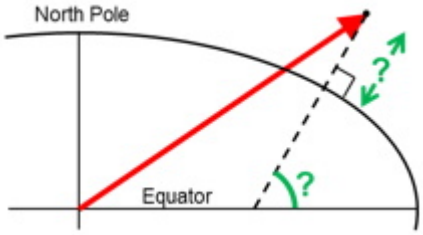


Nvector Documentation for Python

Release 0.7.7

Kenneth Gade and Per A. Brodtkorb

Jun 03, 2021

CONTENTS:

This is the documentation of **nvector** version 0.7.7 for Python released Jun 03, 2021.

Bleeding edge available at: https://github.com/pbrod/nvector.

Official releases are available at: http://pypi.python.org/pypi/nvector.

Official homepage are available at: http://www.navlab.net/nvector/

CHAPTER

ONE

INTRODUCTION

1.1 What is nvector?

The nvector library is a suite of tools written in Python to solve geographical position calculations. Currently the following operations are implemented:

- Calculate the surface distance between two geographical positions.
- Convert positions given in one reference frame into another reference frame.
- Find the destination point given start point, azimuth/bearing and distance.
- Find the mean position (center/midpoint) of several geographical positions.
- Find the intersection between two paths.
- Find the cross track distance between a path and a position.

Using n-vector, the calculations become simple and non-singular. Full accuracy is achieved for any global position (and for any distance).

1.1.1 Description

In this library, we represent position with an "n-vector", which is the normal vector to the Earth model (the same reference ellipsoid that is used for latitude and longitude). When using n-vector, all Earth-positions are treated equally, and there is no need to worry about singularities or discontinuities. An additional benefit with using n-vector is that many position calculations can be solved with simple vector algebra (e.g. dot product and cross product).

Converting between n-vector and latitude/longitude is unambiguous and easy using the provided functions.

n_E is n-vector in the program code, while in documents we use nE. E denotes an Earth-fixed coordinate frame, and it indicates that the three components of n-vector are along the three axes of E. More details about the notation and reference frames can be found in the documentation.[1]

1.2 How the documentation is organized

Nvector has a lot of documentation. A high-level overview of how it's organized will help you know where to look for certain things:

- *Tutorials* take you by the hand through a series of typical usecases on how to use it. Start here if you're new to nvector.
- *Topic guides* discuss key topics and concepts at a fairly high level and provide useful background information and explanation.

[1] https://www.navlab.net/nvector/#vector_symbols

- *Reference guides* contain technical reference for APIs and other aspects of nvector's machinery. They describe how it works and how to use it but assume that you have a basic understanding of key concepts.
- *How-to guides* are recipes. They guide you through the steps involved in addressing key problems and use-cases. They are more advanced than tutorials and assume some knowledge of how nvector works.

CHAPTER

TWO

TUTORIALS

The pages in this section of the documentation are aimed at the newcomer to nvector. They're designed to help you get started quickly, and show how easy it is to work with nvector as a developer who wants to customise it and get it working according to their own requirements.

These tutorials take you step-by-step through some key aspects of this work. They're not intended to explain the *topics in depth*, or provide *reference material*, but they will leave you with a good idea of what it's possible to achieve in just a few steps, and how to go about it.

Once you're familiar with the basics presented in these tutorials, you'll find the more in-depth coverage of the same topics in the *How-to* section.

The tutorials follow a logical progression, starting from installation of nvector and the creation of a brand new project, and build on each other, so it's recommended to work through them in the order presented here.

2.1 Install guide

Before you can use nvector, you'll need to get it installed. This guide will guide you through a simple installation that'll work while you walk through the introduction.

2.1.1 Install nvector

If you have pip installed and are online, then simply type:

$ pip install nvector

to get the lastest stable version. Using pip also has the advantage that all requirements are automatically installed.

You can download nvector and all dependencies to a folder "pkg", by the following:

$ pip install –download=pkg nvector

To install the downloaded nvector, just type:

$ pip install –no-index –find-links=pkg nvector

2.1.2 Verifying installation

To verify that nvector can be seen by Python, type `python` from your shell. Then at the Python prompt, try to import nvector:

```
>>> import nvector as nv
>>> print(nv.__version__)
0.7.7
```

To test if the toolbox is working correctly paste the following in an interactive python session:

```
import nvector as nv
nv.test('--doctest-modules')
```

or

$ py.test –pyargs nvector –doctest-modules

at the command prompt.

2.2 Getting Started

Below the object-oriented solution to some common geodesic problems are given. In the first example the functional solution is also given. The functional solutions to the remaining problems can be found in the *functional examples* section of the tutorial.

2.2.1 Example 1: "A and B to delta"

Given two positions, A and B as latitudes, longitudes and depths relative to Earth, E.

Find the exact vector between the two positions, given in meters north, east, and down, and find the direction (azimuth) to B, relative to north. Assume WGS-84 ellipsoid. The given depths are from the ellipsoid surface. Use position A to define north, east, and down directions. (Due to the curvature of Earth and different directions to the North Pole, the north, east, and down directions will change (relative to Earth) for different places. Position A must be outside the poles for the north and east directions to be defined.)

Solution:

```
>>> import numpy as np
>>> import nvector as nv
>>> wgs84 = nv.FrameE(name='WGS84')
>>> pointA = wgs84.GeoPoint(latitude=1, longitude=2, z=3, degrees=True)
>>> pointB = wgs84.GeoPoint(latitude=4, longitude=5, z=6, degrees=True)
```

Step1: Find p_AB_N (delta decomposed in N).

```
>>> p_AB_N = pointA.delta_to(pointB)
>>> x, y, z = p_AB_N.pvector.ravel()
>>> 'Ex1: delta north, east, down = {0:8.2f}, {1:8.2f}, {2:8.2f}'.format(x, y, z)
'Ex1: delta north, east, down = 331730.23, 332997.87, 17404.27'
```

Step2: Also find the direction (azimuth) to B, relative to north:

```
>>> 'azimuth = {0:4.2f} deg'.format(p_AB_N.azimuth_deg)
'azimuth = 45.11 deg'
>>> 'elevation = {0:4.2f} deg'.format(p_AB_N.elevation_deg)
'elevation = 2.12 deg'
>>> 'distance = {0:4.2f} m'.format(p_AB_N.length)
'distance = 470356.72 m'
```

Functional Solution:

```
>>> import numpy as np
>>> import nvector as nv
>>> from nvector import rad, deg
```

```
>>> lat_EA, lon_EA, z_EA = rad(1), rad(2), 3
>>> lat_EB, lon_EB, z_EB = rad(4), rad(5), 6
```

Step1: Convert to n-vectors:

```
>>> n_EA_E = nv.lat_lon2n_E(lat_EA, lon_EA)
>>> n_EB_E = nv.lat_lon2n_E(lat_EB, lon_EB)
```

Step2: Find p_AB_E (delta decomposed in E).WGS-84 ellipsoid is default:

```
>>> p_AB_E = nv.n_EA_E_and_n_EB_E2p_AB_E(n_EA_E, n_EB_E, z_EA, z_EB)
```

Step3: Find R_EN for position A:

```
>>> R_EN = nv.n_E2R_EN(n_EA_E)
```

Step4: Find p_AB_N (delta decomposed in N).

```
>>> p_AB_N = np.dot(R_EN.T, p_AB_E).ravel()
>>> x, y, z = p_AB_N
>>> 'Ex1: delta north, east, down = {0:8.2f}, {1:8.2f}, {2:8.2f}'.format(x, y, z)
'Ex1: delta north, east, down = 331730.23, 332997.87, 17404.27'
```

Step5: Also find the direction (azimuth) to B, relative to north:

```
>>> azimuth = np.arctan2(y, x)
>>> 'azimuth = {0:4.2f} deg'.format(deg(azimuth))
'azimuth = 45.11 deg'
```

```
>>> distance = np.linalg.norm(p_AB_N)
>>> elevation = np.arcsin(z / distance)
>>> 'elevation = {0:4.2f} deg'.format(deg(elevation))
'elevation = 2.12 deg'
```

```
>>> 'distance = {0:4.2f} m'.format(distance)
'distance = 470356.72 m'
```

See also Example 1 at www.navlab.net[2]

2.2.2 Example 2: "B and delta to C"

[2] http://www.navlab.net/nvector/#example_1

A radar or sonar attached to a vehicle B (Body coordinate frame) measures the distance and direction to an object C. We assume that the distance and two angles (typically bearing and elevation relative to B) are already combined to the vector p_BC_B (i.e. the vector from B to C, decomposed in B). The position of B is given as n_EB_E and z_EB, and the orientation (attitude) of B is given as R_NB (this rotation matrix can be found from roll/pitch/yaw by using zyx2R).

Find the exact position of object C as n-vector and depth (n_EC_E and z_EC), assuming Earth ellipsoid with semi-major axis a and flattening f. For WGS-72, use a = 6 378 135 m and f = 1/298.26.

Solution:

```
>>> import numpy as np
>>> import nvector as nv
>>> wgs72 = nv.FrameE(name='WGS72')
>>> wgs72 = nv.FrameE(a=6378135, f=1.0/298.26)
```

Step 1: Position and orientation of B is given 400m above E:

```
>>> n_EB_E = wgs72.Nvector(nv.unit([[1], [2], [3]]), z=-400)
>>> frame_B = nv.FrameB(n_EB_E, yaw=10, pitch=20, roll=30, degrees=True)
```

Step 2: Delta BC decomposed in B

```
>>> p_BC_B = frame_B.Pvector(np.r_[3000, 2000, 100].reshape((-1, 1)))
```

Step 3: Decompose delta BC in E

```
>>> p_BC_E = p_BC_B.to_ecef_vector()
```

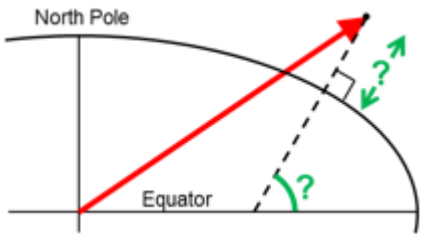
Step 4: Find point C by adding delta BC to EB

```
>>> p_EB_E = n_EB_E.to_ecef_vector()
>>> p_EC_E = p_EB_E + p_BC_E
>>> pointC = p_EC_E.to_geo_point()
```

```
>>> lat, lon, z = pointC.latlon_deg
>>> msg = 'Ex2: PosC: lat, lon = {:4.4f}, {:4.4f} deg,  height = {:4.2f} m'
>>> msg.format(lat, lon, -z)
'Ex2: PosC: lat, lon = 53.3264, 63.4681 deg,  height = 406.01 m'
```

See also Example 2 at www.navlab.net[3]

2.2.3 Example 3: "ECEF-vector to geodetic latitude"



Position B is given as an "ECEF-vector" p_EB_E (i.e. a vector from E, the center of the Earth, to B, decomposed in E). Find the geodetic latitude, longitude and height (latEB, lonEB and hEB), assuming WGS-84 ellipsoid.

Solution:

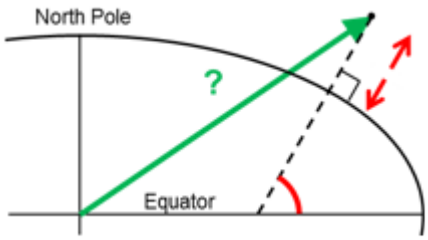
[3] http://www.navlab.net/nvector/#example_2

```
>>> import numpy as np
>>> import nvector as nv
>>> wgs84 = nv.FrameE(name='WGS84')
>>> position_B = 6371e3 * np.vstack((0.9, -1, 1.1))  # m
>>> p_EB_E = wgs84.ECEFvector(position_B)
>>> pointB = p_EB_E.to_geo_point()
```

```
>>> lat, lon, z = pointB.latlon_deg
>>> 'Ex3: Pos B: lat, lon = {:4.4f}, {:4.4f} deg, height = {:9.3f} m'.format(lat,
↪ lon, -z)
'Ex3: Pos B: lat, lon = 39.3787, -48.0128 deg, height = 4702059.834 m'
```

See also Example 3 at www.navlab.net[4]

2.2.4 Example 4: "Geodetic latitude to ECEF-vector"



Geodetic latitude, longitude and height are given for position B as latEB, lonEB and hEB, find the ECEF-vector for this position, p_EB_E.

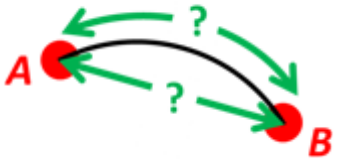
Solution:

```
>>> import nvector as nv
>>> wgs84 = nv.FrameE(name='WGS84')
>>> pointB = wgs84.GeoPoint(latitude=1, longitude=2, z=-3, degrees=True)
>>> p_EB_E = pointB.to_ecef_vector()
```

```
>>> 'Ex4: p_EB_E = {} m'.format(p_EB_E.pvector.ravel().tolist())
'Ex4: p_EB_E = [6373290.277218279, 222560.20067473652, 110568.82718178593] m'
```

See also Example 4 at www.navlab.net[5]

2.2.5 Example 5: "Surface distance"



Find the surface distance sAB (i.e. great circle distance) between two positions A and B. The heights of A and B are ignored, i.e. if they don't have zero height, we seek the distance between the points that are at the surface of the Earth, directly above/below A and B. The Euclidean distance (chord length) dAB should also be found. Use Earth radius 6371e3 m. Compare the results with exact calculations for the WGS-84 ellipsoid.

[4] http://www.navlab.net/nvector/#example_3
[5] http://www.navlab.net/nvector/#example_4

Solution for a sphere:

```
>>> import numpy as np
>>> import nvector as nv
>>> frame_E = nv.FrameE(a=6371e3, f=0)
>>> positionA = frame_E.GeoPoint(latitude=88, longitude=0, degrees=True)
>>> positionB = frame_E.GeoPoint(latitude=89, longitude=-170, degrees=True)
```

```
>>> s_AB, azia, azib = positionA.distance_and_azimuth(positionB)
>>> p_AB_E = positionB.to_ecef_vector() - positionA.to_ecef_vector()
>>> d_AB = p_AB_E.length
```

```
>>> msg = 'Ex5: Great circle and Euclidean distance = {}'
>>> msg = msg.format('{:5.2f} km, {:5.2f} km')
>>> msg.format(s_AB / 1000, d_AB / 1000)
'Ex5: Great circle and Euclidean distance = 332.46 km, 332.42 km'
```

Alternative sphere solution:

```
>>> path = nv.GeoPath(positionA, positionB)
>>> s_AB2 = path.track_distance(method='greatcircle')
>>> d_AB2 = path.track_distance(method='euclidean')
>>> msg.format(s_AB2 / 1000, d_AB2 / 1000)
'Ex5: Great circle and Euclidean distance = 332.46 km, 332.42 km'
```

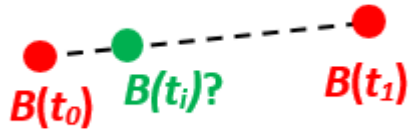
Exact solution for the WGS84 ellipsoid:

```
>>> wgs84 = nv.FrameE(name='WGS84')
>>> point1 = wgs84.GeoPoint(latitude=88, longitude=0, degrees=True)
>>> point2 = wgs84.GeoPoint(latitude=89, longitude=-170, degrees=True)
>>> s_12, azi1, azi2 = point1.distance_and_azimuth(point2)
```

```
>>> p_12_E = point2.to_ecef_vector() - point1.to_ecef_vector()
>>> d_12 = p_12_E.length
>>> msg = 'Ellipsoidal and Euclidean distance = {:5.2f} km, {:5.2f} km'
>>> msg.format(s_12 / 1000, d_12 / 1000)
'Ellipsoidal and Euclidean distance = 333.95 km, 333.91 km'
```

See also Example 5 at www.navlab.net[6]

2.2.6 Example 6 "Interpolated position"



Given the position of B at time t0 and t1, n_EB_E(t0) and n_EB_E(t1).

Find an interpolated position at time ti, n_EB_E(ti). All positions are given as n-vectors.

Solution:

[6] http://www.navlab.net/nvector/#example_5

```
>>> import nvector as nv
>>> wgs84 = nv.FrameE(name='WGS84')
>>> n_EB_E_t0 = wgs84.GeoPoint(89, 0, degrees=True).to_nvector()
>>> n_EB_E_t1 = wgs84.GeoPoint(89, 180, degrees=True).to_nvector()
>>> path = nv.GeoPath(n_EB_E_t0, n_EB_E_t1)
```

```
>>> t0 = 10.
>>> t1 = 20.
>>> ti = 16.  # time of interpolation
>>> ti_n = (ti - t0) / (t1 - t0) # normalized time of interpolation
```

```
>>> g_EB_E_ti = path.interpolate(ti_n).to_geo_point()
```

```
>>> lat_ti, lon_ti, z_ti = g_EB_E_ti.latlon_deg
>>> msg = 'Ex6, Interpolated position: lat, lon = {:2.1f} deg, {:2.1f} deg'
>>> msg.format(lat_ti, lon_ti)
'Ex6, Interpolated position: lat, lon = 89.8 deg, 180.0 deg'
```

Vectorized solution:

```
>>> t = np.array([10, 20])
>>> nvectors = wgs84.GeoPoint([89, 89], [0, 180], degrees=True).to_nvector()
>>> nvectors_i = nvectors.interpolate(ti, t, kind='linear')
>>> lati, loni, zi = nvectors_i.to_geo_point().latlon_deg
>>> msg.format(lat_ti, lon_ti)
'Ex6, Interpolated position: lat, lon = 89.8 deg, 180.0 deg'
```

See also Example 6 at www.navlab.net[7]

2.2.7 Example 7: "Mean position"

Three positions A, B, and C are given as n-vectors n_EA_E, n_EB_E, and n_EC_E. Find the mean position, M, given as n_EM_E. Note that the calculation is independent of the depths of the positions.

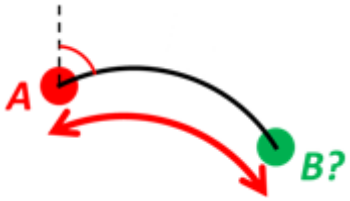
Solution:

```
>>> import nvector as nv
>>> points = nv.GeoPoint(latitude=[90, 60, 50],
...                      longitude=[0, 10, -20], degrees=True)
>>> nvectors = points.to_nvector()
>>> n_EM_E = nvectors.mean()
>>> g_EM_E = n_EM_E.to_geo_point()
>>> lat, lon = g_EM_E.latitude_deg, g_EM_E.longitude_deg
>>> msg = 'Ex7: Pos M: lat, lon = {:4.4f}, {:4.4f} deg'
>>> msg.format(lat, lon)
'Ex7: Pos M: lat, lon = 67.2362, -6.9175 deg'
```

[7] http://www.navlab.net/nvector/#example_6

See also Example 7 at www.navlab.net[8]

2.2.8 Example 8: "A and azimuth/distance to B"



We have an initial position A, direction of travel given as an azimuth (bearing) relative to north (clockwise), and finally the distance to travel along a great circle given as sAB. Use Earth radius 6371e3 m to find the destination point B.

In geodesy this is known as "The first geodetic problem" or "The direct geodetic problem" for a sphere, and we see that this is similar to Example 2[9], but now the delta is given as an azimuth and a great circle distance. ("The second/inverse geodetic problem" for a sphere is already solved in Examples 1[10] and 5[11].)

Exact solution:

```
>>> import numpy as np
>>> import nvector as nv
>>> frame = nv.FrameE(a=6371e3, f=0)
>>> pointA = frame.GeoPoint(latitude=80, longitude=-90, degrees=True)
>>> pointB, azimuthb = pointA.displace(distance=1000, azimuth=200, degrees=True)
>>> lat, lon = pointB.latitude_deg, pointB.longitude_deg
```

```
>>> msg = 'Ex8, Destination: lat, lon = {:4.4f} deg, {:4.4f} deg'
>>> msg.format(lat, lon)
'Ex8, Destination: lat, lon = 79.9915 deg, -90.0177 deg'
```

```
>>> np.allclose(azimuthb, -160.01742926820506)
True
```

Greatcircle solution:

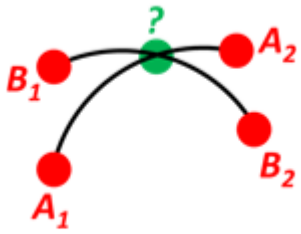
```
>>> pointB2, azimuthb = pointA.displace(distance=1000,
...                                     azimuth=200,
...                                     degrees=True,
...                                     method='greatcircle')
>>> lat2, lon2 = pointB2.latitude_deg, pointB.longitude_deg
>>> msg.format(lat2, lon2)
'Ex8, Destination: lat, lon = 79.9915 deg, -90.0177 deg'
```

```
>>> np.allclose(azimuthb, -160.0174292682187)
True
```

See also Example 8 at www.navlab.net[12]

[8] http://www.navlab.net/nvector/#example_7
[9] http://www.navlab.net/nvector/#example_2
[10] http://www.navlab.net/nvector/#example_1
[11] http://www.navlab.net/nvector/#example_5
[12] http://www.navlab.net/nvector/#example_8

2.2.9 Example 9: "Intersection of two paths"



Define a path from two given positions (at the surface of a spherical Earth), as the great circle that goes through the two points.

Path A is given by A1 and A2, while path B is given by B1 and B2.

Find the position C where the two great circles intersect.

