`, `lat1`: Starting coordinates
  - `lon2`, `lat2`: Ending coordinates
  - `s12`: Distance (meters)
  - `azi12`: Azimuth (degrees)
  - `S12`: Area under rhumb line (square meters)
- `rhumb_path()`: Data frame with columns:
  - `lon`, `lat`: Coordinates along the path
  - `s`: Distance from start (meters)
  - `azi12`: Constant azimuth (degrees)
- `rhumb_line()`: Data frame with columns:
  - `lon`, `lat`: Coordinates at specified distances
  - `azi`: Azimuth (degrees)
  - `s`: Distance from start (meters)
- `rhumb_distance()`: Numeric vector of distances in meters (pairwise).
- `rhumb_distance_matrix()`: Matrix of distances in meters.

See Also

`geodesic_direct()` for shortest-path geodesic calculations.

Examples

```
# Direct problem: Where do you end up starting from London,
# heading east on a rhumb line for 1000 km?
rhumb_direct(c(-0.1, 51.5), azi = 90, s = 1000000)

# Inverse problem: Rhumb distance from London to New York
rhumb_inverse(c(-0.1, 51.5), c(-74, 40.7))

# Compare to geodesic (rhumb is longer!)
geodesic_inverse(c(-0.1, 51.5), c(-74, 40.7))$s12
rhumb_inverse(c(-0.1, 51.5), c(-74, 40.7))$s12

# Generate a rhumb line path
path <- rhumb_path(c(-0.1, 51.5), c(-74, 40.7), n = 10)
path
```

---

tm_fwd *Transverse Mercator projection*

Description

Convert geographic coordinates to/from Transverse Mercator projection with user-specified central meridian and scale factor.

Two versions are provided:

- `tm_fwd()`/`tm_rev()`: Series approximation, fast, accurate to ~5 nanometers
- `tm_exact_fwd()`/`tm_exact_rev()`: Exact formulation, slower but accurate everywhere

Usage

```
tm_fwd(x, lon0, k0 = 0.9996)

tm_rev(x, y, lon0, k0 = 0.9996)

tm_exact_fwd(x, lon0, k0 = 0.9996)

tm_exact_rev(x, y, lon0, k0 = 0.9996)
```

Arguments

`x`	For forward conversion: a two-column matrix or data frame of coordinates (longitude, latitude) in decimal degrees. For reverse conversion: numeric vector of x (easting) coordinates in meters.
`lon0`	Central meridian in decimal degrees. Can be a vector to specify different central meridians for each point.
`k0`	Scale factor on the central meridian. Default is 0.9996 (UTM). Common values: 0.9996 (UTM), 1.0 (many national grids), 0.9999 (some state planes).
`y`	Numeric vector of y (northing) coordinates in meters (reverse only).

Details

The Transverse Mercator projection is a conformal cylindrical projection commonly used for:

- UTM (Universal Transverse Mercator) zones
- Many national and state coordinate systems
- Large-scale topographic mapping

Unlike `utmups_fwd()` which automatically selects UTM zones, these functions allow you to specify any central meridian and scale factor.

The series approximation (`tm_fwd`/`tm_rev`) is accurate to ~5 nanometers within 3900 km of the central meridian. The exact version (`tm_exact_fwd`/`tm_exact_rev`) is slower but works accurately everywhere.

The `lon0` parameter is vectorized, allowing different central meridians for each point (useful for processing data across multiple zones).

Value

Data frame with columns:

- For forward conversion:
  - `x`: Easting in meters
  - `y`: Northing in meters
  - `convergence`: Grid convergence in degrees
  - `scale`: Scale factor at the point
  - `lon`, `lat`: Input coordinates (echoed)
  - `lon0`: Central meridian (echoed)
- For reverse conversion:
  - `lon`: Longitude in decimal degrees
  - `lat`: Latitude in decimal degrees
  - `convergence`: Grid convergence in degrees
  - `scale`: Scale factor at the point
  - `x`, `y`: Input coordinates (echoed)
  - `lon0`: Central meridian (echoed)

See Also

`utmups_fwd()` for automatic UTM zone selection.