Solution:

```
>>> import nvector as nv
>>> pointA1 = nv.GeoPoint(10, 20, degrees=True)
>>> pointA2 = nv.GeoPoint(30, 40, degrees=True)
>>> pointB1 = nv.GeoPoint(50, 60, degrees=True)
>>> pointB2 = nv.GeoPoint(70, 80, degrees=True)
>>> pathA = nv.GeoPath(pointA1, pointA2)
>>> pathB = nv.GeoPath(pointB1, pointB2)
```

```
>>> pointC = pathA.intersect(pathB)
>>> pointC = pointC.to_geo_point()
>>> lat, lon = pointC.latitude_deg, pointC.longitude_deg
>>> msg = 'Ex9, Intersection: lat, lon = {:4.4f}, {:4.4f} deg'
>>> msg.format(lat, lon)
'Ex9, Intersection: lat, lon = 40.3186, 55.9019 deg'
```

Check that PointC is not between A1 and A2 or B1 and B2:

```
>>> pathA.on_path(pointC)
False
>>> pathB.on_path(pointC)
False
```

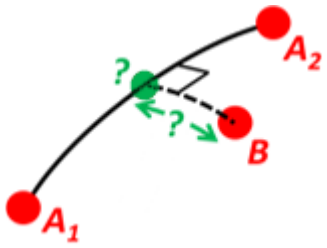
Check that PointC is on the great circle going through path A and path B:

```
>>> pathA.on_great_circle(pointC)
True
>>> pathB.on_great_circle(pointC)
True
```

See also Example 9 at www.navlab.net[13]

[13] http://www.navlab.net/nvector/#example_9

2.2.10 Example 10: "Cross track distance"



Path A is given by the two positions A1 and A2 (similar to the previous example).

Find the cross track distance sxt between the path A (i.e. the great circle through A1 and A2) and the position B (i.e. the shortest distance at the surface, between the great circle and B).

Also find the Euclidean distance dxt between B and the plane defined by the great circle. Use Earth radius 6371e3.

Finally, find the intersection point on the great circle and determine if it is between position A1 and A2.

Solution:

```
>>> import numpy as np
>>> import nvector as nv
>>> frame = nv.FrameE(a=6371e3, f=0)
>>> pointA1 = frame.GeoPoint(0, 0, degrees=True)
>>> pointA2 = frame.GeoPoint(10, 0, degrees=True)
>>> pointB = frame.GeoPoint(1, 0.1, degrees=True)
>>> pathA = nv.GeoPath(pointA1, pointA2)
```

```
>>> s_xt = pathA.cross_track_distance(pointB, method='greatcircle')
>>> d_xt = pathA.cross_track_distance(pointB, method='euclidean')
```

```
>>> val_txt = '{:4.2f} km, {:4.2f} km'.format(s_xt/1000, d_xt/1000)
>>> 'Ex10: Cross track distance: s_xt, d_xt = {}'.format(val_txt)
'Ex10: Cross track distance: s_xt, d_xt = 11.12 km, 11.12 km'
```

```
>>> pointC = pathA.closest_point_on_great_circle(pointB)
>>> np.allclose(pathA.on_path(pointC), True)
True
```

See also Example 10 at www.navlab.net[14]

2.3 Functional examples

Below the functional solution to some common geodesic problems are given. In the first example the object-oriented solution is also given. The object-oriented solutions to the remaining problems can be found in the *getting started* section of the tutorial.

[14] http://www.navlab.net/nvector/#example_10

2.3.1 Example 1: "A and B to delta"

Given two positions, A and B as latitudes, longitudes and depths relative to Earth, E.

Find the exact vector between the two positions, given in meters north, east, and down, and find the direction (azimuth) to B, relative to north. Assume WGS-84 ellipsoid. The given depths are from the ellipsoid surface. Use position A to define north, east, and down directions. (Due to the curvature of Earth and different directions to the North Pole, the north, east, and down directions will change (relative to Earth) for different places. Position A must be outside the poles for the north and east directions to be defined.)

Solution:

```
>>> import numpy as np
>>> import nvector as nv
>>> from nvector import rad, deg
```

```
>>> lat_EA, lon_EA, z_EA = rad(1), rad(2), 3
>>> lat_EB, lon_EB, z_EB = rad(4), rad(5), 6
```

Step1: Convert to n-vectors:

```
>>> n_EA_E = nv.lat_lon2n_E(lat_EA, lon_EA)
>>> n_EB_E = nv.lat_lon2n_E(lat_EB, lon_EB)
```

Step2: Find p_AB_E (delta decomposed in E).WGS-84 ellipsoid is default:

```
>>> p_AB_E = nv.n_EA_E_and_n_EB_E2p_AB_E(n_EA_E, n_EB_E, z_EA, z_EB)
```

Step3: Find R_EN for position A:

```
>>> R_EN = nv.n_E2R_EN(n_EA_E)
```

Step4: Find p_AB_N (delta decomposed in N).

```
>>> p_AB_N = np.dot(R_EN.T, p_AB_E).ravel()
>>> x, y, z = p_AB_N
>>> 'Ex1: delta north, east, down = {0:8.2f}, {1:8.2f}, {2:8.2f}'.format(x, y, z)
'Ex1: delta north, east, down = 331730.23, 332997.87, 17404.27'
```

Step5: Also find the direction (azimuth) to B, relative to north:

```
>>> azimuth = np.arctan2(y, x)
>>> 'azimuth = {0:4.2f} deg'.format(deg(azimuth))
'azimuth = 45.11 deg'
```

```
>>> distance = np.linalg.norm(p_AB_N)
>>> elevation = np.arcsin(z / distance)
>>> 'elevation = {0:4.2f} deg'.format(deg(elevation))
'elevation = 2.12 deg'
```

```
>>> 'distance = {0:4.2f} m'.format(distance)
'distance = 470356.72 m'
```

OO-Solution:

```
>>> import numpy as np
>>> import nvector as nv
>>> wgs84 = nv.FrameE(name='WGS84')
>>> pointA = wgs84.GeoPoint(latitude=1, longitude=2, z=3, degrees=True)
>>> pointB = wgs84.GeoPoint(latitude=4, longitude=5, z=6, degrees=True)
```

Step1: Find p_AB_N (delta decomposed in N).

```
>>> p_AB_N = pointA.delta_to(pointB)
>>> x, y, z = p_AB_N.pvector.ravel()
>>> 'Ex1: delta north, east, down = {0:8.2f}, {1:8.2f}, {2:8.2f}'.format(x, y, z)
'Ex1: delta north, east, down = 331730.23, 332997.87, 17404.27'
```

Step2: Also find the direction (azimuth) to B, relative to north:

```
>>> 'azimuth = {0:4.2f} deg'.format(p_AB_N.azimuth_deg)
'azimuth = 45.11 deg'
>>> 'elevation = {0:4.2f} deg'.format(p_AB_N.elevation_deg)
'elevation = 2.12 deg'
>>> 'distance = {0:4.2f} m'.format(p_AB_N.length)
'distance = 470356.72 m'
```

See also Example 1 at www.navlab.net[15]

2.3.2 Example 2: "B and delta to C"

A radar or sonar attached to a vehicle B (Body coordinate frame) measures the distance and direction to an object C. We assume that the distance and two angles (typically bearing and elevation relative to B) are already combined to the vector p_BC_B (i.e. the vector from B to C, decomposed in B). The position of B is given as n_EB_E and z_EB, and the orientation (attitude) of B is given as R_NB (this rotation matrix can be found from roll/pitch/yaw by using zyx2R).

Find the exact position of object C as n-vector and depth (n_EC_E and z_EC), assuming Earth ellipsoid with semi-major axis a and flattening f. For WGS-72, use a = 6 378 135 m and f = 1/298.26.

Solution:

```
>>> import numpy as np
>>> import nvector as nv
>>> from nvector import rad, deg
```

A custom reference ellipsoid is given (replacing WGS-84):

[15] http://www.navlab.net/nvector/#example_1

```
>>> wgs72 = dict(a=6378135, f=1.0/298.26)
```

Step 1 Position and orientation of B is 400m above E:

```
>>> n_EB_E = nv.unit([[1], [2], [3]])  # unit to get unit length of vector
>>> z_EB = -400
>>> yaw, pitch, roll = rad(10), rad(20), rad(30)
>>> R_NB = nv.zyx2R(yaw, pitch, roll)
```

Step 2: Delta BC decomposed in B

```
>>> p_BC_B = np.r_[3000, 2000, 100].reshape((-1, 1))
```

Step 3: Find R_EN:

```
>>> R_EN = nv.n_E2R_EN(n_EB_E)
```

Step 4: Find R_EB, from R_EN and R_NB:

```
>>> R_EB = np.dot(R_EN, R_NB)  # Note: closest frames cancel
```

Step 5: Decompose the delta BC vector in E:

```
>>> p_BC_E = np.dot(R_EB, p_BC_B)
```

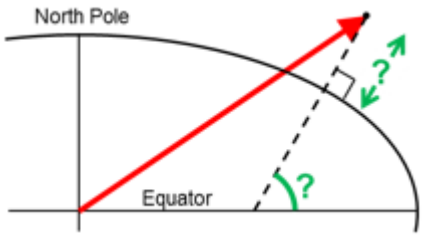
Step 6: Find the position of C, using the functions that goes from one

```
>>> n_EC_E, z_EC = nv.n_EA_E_and_p_AB_E2n_EB_E(n_EB_E, p_BC_E, z_EB, **wgs72)
```

```
>>> lat_EC, lon_EC = nv.n_E2lat_lon(n_EC_E)
>>> lat, lon, z = deg(lat_EC), deg(lon_EC), z_EC
>>> msg = 'Ex2: PosC: lat, lon = {:4.4f}, {:4.4f} deg,  height = {:4.2f} m'
>>> msg.format(lat[0], lon[0], -z[0])
'Ex2: PosC: lat, lon = 53.3264, 63.4681 deg,  height = 406.01 m'
```

See also Example 2 at www.navlab.net[16]

2.3.3 Example 3: "ECEF-vector to geodetic latitude"



Position B is given as an "ECEF-vector" p_EB_E (i.e. a vector from E, the center of the Earth, to B, decomposed in E). Find the geodetic latitude, longitude and height (latEB, lonEB and hEB), assuming WGS-84 ellipsoid.

Solution:

```
>>> import numpy as np
>>> import nvector as nv
>>> from nvector import deg
```

(continues on next page)

[16] http://www.navlab.net/nvector/#example_2

(continued from previous page)

```
>>> wgs84 = dict(a=6378137.0, f=1.0/298.257223563)
>>> p_EB_E = 6371e3 * np.vstack((0.9, -1, 1.1))  # m
```

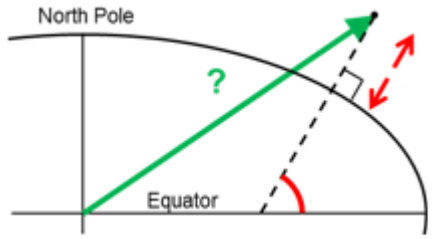
```
>>> n_EB_E, z_EB = nv.p_EB_E2n_EB_E(p_EB_E, **wgs84)
```

```
>>> lat_EB, lon_EB = nv.n_E2lat_lon(n_EB_E)
>>> h = -z_EB
>>> lat, lon = deg(lat_EB), deg(lon_EB)
```

```
>>> msg = 'Ex3: Pos B: lat, lon = {:4.4f}, {:4.4f} deg, height = {:9.3f} m'
>>> msg.format(lat[0], lon[0], h[0])
'Ex3: Pos B: lat, lon = 39.3787, -48.0128 deg, height = 4702059.834 m'
```

See also Example 3 at www.navlab.net[17]

2.3.4 Example 4: "Geodetic latitude to ECEF-vector"



Geodetic latitude, longitude and height are given for position B as latEB, lonEB and hEB, find the ECEF-vector for this position, p_EB_E.

Solution:

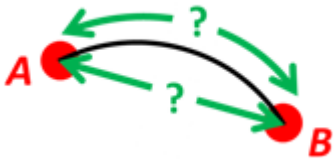
```
>>> import nvector as nv
>>> from nvector import rad
>>> wgs84 = dict(a=6378137.0, f=1.0/298.257223563)
>>> lat_EB, lon_EB = rad(1), rad(2)
>>> h_EB = 3
>>> n_EB_E = nv.lat_lon2n_E(lat_EB, lon_EB)
>>> p_EB_E = nv.n_EB_E2p_EB_E(n_EB_E, -h_EB, **wgs84)
```

```
>>> 'Ex4: p_EB_E = {} m'.format(p_EB_E.ravel().tolist())
'Ex4: p_EB_E = [6373290.277218279, 222560.20067473652, 110568.82718178593] m'
```

See also Example 4 at www.navlab.net[18]

[17] http://www.navlab.net/nvector/#example_3
[18] http://www.navlab.net/nvector/#example_4

2.3.5 Example 5: "Surface distance"



Find the surface distance sAB (i.e. great circle distance) between two positions A and B. The heights of A and B are ignored, i.e. if they don't have zero height, we seek the distance between the points that are at the surface of the Earth, directly above/below A and B. The Euclidean distance (chord length) dAB should also be found. Use Earth radius 6371e3 m. Compare the results with exact calculations for the WGS-84 ellipsoid.

Solution for a sphere:

```
>>> import numpy as np
>>> import nvector as nv
>>> from nvector import rad
```

```
>>> n_EA_E = nv.lat_lon2n_E(rad(88), rad(0))
>>> n_EB_E = nv.lat_lon2n_E(rad(89), rad(-170))
```

```
>>> r_Earth = 6371e3  # m, mean Earth radius
>>> s_AB = nv.great_circle_distance(n_EA_E, n_EB_E, radius=r_Earth)[0]
>>> d_AB = nv.euclidean_distance(n_EA_E, n_EB_E, radius=r_Earth)[0]
```

```
>>> msg = 'Ex5: Great circle and Euclidean distance = {}'
>>> msg = msg.format('{:5.2f} km, {:5.2f} km')
>>> msg.format(s_AB / 1000, d_AB / 1000)
'Ex5: Great circle and Euclidean distance = 332.46 km, 332.42 km'
```

Exact solution for the WGS84 ellipsoid:

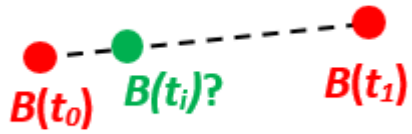
```
>>> wgs84 = nv.FrameE(name='WGS84')
>>> point1 = wgs84.GeoPoint(latitude=88, longitude=0, degrees=True)
>>> point2 = wgs84.GeoPoint(latitude=89, longitude=-170, degrees=True)
>>> s_12, _azi1, _azi2 = point1.distance_and_azimuth(point2)
```

```
>>> p_12_E = point2.to_ecef_vector() - point1.to_ecef_vector()
>>> d_12 = p_12_E.length
>>> msg = 'Ellipsoidal and Euclidean distance = {:5.2f} km, {:5.2f} km'
>>> msg.format(s_12 / 1000, d_12 / 1000)
'Ellipsoidal and Euclidean distance = 333.95 km, 333.91 km'
```

See also Example 5 at www.navlab.net[19]

[19] http://www.navlab.net/nvector/#example_5

2.3.6 Example 6 "Interpolated position"



Given the position of B at time t0 and t1, n_EB_E(t0) and n_EB_E(t1).

Find an interpolated position at time ti, n_EB_E(ti). All positions are given as n-vectors.

Solution:

```
>>> import nvector as nv
>>> from nvector import rad, deg
>>> n_EB_E_t0 = nv.lat_lon2n_E(rad(89), rad(0))
>>> n_EB_E_t1 = nv.lat_lon2n_E(rad(89), rad(180))
```

```
>>> t0 = 10.
>>> t1 = 20.
>>> ti = 16.  # time of interpolation
>>> ti_n = (ti - t0) / (t1 - t0) # normalized time of interpolation
```

```
>>> n_EB_E_ti = nv.unit(n_EB_E_t0 + ti_n * (n_EB_E_t1 - n_EB_E_t0))
>>> lat_EB_ti, lon_EB_ti = nv.n_E2lat_lon(n_EB_E_ti)
```

```
>>> lat_ti, lon_ti = deg(lat_EB_ti), deg(lon_EB_ti)
>>> msg = 'Ex6, Interpolated position: lat, lon = {:2.1f} deg, {:2.1f} deg'
>>> msg.format(lat_ti[0], lon_ti[0])
'Ex6, Interpolated position: lat, lon = 89.8 deg, 180.0 deg'
```

Vectorized solution:

```
>>> nvectors = nv.lat_lon2n_E(rad([89, 89]), rad([0, 180]))
>>> t = np.array([10, 20])
>>> nvectors_i = nv.interp_nvectors(ti, t, nvectors, kind='linear')
>>> lati, loni = nv.deg(*nv.n_E2lat_lon(nvectors_i))
>>> msg.format(lat_ti[0], lon_ti[0])
'Ex6, Interpolated position: lat, lon = 89.8 deg, 180.0 deg'
```

See also Example 6 at www.navlab.net[20]

2.3.7 Example 7: "Mean position"

[20] http://www.navlab.net/nvector/#example_6

Three positions A, B, and C are given as n-vectors n_EA_E, n_EB_E, and n_EC_E. Find the mean position, M, given as n_EM_E. Note that the calculation is independent of the depths of the positions.

Solution:

```
>>> import numpy as np
>>> import nvector as nv
>>> from nvector import rad, deg
```

```
>>> n_EA_E = nv.lat_lon2n_E(rad(90), rad(0))
>>> n_EB_E = nv.lat_lon2n_E(rad(60), rad(10))
>>> n_EC_E = nv.lat_lon2n_E(rad(50), rad(-20))
```

```
>>> n_EM_E = nv.unit(n_EA_E + n_EB_E + n_EC_E)
```

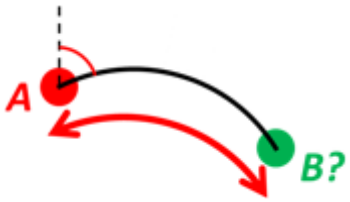
or

```
>>> n_EM_E = nv.mean_horizontal_position(np.hstack((n_EA_E, n_EB_E, n_EC_E)))
```

```
>>> lat, lon = nv.n_E2lat_lon(n_EM_E)
>>> lat, lon = deg(lat), deg(lon)
>>> msg = 'Ex7: Pos M: lat, lon = {:4.4f}, {:4.4f} deg'
>>> msg.format(lat[0], lon[0])
'Ex7: Pos M: lat, lon = 67.2362, -6.9175 deg'
```

See also Example 7 at www.navlab.net[21]

2.3.8 Example 8: "A and azimuth/distance to B"



We have an initial position A, direction of travel given as an azimuth (bearing) relative to north (clockwise), and finally the distance to travel along a great circle given as sAB. Use Earth radius 6371e3 m to find the destination point B.

In geodesy this is known as "The first geodetic problem" or "The direct geodetic problem" for a sphere, and we see that this is similar to Example 2[22], but now the delta is given as an azimuth and a great circle distance. ("The second/inverse geodetic problem" for a sphere is already solved in Examples 1[23] and 5[24].)

Solution:

```
>>> import nvector as nv
>>> from nvector import rad, deg
>>> lat, lon = rad(80), rad(-90)
```

```
>>> n_EA_E = nv.lat_lon2n_E(lat, lon)
>>> azimuth = rad(200)
```

(continues on next page)

[21] http://www.navlab.net/nvector/#example_7
[22] http://www.navlab.net/nvector/#example_2
[23] http://www.navlab.net/nvector/#example_1
[24] http://www.navlab.net/nvector/#example_5

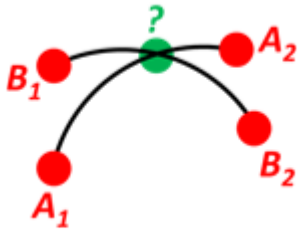
(continued from previous page)

```
>>> s_AB = 1000.0  # [m]
>>> r_earth = 6371e3  # [m], mean earth radius
```

```
>>> distance_rad = s_AB / r_earth
>>> n_EB_E = nv.n_EA_E_distance_and_azimuth2n_EB_E(n_EA_E, distance_rad, azimuth)
>>> lat_EB, lon_EB = nv.n_E2lat_lon(n_EB_E)
>>> lat, lon = deg(lat_EB), deg(lon_EB)
>>> msg = 'Ex8, Destination: lat, lon = {:4.4f} deg, {:4.4f} deg'
>>> msg.format(lat[0], lon[0])
'Ex8, Destination: lat, lon = 79.9915 deg, -90.0177 deg'
```

See also Example 8 at www.navlab.net[25]

2.3.9 Example 9: "Intersection of two paths"



Define a path from two given positions (at the surface of a spherical Earth), as the great circle that goes through the two points.

Path A is given by A1 and A2, while path B is given by B1 and B2.

Find the position C where the two great circles intersect.

Solution:

```
>>> import numpy as np
>>> import nvector as nv
>>> from nvector import rad, deg
```

```
>>> n_EA1_E = nv.lat_lon2n_E(rad(10), rad(20))
>>> n_EA2_E = nv.lat_lon2n_E(rad(30), rad(40))
>>> n_EB1_E = nv.lat_lon2n_E(rad(50), rad(60))
>>> n_EB2_E = nv.lat_lon2n_E(rad(70), rad(80))
```

```
>>> n_EC_E = nv.unit(np.cross(np.cross(n_EA1_E, n_EA2_E, axis=0),
...                           np.cross(n_EB1_E, n_EB2_E, axis=0),
...                           axis=0))
>>> n_EC_E *= np.sign(np.dot(n_EC_E.T, n_EA1_E))
```

or alternatively

```
>>> path_a, path_b = (n_EA1_E, n_EA2_E), (n_EB1_E, n_EB2_E)
>>> n_EC_E = nv.intersect(path_a, path_b)
```

```
>>> lat_EC, lon_EC = nv.n_E2lat_lon(n_EC_E)
```

[25] http://www.navlab.net/nvector/#example_8

```
>>> lat, lon = deg(lat_EC), deg(lon_EC)
>>> msg = 'Ex9, Intersection: lat, lon = {:4.4f}, {:4.4f} deg'
>>> msg.format(lat[0], lon[0])
'Ex9, Intersection: lat, lon = 40.3186, 55.9019 deg'
```

Check that PointC is not between A1 and A2 or B1 and B2:

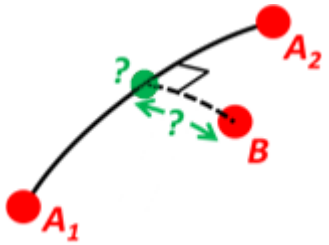
```
>>> np.allclose([nv.on_great_circle_path(path_a, n_EC_E),
...              nv.on_great_circle_path(path_b, n_EC_E)], False)
True
```

Check that PointC is on the great circle going through path A and path B:

```
>>> np.allclose([nv.on_great_circle(path_a, n_EC_E), nv.on_great_circle(path_b,␣
↪n_EC_E)], True)
True
```

See also Example 9 at www.navlab.net[26]

2.3.10 Example 10: "Cross track distance"



Path A is given by the two positions A1 and A2 (similar to the previous example).

Find the cross track distance sxt between the path A (i.e. the great circle through A1 and A2) and the position B (i.e. the shortest distance at the surface, between the great circle and B).

Also find the Euclidean distance dxt between B and the plane defined by the great circle. Use Earth radius 6371e3.

Finally, find the intersection point on the great circle and determine if it is between position A1 and A2.

Solution:

```
>>> import numpy as np
>>> import nvector as nv
>>> from nvector import rad, deg
>>> n_EA1_E = nv.lat_lon2n_E(rad(0), rad(0))
>>> n_EA2_E = nv.lat_lon2n_E(rad(10), rad(0))
>>> n_EB_E = nv.lat_lon2n_E(rad(1), rad(0.1))
>>> path = (n_EA1_E, n_EA2_E)
>>> radius = 6371e3  # mean earth radius [m]
>>> s_xt = nv.cross_track_distance(path, n_EB_E, radius=radius)
>>> d_xt = nv.cross_track_distance(path, n_EB_E, method='euclidean',
...                                radius=radius)
```

```
>>> val_txt = '{:4.2f} km, {:4.2f} km'.format(s_xt[0]/1000, d_xt[0]/1000)
>>> 'Ex10: Cross track distance: s_xt, d_xt = {0}'.format(val_txt)
'Ex10: Cross track distance: s_xt, d_xt = 11.12 km, 11.12 km'
```

[26] http://www.navlab.net/nvector/#example_9

```
>>> n_EC_E = nv.closest_point_on_great_circle(path, n_EB_E)
>>> np.allclose(nv.on_great_circle_path(path, n_EC_E, radius), True)
True
```

Alternative solution 2:

```
>>> s_xt2 = nv.great_circle_distance(n_EB_E, n_EC_E, radius)
>>> d_xt2 = nv.euclidean_distance(n_EB_E, n_EC_E, radius)
>>> np.allclose(s_xt, s_xt2), np.allclose(d_xt, d_xt2)
(True, True)
```

Alternative solution 3:

```
>>> c_E = nv.great_circle_normal(n_EA1_E, n_EA2_E)
>>> sin_theta = -np.dot(c_E.T, n_EB_E).ravel()
>>> s_xt3 = np.arcsin(sin_theta) * radius
>>> d_xt3 = sin_theta * radius
>>> np.allclose(s_xt, s_xt3), np.allclose(d_xt, d_xt3)
(True, True)
```

See also Example 10 at www.navlab.net[27]

2.4 What to read next

So you've read all the *introductory material* and have decided you'd like to keep using nvector. We've only just scratched the surface with this intro.

So what's next?

Well, we've always been big fans of learning by doing. At this point you should know enough to start a project of your own and start fooling around. As you need to learn new tricks, come back to the documentation.

We've put a lot of effort into making nvector's documentation useful, easy to read and as complete as possible. The rest of this document explains more about how the documentation works so that you can get the most out of it.

2.4.1 Finding documentation

The nvector library got a *lot* of documentation, so finding what you need can sometimes be tricky. A few good places to start are the search and the genindex.

Or you can just browse around!

2.4.2 How the documentation is organized

The nvector main documentation is broken up into "chunks" designed to fill different needs:

- The *introductory material* is designed for people new to nvector. It doesn't cover anything in depth, but instead gives a hands on overview of how to use nvector.
- The *topic guides*, on the other hand, dive deep into individual parts of nvector from a theoretical perspective.
- We've written a set of *how-to guides* that answer common "How do I . . . ?" questions.
- The guides and how-to's don't cover every single class, function, and method available in nvector – that would be overwhelming when you're trying to learn. Instead, details about individual classes, functions, methods, and modules are kept in the *reference*. This is where you'll turn to find the details of a particular function or whatever you need.

[27] http://www.navlab.net/nvector/#example_10

2.4.3 How documentation is updated

Just as the nvector code base is developed and improved on a daily basis, our documentation is consistently improving. We improve documentation for several reasons:

- To make content fixes, such as grammar/typo corrections.
- To add information and/or examples to existing sections that need to be expanded.
- To document nvector features that aren't yet documented. (The list of such features is shrinking but exists nonetheless.)
- To add documentation for new features as new features get added, or as nvector APIs or behaviors change.

2.4.3.1 In plain text

For offline reading, or just for convenience, you can read the nvector documentation in plain text.

If you're using an official release of nvector, the zipped package (tarball) of the code includes a `docs/` directory, which contains all the documentation for that release.

If you're using the development version of nvector (aka the master branch), the `docs/` directory contains all of the documentation. You can update your Git checkout to get the latest changes.

One low-tech way of taking advantage of the text documentation is by using the Unix `grep` utility to search for a phrase in all of the documentation. For example, this will show you each mention of the phrase "max_length" in any nvector document:

```
$ grep -r max_length /path/to/nvector/docs/
```

2.4.3.2 As HTML, locally

You can get a local copy of the HTML documentation following a few easy steps:

- nvector's documentation uses a system called Sphinx[28] to convert from plain text to HTML. You'll need to install Sphinx by either downloading and installing the package from the Sphinx website, or with `pip`:

  ```
  $ pip install Sphinx
  ```

- Then, just use the included `Makefile` to turn the documentation into HTML:

  ```
  $ cd path/to/nvector/docs
  $ make html
  ```

  You'll need GNU Make[29] installed for this.

  If you're on Windows you can alternatively use the included batch file:

  ```
  $ cd path\to\nvector\docs
  $ make.bat html
  ```

- The HTML documentation will be placed in `docs/_build/html`.

[28] http://sphinx-doc.org/
[29] https://www.gnu.org/software/make/

2.4.3.3 Using pydoc

The pydoc module automatically generates documentation from Python modules. The documentation can be presented as pages of text on the console, served to a Web browser, or saved to HTML files.

For modules, classes, functions and methods, the displayed documentation is derived from the docstring (i.e. the __doc__ attribute) of the object, and recursively of its documentable members. If there is no docstring, pydoc tries to obtain a description from the block of comment lines just above the definition of the class, function or method in the source file, or at the top of the module (see inspect.getcomments()).

The built-in function help() invokes the online help system in the interactive interpreter, which uses pydoc to generate its documentation as text on the console. The same text documentation can also be viewed from outside the Python interpreter by running pydoc as a script at the operating system's command prompt. For example, running

```
$ pydoc nvector
```

at a shell prompt will display documentation on the nvector module, in a style similar to the manual pages shown by the Unix man command. The argument to pydoc can be the name of a function, module, or package, or a dotted reference to a class, method, or function within a module or module in a package. If the argument to pydoc looks like a path (that is, it contains the path separator for your operating system, such as a slash in Unix), and refers to an existing Python source file, then documentation is produced for that file.

You can also use pydoc to start an HTTP server on the local machine that will serve documentation to visiting Web browsers. For example, running

```
$ pydoc -b
```

will start the server and additionally open a web browser to a module index page. Each served page has a navigation bar at the top where you can Get help on an individual item, Search all modules with a keyword in their synopsis line, and go to the Module index, Topics and Keywords pages. To quit the server just type

```
$ quit
```

See also:

Nvector is 100% Python[30], so if you're new to Python[31], you might want to start by getting an idea of what the language is like. Below we have given some pointers to some resources you can use to get acquainted with the language.

If you're new to programming entirely, you might want to start with this list of Python resources for non-programmers[32]

If you already know a few other languages and want to get up to speed with Python quickly, we recommend Dive Into Python[33]. If that's not quite your style, there are many other books about Python[34].

[30] https://python.org/
[31] https://python.org/
[32] https://wiki.python.org/moin/BeginnersGuide/NonProgrammers
[33] https://www.diveinto.org/python3/
[34] https://wiki.python.org/moin/PythonBooks

CHAPTER

THREE

HOW-TO GUIDES

Here you'll find short answers to "How do I....?" types of questions. These how-to guides don't cover topics in depth – you'll find that material in the *Topics guides* and the *Reference nvector package*. However, these guides will help you quickly accomplish common tasks using the "best practices".

3.1 Contributing

3.1.1 Contribute a patch

CHAPTER

FOUR

TOPICS GUIDES

This section explains and analyses some key concepts in nvector. It's less concerned with explaining *how to do things* than with helping you understand *how it works*.

CHAPTER

FIVE

REFERENCE NVECTOR PACKAGE

Technical reference material that details functions, modules, and objects included in nvector version 0.7.7, describing what they are and what they do.

5.1 Object Oriented interface to Geodesic functions

delta_E(point_a, point_b)	Returns cartesian delta vector from positions a to b decomposed in E.
delta_N(point_a, point_b)	Returns cartesian delta vector from positions a to b decomposed in N.
delta_L(point_a, point_b[, wander_azimuth])	Returns cartesian delta vector from positions a to b decomposed in L.
diff_positions(*args, **kwds)	*diff_positions* is deprecated, use *delta_E* instead!
ECEFvector(pvector[, frame, scalar])	Geographical position given as cartesian position vector in frame E
FrameB(point[, yaw, pitch, roll, degrees])	Body frame
FrameE([a, f, name, axes])	Earth-fixed frame
FrameN(point)	North-East-Down frame
FrameL(point[, wander_azimuth])	Local level, Wander azimuth frame
GeoPath(point_a, point_b)	Geographical path between two positions in Frame E
GeoPoint(latitude, longitude[, z, frame, ...])	Geographical position given as latitude, longitude, depth in frame E.
Nvector(normal[, z, frame])	Geographical position given as n-vector and depth in frame E
Pvector(pvector, frame[, scalar])	Geographical position given as cartesian position vector in a frame.

5.1.1 nvector.objects.delta_E

delta_E(*point_a*, *point_b*)

Returns cartesian delta vector from positions a to b decomposed in E.

Parameters

point_a, point_b: Nvector, GeoPoint or ECEFvector objects position a and b, decomposed in E.

Returns

p_ab_E: ECEFvector Cartesian position vector(s) from a to b, decomposed in E.

See also:

n_EA_E_and_p_AB_E2n_EB_E

p_EB_E2n_EB_E

n_EB_E2p_EB_E.

Notes

The calculation is exact, taking the ellipsity of the Earth into account. It is also non-singular as both n-vector and p-vector are non-singular (except for the center of the Earth).

Examples

Example 1: "A and B to delta"

Given two positions, A and B as latitudes, longitudes and depths relative to Earth, E.

Find the exact vector between the two positions, given in meters north, east, and down, and find the direction (azimuth) to B, relative to north. Assume WGS-84 ellipsoid. The given depths are from the ellipsoid surface. Use position A to define north, east, and down directions. (Due to the curvature of Earth and different directions to the North Pole, the north, east, and down directions will change (relative to Earth) for different places. Position A must be outside the poles for the north and east directions to be defined.)

Solution:

```
>>> import numpy as np
>>> import nvector as nv
>>> wgs84 = nv.FrameE(name='WGS84')
>>> pointA = wgs84.GeoPoint(latitude=1, longitude=2, z=3, degrees=True)
>>> pointB = wgs84.GeoPoint(latitude=4, longitude=5, z=6, degrees=True)
```

Step1: Find p_AB_N (delta decomposed in N).

```
>>> p_AB_N = pointA.delta_to(pointB)
>>> x, y, z = p_AB_N.pvector.ravel()
>>> 'Ex1: delta north, east, down = {0:8.2f}, {1:8.2f}, {2:8.2f}'.format(x,␣
↪y, z)
'Ex1: delta north, east, down = 331730.23, 332997.87, 17404.27'
```

Step2: Also find the direction (azimuth) to B, relative to north:

```
>>> 'azimuth = {0:4.2f} deg'.format(p_AB_N.azimuth_deg)
'azimuth = 45.11 deg'
>>> 'elevation = {0:4.2f} deg'.format(p_AB_N.elevation_deg)
'elevation = 2.12 deg'
>>> 'distance = {0:4.2f} m'.format(p_AB_N.length)
'distance = 470356.72 m'
```

5.1.2 nvector.objects.delta_N

delta_N(*point_a*, *point_b*)

Returns cartesian delta vector from positions a to b decomposed in N.

Parameters

point_a, point_b: Nvector, GeoPoint or ECEFvector objects position a and b, decomposed in E.

See also:

delta_E, *delta_L*

5.1.3 nvector.objects.delta_L

delta_L(*point_a*, *point_b*, *wander_azimuth=0*)

Returns cartesian delta vector from positions a to b decomposed in L.

Parameters

point_a, point_b: Nvector, GeoPoint or ECEFvector objects position a and b, decomposed in E.

wander_azimuth: real scalar Angle [rad] between the x-axis of L and the north direction.

See also:

delta_E, *delta_N*

5.1.4 nvector.objects.diff_positions

diff_positions(**args*, ***kwds*)

diff_positions is deprecated, use *delta_E* instead!

Returns cartesian delta vector from positions a to b decomposed in E.

Parameters

point_a, point_b: Nvector, GeoPoint or ECEFvector objects position a and b, decomposed in E.

Returns

p_ab_E: ECEFvector Cartesian position vector(s) from a to b, decomposed in E.

See also:

n_EA_E_and_p_AB_E2n_EB_E

p_EB_E2n_EB_E

n_EB_E2p_EB_E.

Notes

The calculation is exact, taking the ellipsity of the Earth into account. It is also non-singular as both n-vector and p-vector are non-singular (except for the center of the Earth).

Examples

Example 1: "A and B to delta"

Given two positions, A and B as latitudes, longitudes and depths relative to Earth, E.

Find the exact vector between the two positions, given in meters north, east, and down, and find the direction (azimuth) to B, relative to north. Assume WGS-84 ellipsoid. The given depths are from the ellipsoid surface. Use position A to define north, east, and down directions. (Due to the curvature of Earth and different directions to the North Pole, the north, east, and down directions will change (relative to Earth) for different places. Position A must be outside the poles for the north and east directions to be defined.)

Solution:

```
>>> import numpy as np
>>> import nvector as nv
>>> wgs84 = nv.FrameE(name='WGS84')
>>> pointA = wgs84.GeoPoint(latitude=1, longitude=2, z=3, degrees=True)
>>> pointB = wgs84.GeoPoint(latitude=4, longitude=5, z=6, degrees=True)
```

Step1: Find p_AB_N (delta decomposed in N).

```
>>> p_AB_N = pointA.delta_to(pointB)
>>> x, y, z = p_AB_N.pvector.ravel()
>>> 'Ex1: delta north, east, down = {0:8.2f}, {1:8.2f}, {2:8.2f}'.format(x,␣
↪y, z)
'Ex1: delta north, east, down = 331730.23, 332997.87, 17404.27'
```

Step2: Also find the direction (azimuth) to B, relative to north:

```
>>> 'azimuth = {0:4.2f} deg'.format(p_AB_N.azimuth_deg)
'azimuth = 45.11 deg'
>>> 'elevation = {0:4.2f} deg'.format(p_AB_N.elevation_deg)
'elevation = 2.12 deg'
>>> 'distance = {0:4.2f} m'.format(p_AB_N.length)
'distance = 470356.72 m'
```

5.1.5 nvector.objects.ECEFvector

class ECEFvector(*pvector*, *frame=None*, *scalar=None*)

Geographical position given as cartesian position vector in frame E

Parameters

pvector: 3 x n array Cartesian position vector(s) [m] from E to B, decomposed in E.

frame: FrameE object reference ellipsoid. The default ellipsoid model used is WGS84, but other ellipsoids/spheres might be specified.

See also:

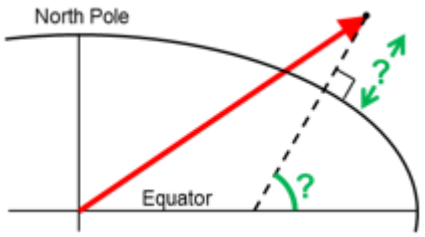
GeoPoint, *ECEFvector*, *Pvector*

Notes

The position of B (typically body) relative to E (typically Earth) is given into this function as p-vector, p_EB_E relative to the center of the frame.

Examples

Example 3: "ECEF-vector to geodetic latitude"



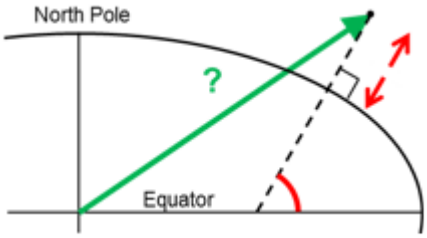
Position B is given as an "ECEF-vector" p_EB_E (i.e. a vector from E, the center of the Earth, to B, decomposed in E). Find the geodetic latitude, longitude and height (latEB, lonEB and hEB), assuming WGS-84 ellipsoid.

Solution:

```
>>> import numpy as np
>>> import nvector as nv
>>> wgs84 = nv.FrameE(name='WGS84')
>>> position_B = 6371e3 * np.vstack((0.9, -1, 1.1))  # m
>>> p_EB_E = wgs84.ECEFvector(position_B)
>>> pointB = p_EB_E.to_geo_point()
```

```
>>> lat, lon, z = pointB.latlon_deg
>>> 'Ex3: Pos B: lat, lon = {:4.4f}, {:4.4f} deg, height = {:9.3f} m'.
↪format(lat, lon, -z)
'Ex3: Pos B: lat, lon = 39.3787, -48.0128 deg, height = 4702059.834 m'
```

Example 4: "Geodetic latitude to ECEF-vector"



Geodetic latitude, longitude and height are given for position B as latEB, lonEB and hEB, find the ECEF-vector for this position, p_EB_E.

Solution:

```
>>> import nvector as nv
>>> wgs84 = nv.FrameE(name='WGS84')
>>> pointB = wgs84.GeoPoint(latitude=1, longitude=2, z=-3, degrees=True)
>>> p_EB_E = pointB.to_ecef_vector()
```

```
>>> 'Ex4: p_EB_E = {} m'.format(p_EB_E.pvector.ravel().tolist())
'Ex4: p_EB_E = [6373290.277218279, 222560.20067473652, 110568.82718178593] m'
```

__init__(*pvector*, *frame=None*, *scalar=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(pvector[, frame, scalar])	Initialize self.
change_frame(frame)	Converts to Cartesian position vector in another frame
delta_to(other)	Returns cartesian delta vector from positions a to b decomposed in N.
to_ecef_vector()	Returns position as ECEFvector object.
to_geo_point()	Returns position as GeoPoint object.
to_nvector()	Returns position as Nvector object.

Attributes

azimuth	Azimuth in radian clockwise relative to the x-axis.
azimuth_deg	Azimuth in degree clockwise relative to the x-axis.
elevation	Elevation in radian relative to the xy-plane.
elevation_deg	Elevation in degree relative to the xy-plane.
length	Length of the pvector.

5.1.6 nvector.objects.FrameB

class FrameB(*point*, *yaw=0*, *pitch=0*, *roll=0*, *degrees=False*)
Body frame

Parameters

point: ECEFvector, GeoPoint or Nvector object position of the vehicle's reference point which also coincides with the origin of the frame B.

yaw, pitch, roll: real scalars defining the orientation of frame B in [deg] or [rad].

degrees [bool] if True yaw, pitch, roll are given in degrees otherwise in radians

See also:

FrameE, *FrameL*, *FrameN*

Notes

The frame is fixed to the vehicle where the x-axis points forward, the y-axis to the right (starboard) and the z-axis in the vehicle's down direction.

Examples

Example 2: "B and delta to C"

A radar or sonar attached to a vehicle B (Body coordinate frame) measures the distance and direction to an object C. We assume that the distance and two angles (typically bearing and elevation relative to B) are already combined to the vector p_BC_B (i.e. the vector from B to C, decomposed in B). The position of B is given as n_EB_E and z_EB, and the orientation (attitude) of B is given as R_NB (this rotation matrix can be found from roll/pitch/yaw by using zyx2R).

Find the exact position of object C as n-vector and depth (n_EC_E and z_EC), assuming Earth ellipsoid with semi-major axis a and flattening f. For WGS-72, use a = 6 378 135 m and f = 1/298.26.