Examples

```
# Basic Transverse Mercator (like UTM zone 55)
pts <- cbind(lon = c(147, 148, 149), lat = c(-42, -43, -44))
tm_fwd(pts, lon0 = 147, k0 = 0.9996)

# Compare with UTM
utmups_fwd(pts)
```

```
# Custom scale factor (k0 = 1.0)
tm_fwd(pts, lon0 = 147, k0 = 1.0)

# Different central meridian for each point
tm_fwd(pts, lon0 = c(147, 148, 149), k0 = 0.9996)

# Round-trip conversion
fwd <- tm_fwd(pts, lon0 = 147, k0 = 0.9996)
tm_rev(fwd$x, fwd$y, lon0 = 147, k0 = 0.9996)

# Exact version for high precision or extreme locations
tm_exact_fwd(pts, lon0 = 147, k0 = 0.9996)
```

`utmups_fwd` *Convert coordinates to/from UTM/UPS projection*

Description

Convert geographic coordinates (longitude/latitude) to UTM or UPS projected coordinates, or convert projected coordinates back to geographic coordinates.

Usage

```
utmups_fwd(x)

utmups_rev(easting, northing, zone, northp)
```

Arguments

`x`	A two-column matrix or data frame of coordinates (longitude, latitude) in decimal degrees for forward conversion, or a list with longitude and latitude components. Can also be a length-2 numeric vector for a single point.
`easting`	Numeric vector of easting values (x coordinates) in meters for reverse conversion.
`northing`	Numeric vector of northing values (y coordinates) in meters for reverse conversion.
`zone`	Integer vector of UTM zone numbers (1-60) or 0 for UPS (polar regions).
`northp`	Logical vector indicating hemisphere: TRUE for northern hemisphere, FALSE for southern hemisphere.

Details

The Universal Transverse Mercator (UTM) system divides the Earth into 60 zones, each 6 degrees of longitude wide. For polar regions (latitude > 84°N or < 80°S), the Universal Polar Stereographic (UPS) system is used instead, indicated by zone = 0.

Both functions are fully vectorized. Missing values (NA) are not currently supported.

The convergence angle represents the angle between true north and grid north at a point. The scale factor represents the ratio of the scale along a line to the scale on the reference surface (typically very close to 1.0).

Value

- `utmups_fwd()`: Data frame with columns:
  - `x`: Easting in meters
  - `y`: Northing in meters
  - `zone`: UTM zone number (1-60) or 0 for UPS
  - `northp`: Logical, TRUE for northern hemisphere
  - `convergence`: Meridian convergence in degrees (angle between true north and grid north)
  - `scale`: Scale factor at the point (dimensionless, typically near 1.0)
  - `lon`: Longitude in decimal degrees (echoed from input)
  - `lat`: Latitude in decimal degrees (echoed from input)
  - `crs`: EPSG code string for the UTM/UPS projection
- `utmups_rev()`: Data frame with columns:
  - `lon`: Longitude in decimal degrees
  - `lat`: Latitude in decimal degrees
  - `x`: Easting in meters (echoed from input)
  - `y`: Northing in meters (echoed from input)
  - `zone`: UTM zone number (echoed from input)
  - `northp`: Hemisphere indicator (echoed from input)
  - `convergence`: Meridian convergence in degrees
  - `scale`: Scale factor at the point
  - `crs`: EPSG code string for the UTM/UPS projection

Examples

```
# Single point forward conversion
result <- utmups_fwd(c(147.325, -42.881))
result

# Multiple points
pts <- cbind(lon = c(147, 148, -100, 0),
             lat = c(-42, -43, -42, 0))
utmups_fwd(pts)

# Reverse conversion
utmups_rev(result$x, result$y, result$zone, result$northp)

# Round-trip conversion
fwd <- utmups_fwd(pts)
rev <- utmups_rev(fwd$x, fwd$y, fwd$zone, fwd$northp)
cbind(original = pts, converted = rev[, c("lon", "lat")])

# Polar regions use UPS (zone 0)
```

```
polar <- cbind(c(147, 148, -100), c(88, -88, -85))
utmups_fwd(polar)
```

Index