Solution:

```
>>> import numpy as np
>>> import nvector as nv
>>> wgs72 = nv.FrameE(name='WGS72')
>>> wgs72 = nv.FrameE(a=6378135, f=1.0/298.26)
```

Step 1: Position and orientation of B is given 400m above E:

```
>>> n_EB_E = wgs72.Nvector(nv.unit([[1], [2], [3]]), z=-400)
>>> frame_B = nv.FrameB(n_EB_E, yaw=10, pitch=20, roll=30, degrees=True)
```

Step 2: Delta BC decomposed in B

```
>>> p_BC_B = frame_B.Pvector(np.r_[3000, 2000, 100].reshape((-1, 1)))
```

Step 3: Decompose delta BC in E

```
>>> p_BC_E = p_BC_B.to_ecef_vector()
```

Step 4: Find point C by adding delta BC to EB

```
>>> p_EB_E = n_EB_E.to_ecef_vector()
>>> p_EC_E = p_EB_E + p_BC_E
>>> pointC = p_EC_E.to_geo_point()
```

```
>>> lat, lon, z = pointC.latlon_deg
>>> msg = 'Ex2: PosC: lat, lon = {:4.4f}, {:4.4f} deg,  height = {:4.2f} m'
>>> msg.format(lat, lon, -z)
'Ex2: PosC: lat, lon = 53.3264, 63.4681 deg,  height = 406.01 m'
```

__init__(*point*, *yaw=0*, *pitch=0*, *roll=0*, *degrees=False*)
Initialize self. See help(type(self)) for accurate signature.

Methods

Pvector(pvector)	Returns Pvector relative to the local frame.
__init__(point[, yaw, pitch, roll, degrees])	Initialize self.

Attributes

R_EN	Rotation matrix to go between E and B frame

5.1.7 nvector.objects.FrameE

class FrameE(*a=None*, *f=None*, *name='WGS84'*, *axes='e'*)
Earth-fixed frame

Parameters

a: real scalar, default WGS-84 ellipsoid. Semi-major axis of the Earth ellipsoid given in [m].

f: real scalar, default WGS-84 ellipsoid. Flattening [no unit] of the Earth ellipsoid. If f==0 then spherical Earth with radius a is used in stead of WGS-84.

name: string defining the default ellipsoid.

axes: 'e' or 'E' defines axes orientation of E frame. Default is axes='e' which means that the orientation of the axis is such that: z-axis -> North Pole, x-axis -> Latitude=Longitude=0.

See also:

FrameN, *FrameL*, *FrameB*

Notes

The frame is Earth-fixed (rotates and moves with the Earth) where the origin coincides with Earth's centre (geometrical centre of ellipsoid model).

__init__(*a=None*, *f=None*, *name='WGS84'*, *axes='e'*)
Initialize self. See help(type(self)) for accurate signature.

Methods

ECEFvector(*args, **kwds)	Geographical position given as cartesian position vector in frame E
GeoPoint(*args, **kwds)	Geographical position given as latitude, longitude, depth in frame E.
Nvector(*args, **kwds)	Geographical position given as n-vector and depth in frame E
__init__([a, f, name, axes])	Initialize self.
direct(lat_a, lon_a, azimuth, distance[, z, ...])	Returns position B computed from position A, distance and azimuth.
inverse(lat_a, lon_a, lat_b, lon_b[, z, ...])	Returns ellipsoidal distance between positions as well as the direction.

Attributes

R_Ee	Rotation matrix R_Ee defining the axes of the coordinate frame E

5.1.8 nvector.objects.FrameN

class FrameN(*point*)

North-East-Down frame

Parameters

point: ECEFvector, GeoPoint or Nvector object position of the vehicle (B) which also defines the origin of the local frame N. The origin is directly beneath or above the vehicle (B), at Earth's surface (surface of ellipsoid model).

See also:

FrameE, *FrameL*, *FrameB*

Notes

The Cartesian frame is local and oriented North-East-Down, i.e., the x-axis points towards north, the y-axis points towards east (both are horizontal), and the z-axis is pointing down.

When moving relative to the Earth, the frame rotates about its z-axis to allow the x-axis to always point towards north. When getting close to the poles this rotation rate will increase, being infinite at the poles. The poles are thus singularities and the direction of the x- and y-axes are not defined here. Hence, this coordinate frame is NOT SUITABLE for general calculations.

Examples

Example 1: "A and B to delta"

Given two positions, A and B as latitudes, longitudes and depths relative to Earth, E.

Find the exact vector between the two positions, given in meters north, east, and down, and find the direction (azimuth) to B, relative to north. Assume WGS-84 ellipsoid. The given depths are from the ellipsoid surface. Use position A to define north, east, and down directions. (Due to the curvature of Earth and different directions to the North Pole, the north, east, and down directions will change (relative to Earth) for different places. Position A must be outside the poles for the north and east directions to be defined.)

Solution:

```
>>> import numpy as np
>>> import nvector as nv
>>> wgs84 = nv.FrameE(name='WGS84')
>>> pointA = wgs84.GeoPoint(latitude=1, longitude=2, z=3, degrees=True)
>>> pointB = wgs84.GeoPoint(latitude=4, longitude=5, z=6, degrees=True)
```

Step1: Find p_AB_N (delta decomposed in N).

```
>>> p_AB_N = pointA.delta_to(pointB)
>>> x, y, z = p_AB_N.pvector.ravel()
>>> 'Ex1: delta north, east, down = {0:8.2f}, {1:8.2f}, {2:8.2f}'.format(x,␣
↪y, z)
'Ex1: delta north, east, down = 331730.23, 332997.87, 17404.27'
```

Step2: Also find the direction (azimuth) to B, relative to north:

```
>>> 'azimuth = {0:4.2f} deg'.format(p_AB_N.azimuth_deg)
'azimuth = 45.11 deg'
>>> 'elevation = {0:4.2f} deg'.format(p_AB_N.elevation_deg)
'elevation = 2.12 deg'
>>> 'distance = {0:4.2f} m'.format(p_AB_N.length)
'distance = 470356.72 m'
```

__init__(*point*)
Initialize self. See help(type(self)) for accurate signature.

Methods

Pvector(pvector)	Returns Pvector relative to the local frame.
__init__(point)	Initialize self.

Attributes

R_EN	Rotation matrix to go between E and N frame

5.1.9 nvector.objects.FrameL

class FrameL(*point*, *wander_azimuth=0*)

Local level, Wander azimuth frame

Parameters

point: ECEFvector, GeoPoint or Nvector object position of the vehicle (B) which also defines the origin of the local frame L. The origin is directly beneath or above the vehicle (B), at Earth's surface (surface of ellipsoid model).

wander_azimuth: real scalar Angle [rad] between the x-axis of L and the north direction.

See also:

FrameE, *FrameN*, *FrameB*

Notes

The Cartesian frame is local and oriented Wander-azimuth-Down. This means that the z-axis is pointing down. Initially, the x-axis points towards north, and the y-axis points towards east, but as the vehicle moves they are not rotating about the z-axis (their angular velocity relative to the Earth has zero component along the z-axis).

(Note: Any initial horizontal direction of the x- and y-axes is valid for L, but if the initial position is outside the poles, north and east are usually chosen for convenience.)

The L-frame is equal to the N-frame except for the rotation about the z-axis, which is always zero for this frame (relative to E). Hence, at a given time, the only difference between the frames is an angle between the x-axis of L and the north direction; this angle is called the wander azimuth angle. The L-frame is well suited for general calculations, as it is non-singular.

__init__(*point*, *wander_azimuth=0*)

Initialize self. See help(type(self)) for accurate signature.

Methods

Pvector(pvector)	Returns Pvector relative to the local frame.
__init__(point[, wander_azimuth])	Initialize self.

Attributes

R_EN	Rotation matrix to go between E and L frame

5.1.10 nvector.objects.GeoPath

class GeoPath(*point_a*, *point_b*)

Geographical path between two positions in Frame E

Parameters

point_a, point_b: Nvector, GeoPoint or ECEFvector objects The path is defined by the line between position A and B, decomposed in E.

Notes

Please note that either position A or B or both might be a vector of points. In this case the GeoPath instance represents all the paths between the positions of A and the corresponding positions of B.

Examples

Example 5: "Surface distance"

Find the surface distance sAB (i.e. great circle distance) between two positions A and B. The heights of A and B are ignored, i.e. if they don't have zero height, we seek the distance between the points that are at the surface of the Earth, directly above/below A and B. The Euclidean distance (chord length) dAB should also be found. Use Earth radius 6371e3 m. Compare the results with exact calculations for the WGS-84 ellipsoid.

Solution for a sphere:

```
>>> import numpy as np
>>> import nvector as nv
>>> frame_E = nv.FrameE(a=6371e3, f=0)
>>> positionA = frame_E.GeoPoint(latitude=88, longitude=0, degrees=True)
>>> positionB = frame_E.GeoPoint(latitude=89, longitude=-170, degrees=True)
```

```
>>> s_AB, azia, azib = positionA.distance_and_azimuth(positionB)
>>> p_AB_E = positionB.to_ecef_vector() - positionA.to_ecef_vector()
>>> d_AB = p_AB_E.length
```

```
>>> msg = 'Ex5: Great circle and Euclidean distance = {}'
>>> msg = msg.format('{:5.2f} km, {:5.2f} km')
>>> msg.format(s_AB / 1000, d_AB / 1000)
'Ex5: Great circle and Euclidean distance = 332.46 km, 332.42 km'
```

Alternative sphere solution:

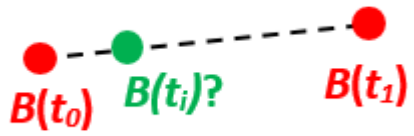
```
>>> path = nv.GeoPath(positionA, positionB)
>>> s_AB2 = path.track_distance(method='greatcircle')
>>> d_AB2 = path.track_distance(method='euclidean')
>>> msg.format(s_AB2 / 1000, d_AB2 / 1000)
'Ex5: Great circle and Euclidean distance = 332.46 km, 332.42 km'
```

Exact solution for the WGS84 ellipsoid:

```
>>> wgs84 = nv.FrameE(name='WGS84')
>>> point1 = wgs84.GeoPoint(latitude=88, longitude=0, degrees=True)
>>> point2 = wgs84.GeoPoint(latitude=89, longitude=-170, degrees=True)
>>> s_12, azi1, azi2 = point1.distance_and_azimuth(point2)
```

```
>>> p_12_E = point2.to_ecef_vector() - point1.to_ecef_vector()
>>> d_12 = p_12_E.length
>>> msg = 'Ellipsoidal and Euclidean distance = {:5.2f} km, {:5.2f} km'
>>> msg.format(s_12 / 1000, d_12 / 1000)
'Ellipsoidal and Euclidean distance = 333.95 km, 333.91 km'
```

Example 6 "Interpolated position"



Given the position of B at time t0 and t1, n_EB_E(t0) and n_EB_E(t1).

Find an interpolated position at time ti, n_EB_E(ti). All positions are given as n-vectors.

Solution:

```
>>> import nvector as nv
>>> wgs84 = nv.FrameE(name='WGS84')
>>> n_EB_E_t0 = wgs84.GeoPoint(89, 0, degrees=True).to_nvector()
>>> n_EB_E_t1 = wgs84.GeoPoint(89, 180, degrees=True).to_nvector()
>>> path = nv.GeoPath(n_EB_E_t0, n_EB_E_t1)
```

```
>>> t0 = 10.
>>> t1 = 20.
>>> ti = 16.  # time of interpolation
>>> ti_n = (ti - t0) / (t1 - t0) # normalized time of interpolation
```

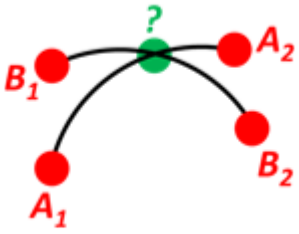
```
>>> g_EB_E_ti = path.interpolate(ti_n).to_geo_point()
```

```
>>> lat_ti, lon_ti, z_ti = g_EB_E_ti.latlon_deg
>>> msg = 'Ex6, Interpolated position: lat, lon = {:2.1f} deg, {:2.1f} deg'
>>> msg.format(lat_ti, lon_ti)
'Ex6, Interpolated position: lat, lon = 89.8 deg, 180.0 deg'
```

Vectorized solution:

```
>>> t = np.array([10, 20])
>>> nvectors = wgs84.GeoPoint([89, 89], [0, 180], degrees=True).to_nvector()
>>> nvectors_i = nvectors.interpolate(ti, t, kind='linear')
>>> lati, loni, zi = nvectors_i.to_geo_point().latlon_deg
>>> msg.format(lat_ti, lon_ti)
'Ex6, Interpolated position: lat, lon = 89.8 deg, 180.0 deg'
```

Example 9: "Intersection of two paths"



Define a path from two given positions (at the surface of a spherical Earth), as the great circle that goes through the two points.

Path A is given by A1 and A2, while path B is given by B1 and B2.

Find the position C where the two great circles intersect.

Solution:

```
>>> import nvector as nv
>>> pointA1 = nv.GeoPoint(10, 20, degrees=True)
>>> pointA2 = nv.GeoPoint(30, 40, degrees=True)
>>> pointB1 = nv.GeoPoint(50, 60, degrees=True)
>>> pointB2 = nv.GeoPoint(70, 80, degrees=True)
>>> pathA = nv.GeoPath(pointA1, pointA2)
>>> pathB = nv.GeoPath(pointB1, pointB2)
```

```
>>> pointC = pathA.intersect(pathB)
>>> pointC = pointC.to_geo_point()
>>> lat, lon = pointC.latitude_deg, pointC.longitude_deg
>>> msg = 'Ex9, Intersection: lat, lon = {:4.4f}, {:4.4f} deg'
>>> msg.format(lat, lon)
'Ex9, Intersection: lat, lon = 40.3186, 55.9019 deg'
```

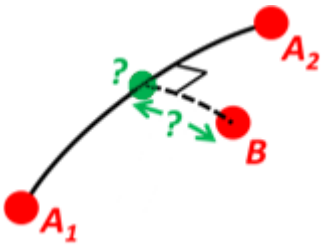
Check that PointC is not between A1 and A2 or B1 and B2:

```
>>> pathA.on_path(pointC)
False
>>> pathB.on_path(pointC)
False
```

Check that PointC is on the great circle going through path A and path B:

```
>>> pathA.on_great_circle(pointC)
True
>>> pathB.on_great_circle(pointC)
True
```

Example 10: "Cross track distance"



Path A is given by the two positions A1 and A2 (similar to the previous example).

Find the cross track distance sxt between the path A (i.e. the great circle through A1 and A2) and the position B (i.e. the shortest distance at the surface, between the great circle and B).

Also find the Euclidean distance dxt between B and the plane defined by the great circle. Use Earth radius 6371e3.

Finally, find the intersection point on the great circle and determine if it is between position A1 and A2.

Solution:

```
>>> import numpy as np
>>> import nvector as nv
>>> frame = nv.FrameE(a=6371e3, f=0)
>>> pointA1 = frame.GeoPoint(0, 0, degrees=True)
>>> pointA2 = frame.GeoPoint(10, 0, degrees=True)
>>> pointB = frame.GeoPoint(1, 0.1, degrees=True)
>>> pathA = nv.GeoPath(pointA1, pointA2)
```

```
>>> s_xt = pathA.cross_track_distance(pointB, method='greatcircle')
>>> d_xt = pathA.cross_track_distance(pointB, method='euclidean')
```

```
>>> val_txt = '{:4.2f} km, {:4.2f} km'.format(s_xt/1000, d_xt/1000)
>>> 'Ex10: Cross track distance: s_xt, d_xt = {}'.format(val_txt)
'Ex10: Cross track distance: s_xt, d_xt = 11.12 km, 11.12 km'
```

```
>>> pointC = pathA.closest_point_on_great_circle(pointB)
>>> np.allclose(pathA.on_path(pointC), True)
True
```

__init__(*point_a*, *point_b*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(point_a, point_b)	Initialize self.
closest_point_on_great_circle(point)	Returns closest point on great circle path to the point.
closest_point_on_path(point)	Returns closest point on great circle path segment to the point.
cross_track_distance(point[, method, radius])	Returns cross track distance from path to point.
ecef_vectors()	Returns point_a and point_b as ECEF-vectors
geo_points()	Returns point_a and point_b as geo-points
interpolate(ti)	Returns the interpolated point along the path
intersect(path)	Returns the intersection(s) between the great circles of the two paths
intersection(**kwds)	*intersection* is deprecated, use *intersect* instead!
nvector_normals()	Returns nvector normals for position a and b
nvectors()	Returns point_a and point_b as n-vectors
on_great_circle(point[, atol])	Returns True if point is on the great circle within a tolerance.
on_path(point[, method, rtol, atol])	Returns True if point is on the path between A and B witin a tolerance.
track_distance([method, radius])	Returns the path distance computed at the average height.

Attributes

positionA	positionA is deprecated, use point_a instead!
positionB	positionB is deprecated, use point_b instead!

5.1.11 nvector.objects.GeoPoint

class GeoPoint(*latitude*, *longitude*, *z=0*, *frame=None*, *degrees=False*)

Geographical position given as latitude, longitude, depth in frame E.

Parameters

latitude, longitude: real scalars or vectors of length n. Geodetic latitude and longitude given in [rad or deg]

z: real scalar or vector of length n. Depth(s) [m] relative to the ellipsoid (depth = -height)

frame: FrameE object reference ellipsoid. The default ellipsoid model used is WGS84, but other ellipsoids/spheres might be specified.

degrees: bool True if input are given in degrees otherwise radians are assumed.

Examples

Solve geodesic problems.

The following illustrates its use

```
>>> import nvector as nv
>>> wgs84 = nv.FrameE(name='WGS84')
>>> point_a = wgs84.GeoPoint(-41.32, 174.81, degrees=True)
>>> point_b = wgs84.GeoPoint(40.96, -5.50, degrees=True)
```

```
>>> print(point_a)
GeoPoint(latitude=-0.721170046924057,
         longitude=3.0510100654112877,
         z=0,
        frame=FrameE(a=6378137.0,
                     f=0.0033528106647474805,
                     name='WGS84',
                     axes='e'))
```

The geodesic inverse problem

```
>>> s12, az1, az2 = point_a.distance_and_azimuth(point_b, degrees=True)
>>> 's12 = {:5.2f}, az1 = {:5.2f}, az2 = {:5.2f}'.format(s12, az1, az2)
's12 = 19959679.27, az1 = 161.07, az2 = 18.83'
```

The geodesic direct problem

```
>>> point_a = wgs84.GeoPoint(40.6, -73.8, degrees=True)
>>> az1, distance = 45, 10000e3
>>> point_b, az2 = point_a.displace(distance, az1, degrees=True)
>>> lat2, lon2 = point_b.latitude_deg, point_b.longitude_deg
>>> msg = 'lat2 = {:5.2f}, lon2 = {:5.2f}, az2 = {:5.2f}'
>>> msg.format(lat2, lon2, az2)
'lat2 = 32.64, lon2 = 49.01, az2 = 140.37'
```

__init__(*latitude*, *longitude*, *z=0*, *frame=None*, *degrees=False*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(latitude, longitude[, z, frame, ...])	Initialize self.
delta_to(other)	Returns cartesian delta vector from positions a to b decomposed in N.
displace(distance, azimuth[, long_unroll, ...])	Returns position b computed from current position, distance and azimuth.
distance_and_azimuth(point[, long_unroll, ...])	Returns ellipsoidal distance between positions as well as the direction.
to_ecef_vector()	Returns position as ECEFvector object.
to_geo_point()	Returns position as GeoPoint object.
to_nvector()	Returns position as Nvector object.

Attributes

latitude_deg	latitude in degrees.
latlon	(latitude, longitude, z) tuple, angles are in radian.
latlon_deg	(latitude_deg, longitude_deg, z) tuple, angles are in degree.
longitude_deg	longitude in degrees.
scalar	True if the position is a scalar point

5.1.12 nvector.objects.Nvector

class Nvector(*normal*, *z=0*, *frame=None*)
Geographical position given as n-vector and depth in frame E

Parameters

normal: 3 x n array n-vector(s) [no unit] decomposed in E.

z: real scalar or vector of length n. Depth(s) [m] relative to the ellipsoid (depth = -height)

frame: FrameE object reference ellipsoid. The default ellipsoid model used is WGS84, but other ellipsoids/spheres might be specified.

See also:

GeoPoint, *ECEFvector*, *Pvector*

Notes

The position of B (typically body) relative to E (typically Earth) is given into this function as n-vector, n_EB_E and a depth, z relative to the ellipsiod.

Examples

```
>>> import nvector as nv
>>> wgs84 = nv.FrameE(name='WGS84')
>>> point_a = wgs84.GeoPoint(-41.32, 174.81, degrees=True)
>>> point_b = wgs84.GeoPoint(40.96, -5.50, degrees=True)
>>> nv_a = point_a.to_nvector()
>>> print(nv_a)
Nvector(normal=[[-0.7479546170813224], [0.06793758070955484], [-0.
↪66026386839964 61]],
        z=0,
        frame=FrameE(a=6378137.0,
                     f=0.0033528106647474805,
                     name='WGS84',
                     axes='e'))
```

__init__(*normal*, *z=0*, *frame=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(normal[, z, frame])	Initialize self.
delta_to(other)	Returns cartesian delta vector from positions a to b decomposed in N.
interpolate(t_i, t[, kind, window_length, ...])	Returns interpolated values from nvector data.
mean()	Returns mean position of the n-vectors.
mean_horizontal_position(**kwds)	*mean_horizontal_position* is deprecated, use *mean* instead!
to_ecef_vector()	Returns position as ECEFvector object.
to_geo_point()	Returns position as GeoPoint object.
to_nvector()	Returns position as Nvector object.
unit()	Normalizes self to unit vector(s)

Attributes

scalar	True if the position is a scalar point

5.1.13 nvector.objects.Pvector

class Pvector(*pvector*, *frame*, *scalar=None*)
Geographical position given as cartesian position vector in a frame.

__init__(*pvector*, *frame*, *scalar=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(pvector, frame[, scalar])	Initialize self.
delta_to(other)	Returns cartesian delta vector from positions a to b decomposed in N.
to_ecef_vector()	Returns position as ECEFvector object.
to_geo_point()	Returns position as GeoPoint object.
to_nvector()	Returns position as Nvector object.

Attributes

azimuth	Azimuth in radian clockwise relative to the x-axis.
azimuth_deg	Azimuth in degree clockwise relative to the x-axis.
elevation	Elevation in radian relative to the xy-plane.
elevation_deg	Elevation in degree relative to the xy-plane.
length	Length of the pvector.

5.2 Geodesic functions

closest_point_on_great_circle(path, n_EB_E)	Returns closest point C on great circle path A to position B.
cross_track_distance(path, n_EB_E[, method, ...])	Returns cross track distance between path A and position B.
euclidean_distance(n_EA_E, n_EB_E[, radius])	Returns Euclidean distance between positions A and B
great_circle_distance(n_EA_E, n_EB_E[, radius])	Returns great circle distance between positions A and B
great_circle_normal(n_EA_E, n_EB_E)	Returns the unit normal(s) to the great circle(s)
interp_nvectors(t_i, t, nvectors[, kind, ...])	Returns interpolated values from nvector data.
interpolate(path, ti)	Returns the interpolated point along the path
intersect(path_a, path_b)	Returns the intersection(s) between the great circles of the two paths
lat_lon2n_E(latitude, longitude[, R_Ee])	Converts latitude and longitude to n-vector.
mean_horizontal_position(n_EB_E)	Returns the n-vector of the horizontal mean position.
n_E2lat_lon(n_E[, R_Ee])	Converts n-vector to latitude and longitude.
n_EB_E2p_EB_E(n_EB_E[, depth, a, f, R_Ee])	Converts n-vector to Cartesian position vector in meters.
p_EB_E2n_EB_E(p_EB_E[, a, f, R_Ee])	Converts Cartesian position vector in meters to n-vector.
n_EA_E_and_n_EB_E2p_AB_E(n_EA_E, n_EB_E[, ...])	Returns the delta vector from position A to B decomposed in E.
n_EA_E_and_p_AB_E2n_EB_E(n_EA_E, p_AB_E[, ...])	Returns position B from position A and delta E vector.
n_EA_E_and_n_EB_E2azimuth(n_EA_E, n_EB_E[, ...])	Returns azimuth from A to B, relative to North:
n_EA_E_distance_and_azimuth2n_EB_E(n_EA_E, ...)	Returns position B from azimuth and distance from position A
on_great_circle(path, n_EB_E[, radius, atol])	Returns True if position B is on great circle through path A.
on_great_circle_path(path, n_EB_E[, radius, ...])	Returns True if position B is on great circle and between endpoints of path A.

5.2.1 nvector.core.closest_point_on_great_circle

closest_point_on_great_circle(*path*, *n_EB_E*)

Returns closest point C on great circle path A to position B.

Parameters

path: tuple of 2 n-vectors of 3 x n arrays 2 n-vectors of positions defining path A, decomposed in E.

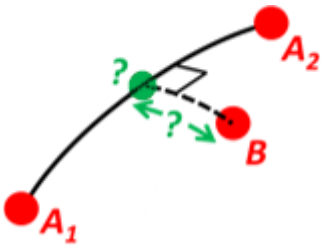
n_EB_E: 3 x m array n-vector(s) of position B to find the closest point to.

Returns

n_EC_E: 3 x max(m, n) array n-vector(s) of closest position C on great circle path A

Examples

Example 10: "Cross track distance"



Path A is given by the two positions A1 and A2 (similar to the previous example).

Find the cross track distance sxt between the path A (i.e. the great circle through A1 and A2) and the position B (i.e. the shortest distance at the surface, between the great circle and B).

Also find the Euclidean distance dxt between B and the plane defined by the great circle. Use Earth radius 6371e3.

Finally, find the intersection point on the great circle and determine if it is between position A1 and A2.

Solution:

```
>>> import numpy as np
>>> import nvector as nv
>>> from nvector import rad, deg
>>> n_EA1_E = nv.lat_lon2n_E(rad(0), rad(0))
>>> n_EA2_E = nv.lat_lon2n_E(rad(10), rad(0))
>>> n_EB_E = nv.lat_lon2n_E(rad(1), rad(0.1))
>>> path = (n_EA1_E, n_EA2_E)
>>> radius = 6371e3  # mean earth radius [m]
>>> s_xt = nv.cross_track_distance(path, n_EB_E, radius=radius)
>>> d_xt = nv.cross_track_distance(path, n_EB_E, method='euclidean',
...                                radius=radius)
```

```
>>> val_txt = '{:4.2f} km, {:4.2f} km'.format(s_xt[0]/1000, d_xt[0]/1000)
>>> 'Ex10: Cross track distance: s_xt, d_xt = {0}'.format(val_txt)
'Ex10: Cross track distance: s_xt, d_xt = 11.12 km, 11.12 km'
```

```
>>> n_EC_E = nv.closest_point_on_great_circle(path, n_EB_E)
>>> np.allclose(nv.on_great_circle_path(path, n_EC_E, radius), True)
True
```

Alternative solution 2:

```
>>> s_xt2 = nv.great_circle_distance(n_EB_E, n_EC_E, radius)
>>> d_xt2 = nv.euclidean_distance(n_EB_E, n_EC_E, radius)
>>> np.allclose(s_xt, s_xt2), np.allclose(d_xt, d_xt2)
(True, True)
```

Alternative solution 3:

```
>>> c_E = nv.great_circle_normal(n_EA1_E, n_EA2_E)
>>> sin_theta = -np.dot(c_E.T, n_EB_E).ravel()
>>> s_xt3 = np.arcsin(sin_theta) * radius
>>> d_xt3 = sin_theta * radius
>>> np.allclose(s_xt, s_xt3), np.allclose(d_xt, d_xt3)
(True, True)
```

5.2.2 nvector.core.cross_track_distance

cross_track_distance(*path*, *n_EB_E*, *method='greatcircle'*, *radius=6371009.0*)

Returns cross track distance between path A and position B.

Parameters

path: tuple of 2 n-vectors 2 n-vectors of positions defining path A, decomposed in E.

n_EB_E: 3 x m array n-vector(s) of position B to measure the cross track distance to.

method: string defining distance calculated. Options are: 'greatcircle' or 'euclidean'

radius: real scalar radius of sphere. (default 6371009.0)

Returns

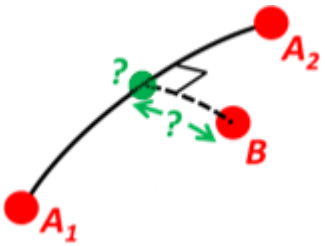
distance [array of length max(n, m)] cross track distance(s)

Notes

The result for spherical Earth is returned.

Examples

Example 10: "Cross track distance"



Path A is given by the two positions A1 and A2 (similar to the previous example).

Find the cross track distance sxt between the path A (i.e. the great circle through A1 and A2) and the position B (i.e. the shortest distance at the surface, between the great circle and B).

Also find the Euclidean distance dxt between B and the plane defined by the great circle. Use Earth radius 6371e3.

Finally, find the intersection point on the great circle and determine if it is between position A1 and A2.

Solution:

```
>>> import numpy as np
>>> import nvector as nv
>>> from nvector import rad, deg
>>> n_EA1_E = nv.lat_lon2n_E(rad(0), rad(0))
>>> n_EA2_E = nv.lat_lon2n_E(rad(10), rad(0))
>>> n_EB_E = nv.lat_lon2n_E(rad(1), rad(0.1))
>>> path = (n_EA1_E, n_EA2_E)
>>> radius = 6371e3  # mean earth radius [m]
>>> s_xt = nv.cross_track_distance(path, n_EB_E, radius=radius)
>>> d_xt = nv.cross_track_distance(path, n_EB_E, method='euclidean',
...                                radius=radius)
```

```
>>> val_txt = '{:4.2f} km, {:4.2f} km'.format(s_xt[0]/1000, d_xt[0]/1000)
>>> 'Ex10: Cross track distance: s_xt, d_xt = {0}'.format(val_txt)
'Ex10: Cross track distance: s_xt, d_xt = 11.12 km, 11.12 km'
```

```
>>> n_EC_E = nv.closest_point_on_great_circle(path, n_EB_E)
>>> np.allclose(nv.on_great_circle_path(path, n_EC_E, radius), True)
True
```

Alternative solution 2:

```
>>> s_xt2 = nv.great_circle_distance(n_EB_E, n_EC_E, radius)
>>> d_xt2 = nv.euclidean_distance(n_EB_E, n_EC_E, radius)
>>> np.allclose(s_xt, s_xt2), np.allclose(d_xt, d_xt2)
(True, True)
```

Alternative solution 3:

```
>>> c_E = nv.great_circle_normal(n_EA1_E, n_EA2_E)
>>> sin_theta = -np.dot(c_E.T, n_EB_E).ravel()
>>> s_xt3 = np.arcsin(sin_theta) * radius
>>> d_xt3 = sin_theta * radius
>>> np.allclose(s_xt, s_xt3), np.allclose(d_xt, d_xt3)
(True, True)
```

5.2.3 nvector.core.euclidean_distance

euclidean_distance(*n_EA_E*, *n_EB_E*, *radius=6371009.0*)

Returns Euclidean distance between positions A and B

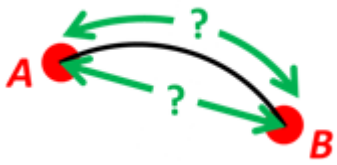
Parameters

n_EA_E, n_EB_E: 3 x n array n-vector(s) [no unit] of position A and B, decomposed in E.

radius: real scalar radius of sphere.

Examples

Example 5: "Surface distance"



Find the surface distance sAB (i.e. great circle distance) between two positions A and B. The heights of A and B are ignored, i.e. if they don't have zero height, we seek the distance between the points that are at the surface of the Earth, directly above/below A and B. The Euclidean distance (chord length) dAB should also be found. Use Earth radius 6371e3 m. Compare the results with exact calculations for the WGS-84 ellipsoid.

Solution for a sphere:

```
>>> import numpy as np
>>> import nvector as nv
>>> from nvector import rad
```

```
>>> n_EA_E = nv.lat_lon2n_E(rad(88), rad(0))
>>> n_EB_E = nv.lat_lon2n_E(rad(89), rad(-170))
```

```
>>> r_Earth = 6371e3  # m, mean Earth radius
>>> s_AB = nv.great_circle_distance(n_EA_E, n_EB_E, radius=r_Earth)[0]
>>> d_AB = nv.euclidean_distance(n_EA_E, n_EB_E, radius=r_Earth)[0]
```

```
>>> msg = 'Ex5: Great circle and Euclidean distance = {}'
>>> msg = msg.format('{:5.2f} km, {:5.2f} km')
>>> msg.format(s_AB / 1000, d_AB / 1000)
'Ex5: Great circle and Euclidean distance = 332.46 km, 332.42 km'
```

Exact solution for the WGS84 ellipsoid:

```
>>> wgs84 = nv.FrameE(name='WGS84')
>>> point1 = wgs84.GeoPoint(latitude=88, longitude=0, degrees=True)
>>> point2 = wgs84.GeoPoint(latitude=89, longitude=-170, degrees=True)
>>> s_12, _azi1, _azi2 = point1.distance_and_azimuth(point2)
```

```
>>> p_12_E = point2.to_ecef_vector() - point1.to_ecef_vector()
>>> d_12 = p_12_E.length
>>> msg = 'Ellipsoidal and Euclidean distance = {:5.2f} km, {:5.2f} km'
>>> msg.format(s_12 / 1000, d_12 / 1000)
'Ellipsoidal and Euclidean distance = 333.95 km, 333.91 km'
```

5.2.4 nvector.core.great_circle_distance

great_circle_distance(*n_EA_E*, *n_EB_E*, *radius=6371009.0*)

Returns great circle distance between positions A and B

Parameters

n_EA_E, n_EB_E: 3 x n array n-vector(s) [no unit] of position A and B, decomposed in E.

radius: real scalar radius of sphere.

Notes

The result for spherical Earth is returned. Formulae is given by equation (16) in Gade (2010) and is well conditioned for all angles.

Examples

Example 5: "Surface distance"

Find the surface distance sAB (i.e. great circle distance) between two positions A and B. The heights of A and B are ignored, i.e. if they don't have zero height, we seek the distance between the points that are at the surface of the Earth, directly above/below A and B. The Euclidean distance (chord length) dAB should also be found. Use Earth radius 6371e3 m. Compare the results with exact calculations for the WGS-84 ellipsoid.

Solution for a sphere:

```
>>> import numpy as np
>>> import nvector as nv
>>> from nvector import rad
```

```
>>> n_EA_E = nv.lat_lon2n_E(rad(88), rad(0))
>>> n_EB_E = nv.lat_lon2n_E(rad(89), rad(-170))
```

```
>>> r_Earth = 6371e3  # m, mean Earth radius
>>> s_AB = nv.great_circle_distance(n_EA_E, n_EB_E, radius=r_Earth)[0]
>>> d_AB = nv.euclidean_distance(n_EA_E, n_EB_E, radius=r_Earth)[0]
```

```
>>> msg = 'Ex5: Great circle and Euclidean distance = {}'
>>> msg = msg.format('{:5.2f} km, {:5.2f} km')
>>> msg.format(s_AB / 1000, d_AB / 1000)
'Ex5: Great circle and Euclidean distance = 332.46 km, 332.42 km'
```

Exact solution for the WGS84 ellipsoid:

```
>>> wgs84 = nv.FrameE(name='WGS84')
>>> point1 = wgs84.GeoPoint(latitude=88, longitude=0, degrees=True)
>>> point2 = wgs84.GeoPoint(latitude=89, longitude=-170, degrees=True)
>>> s_12, _azi1, _azi2 = point1.distance_and_azimuth(point2)
```

```
>>> p_12_E = point2.to_ecef_vector() - point1.to_ecef_vector()
>>> d_12 = p_12_E.length
>>> msg = 'Ellipsoidal and Euclidean distance = {:5.2f} km, {:5.2f} km'
>>> msg.format(s_12 / 1000, d_12 / 1000)
'Ellipsoidal and Euclidean distance = 333.95 km, 333.91 km'
```

5.2.5 nvector.core.great_circle_normal

great_circle_normal(*n_EA_E*, *n_EB_E*)
Returns the unit normal(s) to the great circle(s)

Parameters

n_EA_E, n_EB_E: 3 x n array n-vector(s) [no unit] of position A and B, decomposed in E.

5.2.6 nvector.core.interp_nvectors

interp_nvectors(*t_i*, *t*, *nvectors*, *kind='linear'*, *window_length=0*, *polyorder=2*, *mode='interp'*, *cval=0.0*)
Returns interpolated values from nvector data.

Parameters

t_i: real vector length m Vector of interpolation times.

t: real vector length n Vector of times.

nvectors: 3 x n array n-vector(s) [no unit] decomposed in E.

kind: str or int, optional Specifies the kind of interpolation as a string ('linear', 'nearest', 'zero', 'slinear', 'quadratic', 'cubic' where 'zero', 'slinear', 'quadratic' and 'cubic' refer to a spline interpolation of zeroth, first, second or third order) or as an integer specifying the order of the spline interpolator to use. Default is 'linear'.

window_length: positive odd integer The length of the Savitzky-Golay filter window (i.e., the number of coefficients). Default window_length=0, i.e. no smoothing.

polyorder: int The order of the polynomial used to fit the samples. polyorder must be less than window_length.

mode: 'mirror', 'constant', 'nearest', 'wrap' or 'interp'. Determines the type of extension to use for the padded signal to which the filter is applied. When mode is 'constant', the padding value is given by cval. When the 'interp' mode is selected (the default), no extension is used. Instead, a degree polyorder polynomial is fit to the last window_length values of the edges, and this polynomial is used to evaluate the last window_length // 2 output values.

cval: scalar, optional Value to fill past the edges of the input if mode is 'constant'. Default is 0.0.

Returns

result: 3 x m array Interpolated n-vector(s) [no unit] decomposed in E.

Notes

The result for spherical Earth is returned.

Examples

5.2.7 nvector.core.interpolate

`interpolate`(*path*, *ti*)

Returns the interpolated point along the path

Parameters

path: tuple of n-vectors (positionA, positionB)

ti: real scalar interpolation time assuming position A and B is at t0=0 and t1=1, respectively.

Returns

point: Nvector point of interpolation along path

Notes

The result for spherical Earth is returned.

5.2.8 nvector.core.intersect

`intersect`(*path_a*, *path_b*)

Returns the intersection(s) between the great circles of the two paths

Parameters

path_a, path_b: tuple of 2 n-vectors defining path A and path B, respectively. Path A and B has shape 2 x 3 x n and 2 x 3 x m, respectively.

Returns

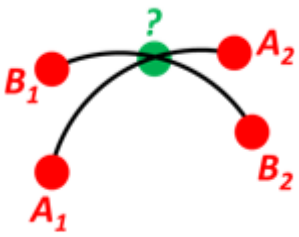
n_EC_E [array of shape 3 x max(n, m)] n-vector(s) [no unit] of position C decomposed in E. point(s) of intersection between paths.

Notes

The result for spherical Earth is returned.

Examples

Example 9: "Intersection of two paths"



Define a path from two given positions (at the surface of a spherical Earth), as the great circle that goes through the two points.

Path A is given by A1 and A2, while path B is given by B1 and B2.

Find the position C where the two great circles intersect.

Solution:

```
>>> import numpy as np
>>> import nvector as nv
>>> from nvector import rad, deg
```

```
>>> n_EA1_E = nv.lat_lon2n_E(rad(10), rad(20))
>>> n_EA2_E = nv.lat_lon2n_E(rad(30), rad(40))
>>> n_EB1_E = nv.lat_lon2n_E(rad(50), rad(60))
>>> n_EB2_E = nv.lat_lon2n_E(rad(70), rad(80))
```

```
>>> n_EC_E = nv.unit(np.cross(np.cross(n_EA1_E, n_EA2_E, axis=0),
...                           np.cross(n_EB1_E, n_EB2_E, axis=0),
...                           axis=0))
>>> n_EC_E *= np.sign(np.dot(n_EC_E.T, n_EA1_E))
```

or alternatively

```
>>> path_a, path_b = (n_EA1_E, n_EA2_E), (n_EB1_E, n_EB2_E)
>>> n_EC_E = nv.intersect(path_a, path_b)
```

```
>>> lat_EC, lon_EC = nv.n_E2lat_lon(n_EC_E)
```

```
>>> lat, lon = deg(lat_EC), deg(lon_EC)
>>> msg = 'Ex9, Intersection: lat, lon = {:4.4f}, {:4.4f} deg'
>>> msg.format(lat[0], lon[0])
'Ex9, Intersection: lat, lon = 40.3186, 55.9019 deg'
```

Check that PointC is not between A1 and A2 or B1 and B2:

```
>>> np.allclose([nv.on_great_circle_path(path_a, n_EC_E),
...              nv.on_great_circle_path(path_b, n_EC_E)], False)
True
```

Check that PointC is on the great circle going through path A and path B:

```
>>> np.allclose([nv.on_great_circle(path_a, n_EC_E), nv.on_great_circle(path_
↪b, n_EC_E)], True)
True
```

5.2.9 nvector.core.lat_lon2n_E

lat_lon2n_E(*latitude*, *longitude*, *R_Ee=None*)

Converts latitude and longitude to n-vector.

Parameters

latitude, longitude: real scalars or vectors of length n. Geodetic latitude and longitude given in [rad]

R_Ee [3 x 3 array] rotation matrix defining the axes of the coordinate frame E.

Returns

n_E: 3 x n array n-vector(s) [no unit] decomposed in E.

See also:

n_E2lat_lon

5.2.10 nvector.core.mean_horizontal_position

mean_horizontal_position(*n_EB_E*)

Returns the n-vector of the horizontal mean position.

Parameters

n_EB_E: 3 x n array n-vectors [no unit] of positions Bi, decomposed in E.

Returns

p_EM_E: 3 x 1 array n-vector [no unit] of the mean positions of all Bi, decomposed in E.

Notes

The result for spherical Earth is returned.

Examples

Example 7: "Mean position"

Three positions A, B, and C are given as n-vectors n_EA_E, n_EB_E, and n_EC_E. Find the mean position, M, given as n_EM_E. Note that the calculation is independent of the depths of the positions.

Solution:

```
>>> import numpy as np
>>> import nvector as nv
>>> from nvector import rad, deg
```

```
>>> n_EA_E = nv.lat_lon2n_E(rad(90), rad(0))
>>> n_EB_E = nv.lat_lon2n_E(rad(60), rad(10))
>>> n_EC_E = nv.lat_lon2n_E(rad(50), rad(-20))
```

```
>>> n_EM_E = nv.unit(n_EA_E + n_EB_E + n_EC_E)
```

or

```
>>> n_EM_E = nv.mean_horizontal_position(np.hstack((n_EA_E, n_EB_E, n_EC_E)))
```

```
>>> lat, lon = nv.n_E2lat_lon(n_EM_E)
>>> lat, lon = deg(lat), deg(lon)
>>> msg = 'Ex7: Pos M: lat, lon = {:4.4f}, {:4.4f} deg'
>>> msg.format(lat[0], lon[0])
'Ex7: Pos M: lat, lon = 67.2362, -6.9175 deg'
```

5.2.11 nvector.core.n_E2lat_lon

n_E2lat_lon(*n_E*, *R_Ee=None*)

Converts n-vector to latitude and longitude.

Parameters

n_E: 3 x n array n-vector [no unit] decomposed in E.

R_Ee [3 x 3 array] rotation matrix defining the axes of the coordinate frame E.

Returns

latitude, longitude: real scalars or vectors of length n. Geodetic latitude and longitude given in [rad]

See also:

lat_lon2n_E

5.2.12 nvector.core.n_EB_E2p_EB_E

n_EB_E2p_EB_E(*n_EB_E*, *depth=0*, *a=6378137*, *f=0.0033528106647474805*, *R_Ee=None*)

Converts n-vector to Cartesian position vector in meters.

Parameters

n_EB_E: 3 x n array n-vector(s) [no unit] of position B, decomposed in E.

depth: 1 x n array Depth(s) [m] of system B, relative to the ellipsoid (depth = -height)

a: real scalar, default WGS-84 ellipsoid. Semi-major axis of the Earth ellipsoid given in [m].

f: real scalar, default WGS-84 ellipsoid. Flattening [no unit] of the Earth ellipsoid. If f==0 then spherical Earth with radius a is used in stead of WGS-84.

R_Ee [3 x 3 array] rotation matrix defining the axes of the coordinate frame E.

Returns

p_EB_E: 3 x n array Cartesian position vector(s) from E to B, decomposed in E.

See also:

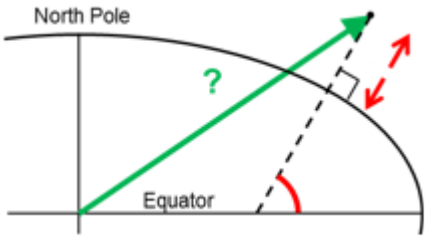
p_EB_E2n_EB_E, *n_EA_E_and_p_AB_E2n_EB_E*, *n_EA_E_and_n_EB_E2p_AB_E*

Notes

The position of B (typically body) relative to E (typically Earth) is given into this function as n-vector, n_EB_E. The function converts to cartesian position vector ("ECEF-vector"), p_EB_E, in meters. The calculation is exact, taking the ellipsity of the Earth into account. It is also non-singular as both n-vector and p-vector are non-singular (except for the center of the Earth). The default ellipsoid model used is WGS-84, but other ellipsoids/spheres might be specified.

Examples

Example 4: "Geodetic latitude to ECEF-vector"



Geodetic latitude, longitude and height are given for position B as latEB, lonEB and hEB, find the ECEF-vector for this position, p_EB_E.

Solution:

```
>>> import nvector as nv
>>> from nvector import rad
>>> wgs84 = dict(a=6378137.0, f=1.0/298.257223563)
>>> lat_EB, lon_EB = rad(1), rad(2)
>>> h_EB = 3
>>> n_EB_E = nv.lat_lon2n_E(lat_EB, lon_EB)
>>> p_EB_E = nv.n_EB_E2p_EB_E(n_EB_E, -h_EB, **wgs84)
```

```
>>> 'Ex4: p_EB_E = {} m'.format(p_EB_E.ravel().tolist())
'Ex4: p_EB_E = [6373290.277218279, 222560.20067473652, 110568.82718178593] m'
```

5.2.13 nvector.core.p_EB_E2n_EB_E

p_EB_E2n_EB_E(*p_EB_E*, *a=6378137*, *f=0.0033528106647474805*, *R_Ee=None*)

Converts Cartesian position vector in meters to n-vector.

Parameters

p_EB_E: 3 x n array Cartesian position vector(s) from E to B, decomposed in E.

a: real scalar, default WGS-84 ellipsoid. Semi-major axis of the Earth ellipsoid given in [m].

f: real scalar, default WGS-84 ellipsoid. Flattening [no unit] of the Earth ellipsoid. If f==0 then spherical Earth with radius a is used in stead of WGS-84.

R_Ee [3 x 3 array] rotation matrix defining the axes of the coordinate frame E.

Returns

n_EB_E: 3 x n array n-vector(s) [no unit] of position B, decomposed in E.

depth: 1 x n array Depth(s) [m] of system B, relative to the ellipsoid (depth = -height)

See also:

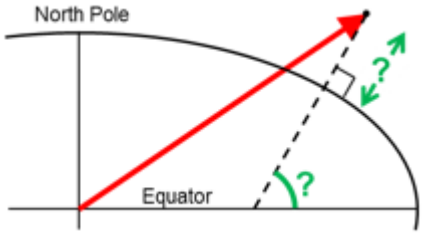
n_EB_E2p_EB_E, *n_EA_E_and_p_AB_E2n_EB_E*, *n_EA_E_and_n_EB_E2p_AB_E*

Notes

The position of B (typically body) relative to E (typically Earth) is given into this function as cartesian position vector p_EB_E, in meters. ("ECEF-vector"). The function converts to n-vector, n_EB_E and its depth, depth. The calculation is excact, taking the ellipsity of the Earth into account. It is also non-singular as both n-vector and p-vector are non-singular (except for the center of the Earth). The default ellipsoid model used is WGS-84, but other ellipsoids/spheres might be specified.

Examples

Example 3: "ECEF-vector to geodetic latitude"



Position B is given as an "ECEF-vector" p_EB_E (i.e. a vector from E, the center of the Earth, to B, decomposed in E). Find the geodetic latitude, longitude and height (latEB, lonEB and hEB), assuming WGS-84 ellipsoid.

Solution:

```
>>> import numpy as np
>>> import nvector as nv
>>> from nvector import deg
>>> wgs84 = dict(a=6378137.0, f=1.0/298.257223563)
>>> p_EB_E = 6371e3 * np.vstack((0.9, -1, 1.1))  # m
```

```
>>> n_EB_E, z_EB = nv.p_EB_E2n_EB_E(p_EB_E, **wgs84)
```

```
>>> lat_EB, lon_EB = nv.n_E2lat_lon(n_EB_E)
>>> h = -z_EB
>>> lat, lon = deg(lat_EB), deg(lon_EB)
```

```
>>> msg = 'Ex3: Pos B: lat, lon = {:4.4f}, {:4.4f} deg, height = {:9.3f} m'
>>> msg.format(lat[0], lon[0], h[0])
'Ex3: Pos B: lat, lon = 39.3787, -48.0128 deg, height = 4702059.834 m'
```

5.2.14 nvector.core.n_EA_E_and_n_EB_E2p_AB_E

n_EA_E_and_n_EB_E2p_AB_E(*n_EA_E*, *n_EB_E*, *z_EA=0*, *z_EB=0*, *a=6378137*, *f=0.0033528106647474805*, *R_Ee=None*)

Returns the delta vector from position A to B decomposed in E.

Parameters

n_EA_E, n_EB_E: 3 x n array n-vector(s) [no unit] of position A and B, decomposed in E.

z_EA, z_EB: 1 x n array Depth(s) [m] of system A and B, relative to the ellipsoid. (z_EA = -height, z_EB = -height)

a: real scalar, default WGS-84 ellipsoid. Semi-major axis of the Earth ellipsoid given in [m].

f: real scalar, default WGS-84 ellipsoid. Flattening [no unit] of the Earth ellipsoid. If f==0 then spherical Earth with radius a is used in stead of WGS-84.

R_Ee [3 x 3 array] rotation matrix defining the axes of the coordinate frame E.

Returns

p_AB_E: 3 x n array Cartesian position vector(s) from A to B, decomposed in E.

See also:

n_EA_E_and_p_AB_E2n_EB_E, *p_EB_E2n_EB_E*, *n_EB_E2p_EB_E*

Notes

The n-vectors for positions A (n_EA_E) and B (n_EB_E) are given. The output is the delta vector from A to B (p_AB_E). The calculation is excact, taking the ellipsity of the Earth into account. It is also non-singular as both n-vector and p-vector are non-singular (except for the center of the Earth). The default ellipsoid model used is WGS-84, but other ellipsoids/spheres might be specified.

Examples

Example 1: "A and B to delta"

Given two positions, A and B as latitudes, longitudes and depths relative to Earth, E.

Find the exact vector between the two positions, given in meters north, east, and down, and find the direction (azimuth) to B, relative to north. Assume WGS-84 ellipsoid. The given depths are from the ellipsoid surface. Use position A to define north, east, and down directions. (Due to the curvature of Earth and different directions to the North Pole, the north, east, and down directions will change (relative to Earth) for different places. Position A must be outside the poles for the north and east directions to be defined.)

Solution:

```
>>> import numpy as np
>>> import nvector as nv
>>> from nvector import rad, deg
```

```
>>> lat_EA, lon_EA, z_EA = rad(1), rad(2), 3
>>> lat_EB, lon_EB, z_EB = rad(4), rad(5), 6
```

Step1: Convert to n-vectors:

```
>>> n_EA_E = nv.lat_lon2n_E(lat_EA, lon_EA)
>>> n_EB_E = nv.lat_lon2n_E(lat_EB, lon_EB)
```

Step2: Find p_AB_E (delta decomposed in E).WGS-84 ellipsoid is default:

```
>>> p_AB_E = nv.n_EA_E_and_n_EB_E2p_AB_E(n_EA_E, n_EB_E, z_EA, z_EB)
```

Step3: Find R_EN for position A:

```
>>> R_EN = nv.n_E2R_EN(n_EA_E)
```

Step4: Find p_AB_N (delta decomposed in N).

```
>>> p_AB_N = np.dot(R_EN.T, p_AB_E).ravel()
>>> x, y, z = p_AB_N
>>> 'Ex1: delta north, east, down = {0:8.2f}, {1:8.2f}, {2:8.2f}'.format(x,␣
↪y, z)
'Ex1: delta north, east, down = 331730.23, 332997.87, 17404.27'
```

Step5: Also find the direction (azimuth) to B, relative to north:

```
>>> azimuth = np.arctan2(y, x)
>>> 'azimuth = {0:4.2f} deg'.format(deg(azimuth))
'azimuth = 45.11 deg'
```

```
>>> distance = np.linalg.norm(p_AB_N)
>>> elevation = np.arcsin(z / distance)
>>> 'elevation = {0:4.2f} deg'.format(deg(elevation))
'elevation = 2.12 deg'
```

```
>>> 'distance = {0:4.2f} m'.format(distance)
'distance = 470356.72 m'
```

5.2.15 nvector.core.n_EA_E_and_p_AB_E2n_EB_E

n_EA_E_and_p_AB_E2n_EB_E(*n_EA_E*, *p_AB_E*, *z_EA=0*, *a=6378137*, *f=0.003352810664747480 5*, *R_Ee=None*)

Returns position B from position A and delta E vector.

Parameters

n_EA_E: 3 x n array n-vector(s) [no unit] of position A, decomposed in E.

p_AB_E: 3 x n array Cartesian position vector(s) from A to B, decomposed in E.

z_EA: 1 x n array Depth(s) [m] of system A, relative to the ellipsoid. (z_EA = -height)

a: real scalar, default WGS-84 ellipsoid. Semi-major axis of the Earth ellipsoid given in [m].

f: real scalar, default WGS-84 ellipsoid. Flattening [no unit] of the Earth ellipsoid. If f==0 then spherical Earth with radius a is used in stead of WGS-84.

R_Ee [3 x 3 array] rotation matrix defining the axes of the coordinate frame E.

Returns

n_EB_E: 3 x n array n-vector(s) [no unit] of position B, decomposed in E.

z_EB: 1 x n array Depth(s) [m] of system B, relative to the ellipsoid. (z_EB = -height)

See also:

n_EA_E_and_n_EB_E2p_AB_E, *p_EB_E2n_EB_E*, *n_EB_E2p_EB_E*

Notes

The n-vector for position A (n_EA_E) and the position-vector from position A to position B (p_AB_E) are given. The output is the n-vector of position B (n_EB_E) and depth of B (z_EB). The calculation is excact, taking the ellipsity of the Earth into account. It is also non-singular as both n-vector and p-vector are non-singular (except for the center of the Earth). The default ellipsoid model used is WGS-84, but other ellipsoids/spheres might be specified.

Example 2: "B and delta to C"

A radar or sonar attached to a vehicle B (Body coordinate frame) measures the distance and direction to an object C. We assume that the distance and two angles (typically bearing and elevation relative to B) are already combined to the vector p_BC_B (i.e. the vector from B to C, decomposed in B). The position of B is given as n_EB_E and z_EB, and the orientation (attitude) of B is given as R_NB (this rotation matrix can be found from roll/pitch/yaw by using zyx2R).

Find the exact position of object C as n-vector and depth (n_EC_E and z_EC), assuming Earth ellipsoid with semi-major axis a and flattening f. For WGS-72, use a = 6 378 135 m and f = 1/298.26.

Solution:

```
>>> import numpy as np
>>> import nvector as nv
>>> from nvector import rad, deg
```

A custom reference ellipsoid is given (replacing WGS-84):

```
>>> wgs72 = dict(a=6378135, f=1.0/298.26)
```

Step 1 Position and orientation of B is 400m above E:

```
>>> n_EB_E = nv.unit([[1], [2], [3]])  # unit to get unit length of vector
>>> z_EB = -400
>>> yaw, pitch, roll = rad(10), rad(20), rad(30)
>>> R_NB = nv.zyx2R(yaw, pitch, roll)
```

Step 2: Delta BC decomposed in B

```
>>> p_BC_B = np.r_[3000, 2000, 100].reshape((-1, 1))
```

Step 3: Find R_EN:

```
>>> R_EN = nv.n_E2R_EN(n_EB_E)
```

Step 4: Find R_EB, from R_EN and R_NB:

```
>>> R_EB = np.dot(R_EN, R_NB)  # Note: closest frames cancel
```

Step 5: Decompose the delta BC vector in E:

```
>>> p_BC_E = np.dot(R_EB, p_BC_B)
```

Step 6: Find the position of C, using the functions that goes from one

```
>>> n_EC_E, z_EC = nv.n_EA_E_and_p_AB_E2n_EB_E(n_EB_E, p_BC_E, z_EB, **wgs72)
```

```
>>> lat_EC, lon_EC = nv.n_E2lat_lon(n_EC_E)
>>> lat, lon, z = deg(lat_EC), deg(lon_EC), z_EC
>>> msg = 'Ex2: PosC: lat, lon = {:4.4f}, {:4.4f} deg,  height = {:4.2f} m'
>>> msg.format(lat[0], lon[0], -z[0])
'Ex2: PosC: lat, lon = 53.3264, 63.4681 deg,  height = 406.01 m'
```

5.2.16 nvector.core.n_EA_E_and_n_EB_E2azimuth

n_EA_E_and_n_EB_E2azimuth(*n_EA_E*, *n_EB_E*, *a=6378137*, *f=0.0033528106647474805*, *R_Ee=None*)
Returns azimuth from A to B, relative to North:

Parameters

n_EA_E, n_EB_E: 3 x n array n-vector(s) [no unit] of position A and B, respectively, decomposed in E.

a: real scalar, default WGS-84 ellipsoid. Semi-major axis of the Earth ellipsoid given in [m].

f: real scalar, default WGS-84 ellipsoid. Flattening [no unit] of the Earth ellipsoid. If f==0 then spherical Earth with radius a is used in stead of WGS-84.

R_Ee [3 x 3 array] rotation matrix defining the axes of the coordinate frame E.

Returns

azimuth: n array Angle [rad] the line makes with a meridian, taken clockwise from north.

5.2.17 nvector.core.n_EA_E_distance_and_azimuth2n_EB_E

n_EA_E_distance_and_azimuth2n_EB_E(*n_EA_E*, *distance_rad*, *azimuth*, *R_Ee=None*)
Returns position B from azimuth and distance from position A

Parameters

n_EA_E: 3 x n array n-vector(s) [no unit] of position A decomposed in E.

distance_rad: n, array great circle distance [rad] from position A to B

azimuth: n array Angle [rad] the line makes with a meridian, taken clockwise from north.

Returns

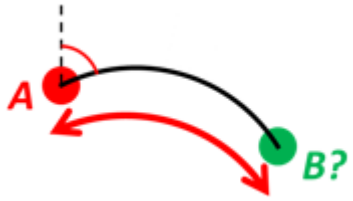
n_EB_E: 3 x n array n-vector(s) [no unit] of position B decomposed in E.

Notes

The result for spherical Earth is returned.

Examples

Example 8: "A and azimuth/distance to B"



We have an initial position A, direction of travel given as an azimuth (bearing) relative to north (clockwise), and finally the distance to travel along a great circle given as sAB. Use Earth radius 6371e3 m to find the destination point B.

In geodesy this is known as "The first geodetic problem" or "The direct geodetic problem" for a sphere, and we see that this is similar to Example 2[35], but now the delta is given as an azimuth and a great circle distance. ("The second/inverse geodetic problem" for a sphere is already solved in Examples 1[36] and 5[37].)

Solution:

```
>>> import nvector as nv
>>> from nvector import rad, deg
>>> lat, lon = rad(80), rad(-90)
```

```
>>> n_EA_E = nv.lat_lon2n_E(lat, lon)
>>> azimuth = rad(200)
>>> s_AB = 1000.0  # [m]
>>> r_earth = 6371e3  # [m], mean earth radius
```

```
>>> distance_rad = s_AB / r_earth
>>> n_EB_E = nv.n_EA_E_distance_and_azimuth2n_EB_E(n_EA_E, distance_rad,␣
↪azimuth)
>>> lat_EB, lon_EB = nv.n_E2lat_lon(n_EB_E)
>>> lat, lon = deg(lat_EB), deg(lon_EB)
>>> msg = 'Ex8, Destination: lat, lon = {:4.4f} deg, {:4.4f} deg'
>>> msg.format(lat[0], lon[0])
'Ex8, Destination: lat, lon = 79.9915 deg, -90.0177 deg'
```

5.2.18 nvector.core.on_great_circle

on_great_circle(*path*, *n_EB_E*, *radius=6371009.0*, *atol=1e-08*)

Returns True if position B is on great circle through path A.

Parameters

path: tuple of 2 n-vectors 2 n-vectors of positions defining path A, decomposed in E.

n_EB_E: 3 x m array n-vector(s) of position B to check to.

radius: real scalar radius of sphere. (default 6371009.0)

atol: real scalar The absolute tolerance parameter (See notes).

Returns

on [bool array of length max(n, m)] True if position B is on great circle through path A.

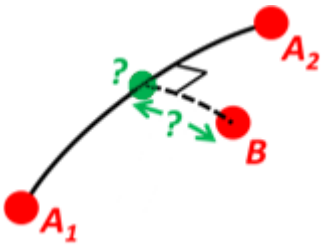
[35] http://www.navlab.net/nvector/#example_2
[36] http://www.navlab.net/nvector/#example_1
[37] http://www.navlab.net/nvector/#example_5

Notes

The default value of *atol* is not zero, and is used to determine what small values should be considered close to zero. The default value is appropriate for expected values of order unity. However, *atol* should be carefully selected for the use case at hand. Typically the value should be set to the accepted error tolerance. For GPS data the error ranges from 0.01 m to 15 m.

Examples

Example 10: "Cross track distance"



Path A is given by the two positions A1 and A2 (similar to the previous example).

Find the cross track distance sxt between the path A (i.e. the great circle through A1 and A2) and the position B (i.e. the shortest distance at the surface, between the great circle and B).

Also find the Euclidean distance dxt between B and the plane defined by the great circle. Use Earth radius 6371e3.

Finally, find the intersection point on the great circle and determine if it is between position A1 and A2.

Solution:

```
>>> import numpy as np
>>> import nvector as nv
>>> from nvector import rad, deg
>>> n_EA1_E = nv.lat_lon2n_E(rad(0), rad(0))
>>> n_EA2_E = nv.lat_lon2n_E(rad(10), rad(0))
>>> n_EB_E = nv.lat_lon2n_E(rad(1), rad(0.1))
>>> path = (n_EA1_E, n_EA2_E)
>>> radius = 6371e3  # mean earth radius [m]
>>> s_xt = nv.cross_track_distance(path, n_EB_E, radius=radius)
>>> d_xt = nv.cross_track_distance(path, n_EB_E, method='euclidean',
...                                radius=radius)
```

```
>>> val_txt = '{:4.2f} km, {:4.2f} km'.format(s_xt[0]/1000, d_xt[0]/1000)
>>> 'Ex10: Cross track distance: s_xt, d_xt = {0}'.format(val_txt)
'Ex10: Cross track distance: s_xt, d_xt = 11.12 km, 11.12 km'
```

```
>>> n_EC_E = nv.closest_point_on_great_circle(path, n_EB_E)
>>> np.allclose(nv.on_great_circle_path(path, n_EC_E, radius), True)
True
```

Alternative solution 2:

```
>>> s_xt2 = nv.great_circle_distance(n_EB_E, n_EC_E, radius)
>>> d_xt2 = nv.euclidean_distance(n_EB_E, n_EC_E, radius)
>>> np.allclose(s_xt, s_xt2), np.allclose(d_xt, d_xt2)
(True, True)
```

Alternative solution 3:

```
>>> c_E = nv.great_circle_normal(n_EA1_E, n_EA2_E)
>>> sin_theta = -np.dot(c_E.T, n_EB_E).ravel()
>>> s_xt3 = np.arcsin(sin_theta) * radius
>>> d_xt3 = sin_theta * radius
>>> np.allclose(s_xt, s_xt3), np.allclose(d_xt, d_xt3)
(True, True)
```

5.2.19 nvector.core.on_great_circle_path

on_great_circle_path(*path*, *n_EB_E*, *radius=6371009.0*, *atol=1e-08*)

Returns True if position B is on great circle and between endpoints of path A.

Parameters

path: tuple of 2 n-vectors 2 n-vectors of positions defining path A, decomposed in E.

n_EB_E: 3 x m array n-vector(s) of position B to measure the cross track distance to.

radius: real scalar radius of sphere. (default 6371009.0)

atol: real scalars The absolute tolerance parameter (See notes).

Returns

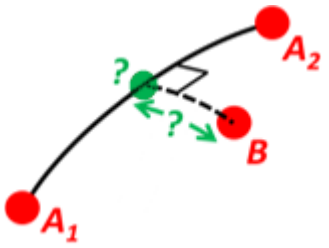
on [bool array of length max(n, m)] True if position B is on great circle and between endpoints of path A.

Notes

The default value of *atol* is not zero, and is used to determine what small values should be considered close to zero. The default value is appropriate for expected values of order unity. However, *atol* should be carefully selected for the use case at hand. Typically the value should be set to the accepted error tolerance. For GPS data the error ranges from 0.01 m to 15 m.

Examples

Example 10: "Cross track distance"



Path A is given by the two positions A1 and A2 (similar to the previous example).

Find the cross track distance sxt between the path A (i.e. the great circle through A1 and A2) and the position B (i.e. the shortest distance at the surface, between the great circle and B).

Also find the Euclidean distance dxt between B and the plane defined by the great circle. Use Earth radius 6371e3.

Finally, find the intersection point on the great circle and determine if it is between position A1 and A2.

Solution:

```
>>> import numpy as np
>>> import nvector as nv
>>> from nvector import rad, deg
>>> n_EA1_E = nv.lat_lon2n_E(rad(0), rad(0))
>>> n_EA2_E = nv.lat_lon2n_E(rad(10), rad(0))
>>> n_EB_E = nv.lat_lon2n_E(rad(1), rad(0.1))
>>> path = (n_EA1_E, n_EA2_E)
>>> radius = 6371e3  # mean earth radius [m]
>>> s_xt = nv.cross_track_distance(path, n_EB_E, radius=radius)
>>> d_xt = nv.cross_track_distance(path, n_EB_E, method='euclidean',
...                                radius=radius)
```

```
>>> val_txt = '{:4.2f} km, {:4.2f} km'.format(s_xt[0]/1000, d_xt[0]/1000)
>>> 'Ex10: Cross track distance: s_xt, d_xt = {0}'.format(val_txt)
'Ex10: Cross track distance: s_xt, d_xt = 11.12 km, 11.12 km'
```

```
>>> n_EC_E = nv.closest_point_on_great_circle(path, n_EB_E)
>>> np.allclose(nv.on_great_circle_path(path, n_EC_E, radius), True)
True
```

Alternative solution 2:

```
>>> s_xt2 = nv.great_circle_distance(n_EB_E, n_EC_E, radius)
>>> d_xt2 = nv.euclidean_distance(n_EB_E, n_EC_E, radius)
>>> np.allclose(s_xt, s_xt2), np.allclose(d_xt, d_xt2)
(True, True)
```

Alternative solution 3:

```
>>> c_E = nv.great_circle_normal(n_EA1_E, n_EA2_E)
>>> sin_theta = -np.dot(c_E.T, n_EB_E).ravel()
>>> s_xt3 = np.arcsin(sin_theta) * radius
>>> d_xt3 = sin_theta * radius
>>> np.allclose(s_xt, s_xt3), np.allclose(d_xt, d_xt3)
(True, True)
```

5.3 Rotation matrices and angles

E_rotation([axes])	Returns rotation matrix R_Ee defining the axes of the coordinate frame E.
n_E2R_EN(n_E[, R_Ee])	Returns the rotation matrix R_EN from n-vector.
n_E_and_wa2R_EL(n_E, wander_azimuth[, R_Ee])	Returns rotation matrix R_EL from n-vector and wander azimuth angle.
R_EL2n_E(R_EL)	Returns n-vector from the rotation matrix R_EL.
R_EN2n_E(R_EN)	Returns n-vector from the rotation matrix R_EN.
R2xyz(R_AB)	Returns the angles about new axes in the xyz-order from a rotation matrix.
R2zyx(R_AB)	Returns the angles about new axes in the zxy-order from a rotation matrix.
xyz2R(x, y, z)	Returns rotation matrix from 3 angles about new axes in the xyz-order.
zyx2R(z, y, x)	Returns rotation matrix from 3 angles about new axes in the zyx-order.

5.3.1 nvector.rotation.E_rotation

E_rotation(*axes='e'*)

Returns rotation matrix R_Ee defining the axes of the coordinate frame E.

Parameters

axes [‘e’ or ‘E’] defines orientation of the axes of the coordinate frame E. If axes is ‘e’ then z-axis points to the North Pole along the Earth’s rotation axis, x-axis points towards the point where latitude = longitude = 0. If axes is ‘E’ then x-axis points to the North Pole along the Earth’s rotation axis, y-axis points towards longitude +90deg (east) and latitude = 0.

Returns

R_Ee [3 x 3 array] rotation matrix defining the axes of the coordinate frame E as described in Table 2 in Gade (2010).

Notes

R_Ee controls the axes of the coordinate frame E (Earth-Centred, Earth-Fixed, ECEF) used by the other functions in this library. It is very common in many fields to choose axes equal to ‘e’, which is also the default in this library. Previously the old matlab toolbox the default value was equal to ‘E’. If you choose axes equal to ‘E’ the yz-plane coincides with the equatorial plane. This choice of axis ensures that at zero latitude and longitude, frame N (North-East-Down) has the same orientation as frame E. If roll/pitch/yaw are zero, also frame B (forward-starboard-down) has this orientation. In this manner, the axes of frame E is chosen to correspond with the axes of frame N and B.

References

Gade, K. (2010). A Nonsingular Horizontal Position Representation, The Journal of Navigation, Volume 63, Issue 03, pp 395-417, July 2010.

Examples

```
>>> import numpy as np
>>> import nvector as nv
>>> np.allclose(nv.E_rotation(axes='e'), [[ 0,  0,  1],
...                                       [ 0,  1,  0],
...                                       [-1,  0,  0]])
True
>>> np.allclose(nv.E_rotation(axes='E'), [[ 1.,  0.,  0.],
...                                       [ 0.,  1.,  0.],
...                                       [ 0.,  0.,  1.]])
True
```

5.3.2 nvector.rotation.n_E2R_EN

n_E2R_EN(*n_E*, *R_Ee=None*)

Returns the rotation matrix R_EN from n-vector.

Parameters

n_E: 3 x n array n-vector [no unit] decomposed in E

R_Ee [3 x 3 array] rotation matrix defining the axes of the coordinate frame E.

Returns

R_EN: 3 x 3 x n array The resulting rotation matrix [no unit] (direction cosine matrix).

See also:

R_EN2n_E, *n_E_and_wa2R_EL*, *R_EL2n_E*

5.3.3 nvector.rotation.n_E_and_wa2R_EL

n_E_and_wa2R_EL(*n_E*, *wander_azimuth*, *R_Ee=None*)

Returns rotation matrix R_EL from n-vector and wander azimuth angle.

Parameters

n_E: 3 x n array n-vector [no unit] decomposed in E

wander_azimuth: real scalar or array of length n Angle [rad] between L's x-axis and north, positive about L's z-axis.

R_Ee [3 x 3 array] rotation matrix defining the axes of the coordinate frame E.

Returns

R_EL: 3 x 3 x n array The resulting rotation matrix. [no unit]

See also:

R_EL2n_E, *R_EN2n_E*, *n_E2R_EN*

Notes

When wander_azimuth=0, we have that N=L. (See Table 2 in Gade (2010) for details)

5.3.4 nvector.rotation.R_EL2n_E

R_EL2n_E(*R_EL*)

Returns n-vector from the rotation matrix R_EL.

Parameters

R_EL: 3 x 3 x n array Rotation matrix (direction cosine matrix) [no unit]

Returns

n_E: 3 x n array n-vector(s) [no unit] decomposed in E.

See also:

R_EN2n_E, *n_E_and_wa2R_EL*, *n_E2R_EN*

5.3.5 nvector.rotation.R_EN2n_E

R_EN2n_E(*R_EN*)

Returns n-vector from the rotation matrix R_EN.

Parameters

R_EN: 3 x 3 x n array Rotation matrix (direction cosine matrix) [no unit]

Returns

n_E: 3 x n array n-vector [no unit] decomposed in E.

See also:

n_E2R_EN, *R_EL2n_E*, *n_E_and_wa2R_EL*

5.3.6 nvector.rotation.R2xyz

R2xyz(*R_AB*)

Returns the angles about new axes in the xyz-order from a rotation matrix.

Parameters

R_AB: 3 x 3 x n array rotation matrix [no unit] (direction cosine matrix) such that the relation between a vector v decomposed in A and B is given by: v_A = mdot(R_AB, v_B)

Returns

x, y, z: real scalars or array of length n. Angles [rad] of rotation about new axes.

See also:

xyz2R, *R2zyx*, *xyz2R*

Notes

The x, y, z angles are called Euler angles or Tait-Bryan angles and are defined by the following procedure of successive rotations: Given two arbitrary coordinate frames A and B. Consider a temporary frame T that initially coincides with A. In order to make T align with B, we first rotate T an angle x about its x-axis (common axis for both A and T). Secondly, T is rotated an angle y about the NEW y-axis of T. Finally, T is rotated an angle z about its NEWEST z-axis. The final orientation of T now coincides with the orientation of B.

The signs of the angles are given by the directions of the axes and the right hand rule.

See also: https://en.wikipedia.org/wiki/Aircraft_principal_axes https://en.wikipedia.org/wiki/Euler_angles https://en.wikipedia.org/wiki/Axes_conventions

5.3.7 nvector.rotation.R2zyx

R2zyx(*R_AB*)

Returns the angles about new axes in the zxy-order from a rotation matrix.

Parameters

R_AB: 3x3 array rotation matrix [no unit] (direction cosine matrix) such that the relation between a vector v decomposed in A and B is given by: v_A = np.dot(R_AB, v_B)

Returns

z, y, x: real scalars Angles [rad] of rotation about new axes.

See also:

zyx2R, *xyz2R*, *R2xyz*

Notes

The z, x, y angles are called Euler angles or Tait-Bryan angles and are defined by the following procedure of successive rotations: Given two arbitrary coordinate frames A and B. Consider a temporary frame T that initially coincides with A. In order to make T align with B, we first rotate T an angle z about its z-axis (common axis for both A and T). Secondly, T is rotated an angle y about the NEW y-axis of T. Finally, T is rotated an angle x about its NEWEST x-axis. The final orientation of T now coincides with the orientation of B.

The signs of the angles are given by the directions of the axes and the right hand rule.

Note that if A is a north-east-down frame and B is a body frame, we have that z=yaw, y=pitch and x=roll.

See also: https://en.wikipedia.org/wiki/Aircraft_principal_axes https://en.wikipedia.org/wiki/Euler_angles https://en.wikipedia.org/wiki/Axes_conventions

5.3.8 nvector.rotation.xyz2R

xyz2R(*x*, *y*, *z*)

Returns rotation matrix from 3 angles about new axes in the xyz-order.

Parameters

x,y,z: real scalars or array of lengths n Angles [rad] of rotation about new axes.

Returns

R_AB: 3 x 3 x n array rotation matrix [no unit] (direction cosine matrix) such that the relation between a vector v decomposed in A and B is given by: v_A = mdot(R_AB, v_B)

See also:

R2xyz, *zyx2R*, *R2zyx*

Notes

The rotation matrix R_AB is created based on 3 angles x,y,z about new axes (intrinsic) in the order x-y-z. The angles are called Euler angles or Tait-Bryan angles and are defined by the following procedure of successive rotations: Given two arbitrary coordinate frames A and B. Consider a temporary frame T that initially coincides with A. In order to make T align with B, we first rotate T an angle x about its x-axis (common axis for both A and T). Secondly, T is rotated an angle y about the NEW y-axis of T. Finally, T is rotated an angle z about its NEWEST z-axis. The final orientation of T now coincides with the orientation of B.

The signs of the angles are given by the directions of the axes and the right hand rule.

See also: https://en.wikipedia.org/wiki/Aircraft_principal_axes https://en.wikipedia.org/wiki/Euler_angles https://en.wikipedia.org/wiki/Axes_conventions

5.3.9 nvector.rotation.zyx2R

zyx2R(*z*, *y*, *x*)

Returns rotation matrix from 3 angles about new axes in the zyx-order.

Parameters

z, y, x: real scalars or arrays of lenths n Angles [rad] of rotation about new axes.

Returns

R_AB: 3 x 3 x n array rotation matrix [no unit] (direction cosine matrix) such that the relation between a vector v decomposed in A and B is given by: v_A = mdot(R_AB, v_B)

See also:

R2zyx, *xyz2R*, *R2xyz*

Notes

The rotation matrix R_AB is created based on 3 angles z,y,x about new axes (intrinsic) in the order z-y-x. The angles are called Euler angles or Tait-Bryan angles and are defined by the following procedure of successive rotations: Given two arbitrary coordinate frames A and B. Consider a temporary frame T that initially coincides with A. In order to make T align with B, we first rotate T an angle z about its z-axis (common axis for both A and T). Secondly, T is rotated an angle y about the NEW y-axis of T. Finally, T is rotated an angle x about its NEWEST x-axis. The final orientation of T now coincides with the orientation of B.

The signs of the angles are given by the directions of the axes and the right hand rule.

Note that if A is a north-east-down frame and B is a body frame, we have that z=yaw, y=pitch and x=roll.

See also: https://en.wikipedia.org/wiki/Aircraft_principal_axes https://en.wikipedia.org/wiki/Euler_angles https://en.wikipedia.org/wiki/Axes_conventions

Examples

Suppose the yaw angle between coordinate system A and B is 45 degrees. Convert position p1_b = (1, 0, 0) in B to a point in A. Convert position p2_a =(0, 1, 0) in A to a point in B.

Solution:

```
>>> import numpy as np
>>> import nvector as nv
>>> x, y, z = nv.rad(0, 0, 45)
>>> R_AB = nv.zyx2R(z, y, x)
```

```
>>> p1_b = np.atleast_2d((1, 0, 0)).T
>>> p1_a = nv.mdot(R_AB, p1_b)
>>> np.allclose(p1_a, [[0.7071067811865476], [0.7071067811865476], [0.0]])
True
```

```
>>> p2_a = np.atleast_2d((0, 1, 0)).T
>>> p2_b = nv.mdot(R_AB.T, p2_a)
>>> np.allclose(p2_b, [[0.7071067811865476], [0.7071067811865476], [0.0]])
True
```

5.4 Utility functions

deg(*rad_angles)	Converts angle in radians to degrees.
mdot(a, b)	Returns multiple matrix multiplications of two arrays i.e. dot(a, b)[i,j,k] = sum(a[i,:,j] * b[:,j,k]).
nthroot(x, n)	Returns the n'th root of x to machine precision
rad(*deg_angles)	Converts angle in degrees to radians.
get_ellipsoid(name)	Returns semi-major axis (a), flattening (f) and name of ellipsoid as a named tuple.
select_ellipsoid(*args, **kwds)	*select_ellipsoid* is deprecated, use *get_ellipsoid* instead!
unit(vector[, norm_zero_vector])	Convert input vector to a vector of unit length.

5.4.1 nvector.util.deg

deg(**rad_angles*)

Converts angle in radians to degrees.

Parameters

rad_angles: angle in radians

Returns

deg_angles: angle in degrees

See also:

rad

Examples

```
>>> import numpy as np
>>> import nvector as nv
>>> nv.deg(np.pi/2)
90.0
>>> nv.deg(np.pi/2, [0, np.pi])
(90.0, array([  0., 180.]))
```

5.4.2 nvector.util.mdot

mdot(*a*, *b*)

Returns multiple matrix multiplications of two arrays i.e. dot(a, b)[i,j,k] = sum(a[i,:,j] * b[:,j,k])

Parameters

a [array_like] First argument.

b [array_like] Second argument.

See also:

numpy.einsum[Page 76, 38]

Notes

if a and b have the same shape this is the same as

np.concatenate([np.dot(a[...,i], b[...,i])[:, :, None] for i in range(n)], axis=2)

Examples

3 x 3 x 2 times 3 x 3 x 2 array -> 3 x 2 x 2 array

```
>>> import numpy as np
>>> import nvector as nv
>>> a = 1.0 * np.arange(18).reshape(3,3,2)
>>> b = - a
>>> t = np.concatenate([np.dot(a[...,i], b[...,i])[:, :, None]
...                     for i in range(2)], axis=2)
>>> tm = nv.mdot(a, b)
>>> tm.shape
(3, 3, 2)
>>> np.allclose(t, tm)
True
```

3 x 3 x 2 times 3 x 1 array -> 3 x 1 x 2 array

```
>>> t1 = np.concatenate([np.dot(a[...,i], b[:,0,0][:,None])[:,:,None]
...                     for i in range(2)], axis=2)
```

```
>>> tm1 = nv.mdot(a, b[:,0,0].reshape(-1,1))
>>> tm1.shape
(3, 1, 2)
>>> np.allclose(t1, tm1)
True
```

3 x 3 times 3 x 3 array -> 3 x 3 array

```
>>> tt0 = nv.mdot(a[...,0], b[...,0])
>>> tt0.shape
(3, 3)
>>> np.allclose(t[...,0], tt0)
True
```

3 x 3 times 3 x 1 array -> 3 x 1 array

```
>>> tt0 = nv.mdot(a[...,0], b[:,:1,0])
>>> tt0.shape
(3, 1)
>>> np.allclose(t[:,:1,0], tt0)
True
```

3 x 3 times 3 x 1 x 2 array -> 3 x 1 x 2 array

```
>>> tt0 = nv.mdot(a[..., 0], b[:, :2, 0][:, None])
>>> tt0.shape
(3, 1, 2)
>>> np.allclose(t[:,:2,0][:,None], tt0)
True
```

[38] https://numpy.org/doc/stable/reference/generated/numpy.einsum.html#numpy.einsum

5.4.3 nvector.util.nthroot

nthroot(*x*, *n*)

Returns the n'th root of x to machine precision

Parameters

x, n

Examples

```
>>> import numpy as np
>>> import nvector as nv
>>> np.allclose(nv.nthroot(27.0, 3), 3.0)
True
```

5.4.4 nvector.util.rad

rad(**deg_angles*)

Converts angle in degrees to radians.

Parameters

deg_angles: angle in degrees

Returns

rad_angles: angle in radians

See also:

deg

Examples

```
>>> import numpy as np
>>> import nvector as nv
>>> nv.deg(nv.rad(90))
90.0
>>> nv.deg(*nv.rad(90, [0, 180]))
(90.0, array([  0., 180.]))
```

5.4.5 nvector.util.get_ellipsoid

get_ellipsoid(*name*)

Returns semi-major axis (a), flattening (f) and name of ellipsoid as a named tuple.

Parameters

name [string] name of ellipsoid. Valid options are: 1) Airy 1858 2) Airy Modified 3) Australian National 4) Bessel 1841 5) Clarke 1880 6) Everest 1830 7) Everest Modified 8) Fisher 1960 9) Fisher 1968 10) Hough 1956 11) International (Hayford)/European Datum (ED50) 12) Krassovsky 1938 13) NWL-9D (WGS 66) 14) South American 1969 15) Soviet Geod. System 1985 16) WGS 72 17) Clarke 1866 (NAD27) 18) GRS80 / WGS84 (NAD83) 19) ETRS89

Examples

```
>>> import nvector as nv
>>> nv.get_ellipsoid(name='wgs84')
Ellipsoid(a=6378137.0, f=0.0033528106647474805, name='GRS80 / WGS84  (NAD83)')
>>> nv.get_ellipsoid(name='GRS80')
Ellipsoid(a=6378137.0, f=0.0033528106647474805, name='GRS80 / WGS84  (NAD83)')
>>> nv.get_ellipsoid(name='NAD83')
Ellipsoid(a=6378137.0, f=0.0033528106647474805, name='GRS80 / WGS84  (NAD83)')
>>> nv.get_ellipsoid(name=18)
Ellipsoid(a=6378137.0, f=0.0033528106647474805, name='GRS80 / WGS84  (NAD83)')
```

```
>>> wgs72 = nv.select_ellipsoid(name="WGS 72")
>>> wgs72.a == 6378135.0
True
>>> wgs72.f == 0.003352779454167505
True
>>> wgs72.name
'WGS 72'
>>> wgs72 == (6378135.0, 0.003352779454167505, 'WGS 72')
True
```

5.4.6 nvector.util.select_ellipsoid

select_ellipsoid(**args*, ***kwds*)

select_ellipsoid is deprecated, use *get_ellipsoid* instead!

Returns semi-major axis (a), flattening (f) and name of ellipsoid as a named tuple.

Parameters

name [string] name of ellipsoid. Valid options are: 1) Airy 1858 2) Airy Modified 3) Australian National 4) Bessel 1841 5) Clarke 1880 6) Everest 1830 7) Everest Modified 8) Fisher 1960 9) Fisher 1968 10) Hough 1956 11) International (Hayford)/European Datum (ED50) 12) Krassovsky 1938 13) NWL-9D (WGS 66) 14) South American 1969 15) Soviet Geod. System 1985 16) WGS 72 17) Clarke 1866 (NAD27) 18) GRS80 / WGS84 (NAD83) 19) ETRS89

Examples

```
>>> import nvector as nv
>>> nv.get_ellipsoid(name='wgs84')
Ellipsoid(a=6378137.0, f=0.0033528106647474805, name='GRS80 / WGS84  (NAD83)')
>>> nv.get_ellipsoid(name='GRS80')
Ellipsoid(a=6378137.0, f=0.0033528106647474805, name='GRS80 / WGS84  (NAD83)')
>>> nv.get_ellipsoid(name='NAD83')
Ellipsoid(a=6378137.0, f=0.0033528106647474805, name='GRS80 / WGS84  (NAD83)')
>>> nv.get_ellipsoid(name=18)
Ellipsoid(a=6378137.0, f=0.0033528106647474805, name='GRS80 / WGS84  (NAD83)')
```

```
>>> wgs72 = nv.select_ellipsoid(name="WGS 72")
>>> wgs72.a == 6378135.0
True
>>> wgs72.f == 0.003352779454167505
True
```

(continues on next page)

(continued from previous page)

```
>>> wgs72.name
'WGS 72'
>>> wgs72 == (6378135.0, 0.003352779454167505, 'WGS 72')
True
```

5.4.7 nvector.util.unit

unit(*vector*, *norm_zero_vector=1*)

Convert input vector to a vector of unit length.

Parameters

vector [3 x m array] m column vectors

Returns

unitvector [3 x m array] normalized unitvector(s) along axis==0.

Notes

The column vector(s) that have zero length will be returned as unit vector(s) pointing in the x-direction, i.e, [[1], [0], [0]]

Examples

```
>>> import numpy as np
>>> import nvector as nv
>>> np.allclose(nv.unit([[1, 0],[1, 0],[1, 0]]), [[ 0.57735027, 1],
...                                               [ 0.57735027, 0],
...                                               [ 0.57735027, 0]])
True
```

APPENDIX

A

CHANGELOG

A.1 Version 0.7.7, June 3, 2021

Per A Brodtkorb (27):

- Added cartopy and matplotlib to requirements.txt
- Updated appveyor.yml, setup.cfg and setup.py
- Updated .gitignore to ignore .pytest_cache
- Corrected failing doctests in objects.py
- Updated version in _installation.py
- Updated failing docstrings for python 2.7 in objects.py.
- Added '# doctest: SKIP' to all plt.show() in order to avoid the doctests hangs on the testserver.
- Fixed a bug in _info_functional.py
- Updated pycodestyle exlude section in setup.cfg Prettified _examples.py, _examples_object_oriented.py and core.py
- Updated pycodestyle ignore section in setup.cfg
- Added doctest option to setup.cfg
- Removed print statements in test_objects.py
- Return "NotImplemented" instead of raising "NotImplementedError" in Nvector._mul__ and Nvector.__div__ in objects.py
- **Fixed .travis.yml so that he file paths in coverage.xml is discoverable** under the sonar.sources folder. The problem is that SonarQube is analysing the checked-out source code (in src/nvector) but the actual unit tests and coverage.py is run against the installed code (in build/lib/nvector). Thus the absolute files paths to the installed code in the generated coverage.xml were causing Sonar to show no coverage. The workaround was to use sed in the pipeline to replace every path to build/lib/nvector with src/nvector in coverage.xml.
- Fixed a bug: Identical expressions should not be used on both sides of a binary operator in test:objects.py.
- Updated solutions to example 9
- Added greatcircle method to GeoPoint.distance_and_azimuth in objects.py
- Added _base_angle function that makes sure an angle is between -pi and pi.
- Added test_direct_and_inverse in test_objects.py
- Added interp_nvectors to docs/reference/nvector_summary.rst
- Added vectorized interpolation routines: interp_nvectors function to core.py and Nvector.interpolate to objects.py.

- **Put try except around code in use_docstring to avoid attribute '__doc__'** of 'type' objects is not writable errors for python2.
- Added interp_nvectors
- Reorganized _displace_great_circle
- Added check that depths also are equal on in _on_ellipsoid_path and in _on_great_circle_path
- **Refactored code from use_docstring_from function into the use_docstring** function in _common.py
- Simplified the adding of examples to the docstrings of functions and classes in core.py and objects.py.

A.2 Version 0.7.6, December 18, 2020

Per A Brodtkorb (30):

- Renamed _core.py to core.py
- Removed the module index from the appendix because it was incomplete.
- Removed nvector.tests package from the reference chapter.
- Added indent function to _common.py to avoid failure on python 2.7.
- Moved isclose, allclose and array_to_list_dict from objects.py to util.py
- **Moved the following function from test_nvector.py to test_rotation.py:**
  - test_n_E_and_wa2R_EL, test_R2zxy, test_R2zxy_x90, test_R2zxy_y90
  - test_R2zxy_z90, test_R2zxy_0, test_R2xyz test_R2xyz_with_vectors
- Replaced assert_array_almost_equal with assert_allclose in test_objects.py
- Renamed test_frames.py to test_objects.py
- Added missing functions great_circle_normal and interpolate to the nvector_summary.rst
- **Moved the following functions related to rotation matrices from _core to rotation module:**
  - E_rotation, n_E_and_wa2R_EL, n_E2R_EN, R_EL2n_E, R_EN2n_E, R2xyz, R2zyx, xyz2R, zyx2R
- Renamed select_ellipsoid to get_ellipsoid
- **Moved the following utility functions from _core to util module:**
  - deg, rad, mdot, nthroot, get_ellipsoid, unit, _check_length_deviation
- Added _get_h1line and _make_summary to _common.py
- Replaced numpy.rollaxis with numpy.swapaxes to make the code clearer.
- _atleast_3d now broadcast the input against each other.
- Added examples to zyx2R
- **Added the following references to zyx2R, xyz2R, R2xyz, R2zyx:**
  - https://en.wikipedia.org/wiki/Aircraft_principal_axes
  - https://en.wikipedia.org/wiki/Euler_angles
  - https://en.wikipedia.org/wiki/Axes_conventions
- Removed tabs from CHANGELOG.rst
- Updated CHANGELOG.rst and prepared for release v0.7.6
- Fixed the documentation so that it shows correctly in the reference manual.

- Added logo.png and docs/reference/nvector.rst
- Updated build_package.py so it generates a valid README.rst file.
- Updated THANKS.rst
- Updated CHANGELOG.rst and prepare for release 0.7.6
- Added Nvector documentation ref https://nvector.readthedocs.io/en/v0.7.5 to refs1.bib and _acknowledgements.py
- Updated README.rst
- Renamed requirements.readthedocs.txt to docs/requirements.txt
- Added .readthedocs.yml
- Added sphinxcontrib-bibtex to requirements.readthedocs.txt
- Added missing docs/tutorials/images/ex3img.png
- Deleted obsolete ex10img.png
- Updated acknowledgement with reference to Karney's article.
- Updated README.rst by moving acknowledgement to the end with references.
- Renamed position input argument to point in the FrameN, FrameB and FrameL classes.
- Deleted _example_images.py
- Renamed nvector.rst to nvector_summary.rst in docs/reference
- Added example images to tutorials/images/ folder
- Added Nvector logo, install.rst to docs
- Added src/nvector/_example_images.py
- Added docs/tutorials/whatsnext.rst
- **Reorganized the documentation in docs by splitting _info.py into:**
  - _intro.py,
  - _documentation.py
  - _examples_object_oriented.py
  - _images.py
  - _installation.py and _acknowledgements.py
- Added docs/tutorials/index.rst, docs/intro/index.rst, docs/how-to/index.rst docs/appendix/index.rst and docs/make.bat
- updated references.

A.3 Version 0.7.5, December 12, 2020

Per A Brodtkorb (32):

- Updated CHANGELOG.rst and prepare for release 0.7.5
- **Changed so that GeoPath.on_great_circle and GeoPath.on_great_circle** returns scalar result if the two points defining the path are scalars. See issue #10.
- Fixed failing doctests.
- Added doctest configuration to docs/conf.py
- Added allclose to nvector/objects.py

- **Added array_to_list_dict and isclose functions in nvector.objects.py** Replaced f-string in the __repr__ method of the _Common class in nvector.objects.py with format in order to work on python version 3.5 and below.
- Made nvector.plot.py more robust.
- Removed rtol parameter from the on_greatcircle function. See issue #12 for a discussion.
- Added nvector solution to the GeoPoint.displace method.
- Updated docs/conf.py
- Updated README.rst and LICENSE.txt
- Replaced import unittest with import pytest in test_frames.py
- **Fixed issue #10: Inconsistent return types in GeoPath.track_distance:**
  - GeoPath, GeoPoint, Nvector and ECEFvector and Pvector now return scalars for the case where the input is not actually arrays of points but just single objects.
- Added extra tests for issue #10 and updated old tests and the examples in the help headers.
- Vectorized FrameE.inverse and FrameE.direct methods.
- Extended deg and rad functions in _core.py.
- Vectorized GeoPoint.distance_and_azimuth
- Made import of cartopy in nvector.plot more robust.
- Updated test_Ex10_cross_track_distance
- Updated sonar-project.properties
- Replaced deprecated sonar.XXXX.reportPath with sonar.XXXX.reportPaths
- Simplified nvector/_core.__doc__
- Updated .travis.yml
- Changed the definition of sonar addon
- Added CC_TEST_REPORTER_ID to .travis.yml
- Added python 3.8 to the CI testing.
- Changed so that setup.py is python 2.7 compatible again.
- Updated build_package.py
- Renamed CHANGES.rst to CHANGELOG.rst
- Updated setup.cfg and setup.py
- Added license.py
- Updated build_package.py
- Removed conda-build from .travis.yml
- Attempt to get travis to run the tests again....
- API change: replaced "python setup.py doctests" with "python setup.py doctest"
- Added doctest example to nvector._core._atleast_3d Made xyz2R and zyx2R code simpler.
- Replaced deprecated Nvector.mean_horizontal_position with Nvector.mean in test_frames.py
- Added mdot to __all__ in nvector/_core.py and in documentation summary.
- Sorted the the documentation summary by function name in nvector.rst
- Removed –pyargs nvector –doctest-modules –pep8 from addopts section in setup.cfg
- Updated documentation and added missing documentation.

A.4 Version 0.7.4, June 4, 2019

Per A Brodtkorb (2):

- Fixed PyPi badge and added downloads badge in nvector/_info.py and README.rst
- Removed obsolete and wrong badges from docs/index.rst

A.5 Version 0.7.3, June 4, 2019

Per A Brodtkorb (6):

- Renamed LICENSE.txt and THANKS.txt to LICENSE.rst and THANKS.rst
- Updated README.rst and nvector/_info.py
- Fixed issue 7# incorrect test for test_n_E_and_wa2R_EL.
- Removed coveralls test coverage report.
- Replaced coverage badge from coveralls to codecov.
- Updated code-climate reporter.
- Simplified duplicated code in nvector._core.
- Added tests/__init__.py
- Added "–pyargs nvector" to pytest options in setup.cfg
- Exclude build_package.py from distribution in MANIFEST.in
- Replaced health_img from landscape to codeclimate.
- Updated travis to explicitly install pytest-cov and pytest-pep8
- Removed dependence on pyscaffold
- Added MANIFEST.in
- Renamed set_package_version.py to build_package.py

A.6 Version 0.7.0, June 2, 2019

Gary van der Merwe (1):

- Add interpolate to __all__ so that it can be imported

Per A Brodtkorb (26):

- Updated long_description in setup.cfg
- Replaced deprecated sphinx.ext.pngmath with sphinx.ext.imgmath
- Added imgmath to requirements for building the docs.
- Fixing shallow clone warning.
- **Replaced property 'sonar.python.coverage.itReportPath' with** 'sonar.python.coverage.reportPaths' instead, because it is has been removed.
- Drop python 3.4 support
- Added python 3.7 support
- Fixed a bug: Mixed scalars and np.array([1]) values don't work with np.rad2deg function.

- **Added ETRS ELLIPSOID in _core.py Added ED50 as alias for International** (Hayford)/European Datum in _core.py Added sad69 as alias for South American 1969 in _core.py
- Simplified docstring for nv.test
- Generalized the setup.py.
- Replaced aliases with the correct names in setup.cfg.

A.7 Version 0.6.0, December 9, 2018

Per A Brodtkorb (79):

- Updated requirements in setup.py
- Removed tox.ini
- Updated documentation on how to set package version
- Made a separate script to set package version in nvector/__init__.py
- Updated docstring for select_ellipsoid
- Replace GeoPoint.geo_point with GeoPoint.displace and removed deprecated GeoPoint.geo_point
- Update .travis.yml
- Fix so that codeclimate is able to parse .travis.yml
- Only run sonar and codeclimate reporter for python v3.6
- Added sonar-project.properties
- **Pinned coverage to v4.3.4 due to fact that codeclimate reporter is only** compatible with Coverage.py versions >=4.0,<4.4.
- Updated with sonar scanner.
- Added .pylintrc
- Set up codeclimate reporter
- Updated docstring for unit function.
- Avoid division by zero in unit function.
- Reenabled the doctest of plot_mean_position
- Reset "pyscaffold==2.5.11"
- Replaced deprecated basemap with cartopy.
- **Replaced doctest of plot_mean_position with test_plot_mean_position in** test_plot.py
- **Fixed failing doctests for python v3.4 and v3.5 and made them more** robust.
- Fixed failing doctests and made them more robust.
- Increased pycoverage version to use.
- moved nvector to src/nvector/
- **Reset the setup.py to require 'pyscaffold==2.5.11' which works on** python version 3.4, 3.5 and 3.6. as well as 2.7
- Updated unittests.
- Updated tests.
- Removed obsolete code
- Added test for delta_L

- **Added corner testcase for** pointA.displace(distance=1000,azimuth=np.deg2rad(200))
- Added test for path.track_distance(method='exact')
- **Added delta_L a function thet teturn cartesian delta vector from** positions A to B decomposed in L.
- Simplified OO-solution in example 1 by using delta_N function
- Refactored duplicated code
- **Vectorized code so that the frames can take more than one position at** the time.
- Keeping only the html docs in the distribution.
- **replaced link from latest to stable docs on readthedocs and updated** crosstrack distance test.
- updated documentation in setup.py

A.8 Version 0.5.2, March 7, 2017

Per A Brodtkorb (10):

- Fixed tests in tests/test_frames.py
- Updated to setup.cfg and tox.ini + pep8
- updated .travis.yml
- Updated Readme.rst with new example 10 picture and link to nvector docs at readthedocs.
- updated official documentation links
- Updated crosstrack distance tests.

A.9 Version 0.5.1, March 5, 2017

Cody (4):

- Explicitely numbered replacement fields
- Migrated % string formating

Per A Brodtkorb (29):

- pep8
- Updated failing examples
- Updated README.rst
- Removed obsolete pass statement
- Documented functions
- added .checkignore for quantifycode
- moved test_docstrings and use_docstring_from into _common.py
- Added .codeclimate.yml
- Updated installation information in _info.py
- Added GeoPath.on_path method. Clearified intersection example
- Added great_circle_normal, cross_track_distance
- Renamed intersection to intersect (Intersection is deprecated.)

- Simplified R2zyx with a call to R2xyz Improved accuracy for great circle cross track distance for small distances.
- Added on_great_circle, _on_great_circle_path, _on_ellipsoid_path, closest_point_on_great_circle and closest_point_on_path to GeoPath
- made __eq__ more robust for frames
- Removed duplicated code
- Updated tests
- Removed fishy test
- replaced zero n-vector with nan
- Commented out failing test.
- Added example 10 image
- Added 'closest_point_on_great_circle', 'on_great_circle','on_great_circle_path'.
- Updated examples + documentation
- Updated index depth
- Updated README.rst and classifier in setup.cfg

A.10 Version 0.4.1, January 19, 2016

pbrod (46):

- Cosmetic updates
- Updated README.rst
- updated docs and removed unused code
- updated README.rst and .coveragerc
- Refactored out _check_frames
- Refactored out _default_frame
- Updated .coveragerc
- Added link to geographiclib
- Updated external link
- Updated documentation
- Added figures to examples
- Added GeoPath.interpolate + interpolation example 6
- Added links to FFI homepage.
- **Updated documentation:**
  - Added link to nvector toolbox for matlab
  - For each example added links to the more detailed explanation on the homepage
- Updated link to nvector toolbox for matlab
- Added link to nvector on pypi
- Updated documentation fro FrameB, FrameE, FrameL and FrameN.
- updated __all__ variable
- Added missing R_Ee to function n_EA_E_and_n_EB_E2azimuth + updated documentation

- Updated CHANGES.rst
- Updated conf.py
- Renamed info.py to _info.py
- All examples are now generated from _examples.py.

A.11 Version 0.1.3, January 1, 2016

pbrod (31):

- Refactored
- Updated tests
- Updated docs
- Moved tests to nvector/tests
- Updated .coverage Added travis.yml, .landscape.yml
- Deleted obsolete LICENSE
- Updated README.rst
- Removed ngs version
- Fixed bug in .travis.yml
- Updated .travis.yml
- Removed dependence on navigator.py
- Updated README.rst
- Updated examples
- Deleted skeleton.py and added tox.ini
- Renamed distance_rad_bearing_rad2point to n_EA_E_distance_and_azimuth2n_EB_E
- Renamed azimuth to n_EA_E_and_n_EB_E2azimuth
- Added tests for R2xyz as well as R2zyx
- Removed backward compatibility
- Added test_n_E_and_wa2R_EL
- Refactored tests
- Commented out failing tests on python 3+
- updated CHANGES.rst
- Removed bug in setup.py

A.12 Version 0.1.1, January 1, 2016

pbrod (31):

- Initial commit: Translated code from Matlab to Python.
- Added object oriented interface to nvector library
- Added tests for object oriented interface
- Added geodesic tests.

APPENDIX

B

DEVELOPERS

- Kenneth Gade, FFI
- Kristian Svartveit, FFI
- Brita Hafskjold Gade, FFI
- Per A. Brodtkorb FFI

APPENDIX

C

LICENSE

```
The content of this library is based on the following publication:

Gade, K. (2010). A Nonsingular Horizontal Position Representation, The Journal
of Navigation, Volume 63, Issue 03, pp 395-417, July 2010.
(www.navlab.net/Publications/A_Nonsingular_Horizontal_Position_Representation.pdf)

This paper should be cited in publications using this library.

Copyright (c) 2015-2021, Norwegian Defence Research Establishment (FFI)
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above publication
information, copyright notice, this list of conditions and the following
disclaimer.

2. Redistributions in binary form must reproduce the above publication
information, copyright notice, this list of conditions and the following
disclaimer in the documentation and/or other materials provided with the
distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS
BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF
THE POSSIBILITY OF SUCH DAMAGE.
```

APPENDIX

D

ACKNOWLEDGMENTS

The nvector package[39] for Python[40] was written by Per A. Brodtkorb at FFI (The Norwegian Defence Research Establishment)[41] based on the nvector toolbox[42] for Matlab[43] written by the navigation group at FFI[44]. The nvector.core and nvector.rotation module is a vectorized reimplementation of the matlab nvector toolbox while the nvector.objects module is a new easy to use object oriented user interface to the nvector core functionality documented in [GB20].

Most of the content is based on the article by K. Gade [Gad10].

Thus this article should be cited in publications using this page or downloaded program code.

However, if you use any of the FrameE.direct, FrameE.inverse, GeoPoint.distance_and_azimuth or GeoPoint.displace methods you should also cite the article by Karney [Kar13] because these methods call Karney's geographiclib[45] library to do the calculations.

[39] http://pypi.python.org/pypi/nvector/
[40] https://www.python.org/
[41] http://www.ffi.no/en
[42] http://www.navlab.net/nvector/#download
[43] http://www.mathworks.com
[44] http://www.ffi.no/en
[45] https://pypi.python.org/pypi/geographiclib

BIBLIOGRAPHY

[Gad10] Kenneth Gade. A nonsingular horizontal position representation. *The Journal of Navigation*, 63(3):395–417, 2010. URL: http://www.navlab.net/Publications/A_Nonsingular_Horizontal_Position_Representation.pdf.

Kenneth Gade. A nonsingular horizontal position representation. *The Journal of Navigation*, 63(3):395–417, 2010. URL: http://www.navlab.net/Publications/A_Nonsingular_Horizontal_Position_Representation.pdf.

[Gad16] Kenneth Gade. The seven ways to find heading. *The Journal of Navigation*, 69():955–970, 2016. URL: https://www.navlab.net/Publications/The_Seven_Ways_to_Find_Heading.pdf.

Kenneth Gade. The seven ways to find heading. *The Journal of Navigation*, 69():955–970, 2016. URL: https://www.navlab.net/Publications/The_Seven_Ways_to_Find_Heading.pdf.

[Gad18] Kenneth Gade. *Inertial Navigation - Theory and Applications.* PhD thesis, Norwegian Univiersity of Science and Technology, 1 2018. ISBN 978-82-326-2871-1. URL: https://www.navlab.net/Publications/Inertial_Navigation_-_Theory_and_Applications.pdf.

Kenneth Gade. *Inertial Navigation - Theory and Applications.* PhD thesis, Norwegian Univiersity of Science and Technology, 1 2018. ISBN 978-82-326-2871-1. URL: https://www.navlab.net/Publications/Inertial_Navigation_-_Theory_and_Applications.pdf.

[GB20] Kenneth Gade and Per A. Brodtkorb. Nvector documentation for python. Online user manual., 2020. URL: https://nvector.readthedocs.io/en/v0.7.6.

Kenneth Gade and Per A. Brodtkorb. Nvector documentation for python. Online user manual., 2020. URL: https://nvector.readthedocs.io/en/v0.7.6.

[Kar13] C. F. F. Karney. Algorithms for geodesics. *Journal of Geodesy*, 87(1):43–55, 2013. URL: https://rdcu.be/cccgm.

C. F. F. Karney. Algorithms for geodesics. *Journal of Geodesy*, 87(1):43–55, 2013. URL: https://rdcu.be/cccgm.

INDEX

Symbols

C

D

E

F

G

I

L

M

N

O

P

R

S

U

X

Z