nvector Documentation

Release 0.7.0

Kenneth Gade and Per A Brodtkorb

Jun 03, 2019

Contents

This is the documentation of **nvector** 0.7 for Python.

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 1

Contents

1.1 Introduction to Nvector

pypi package ?.?.?

Nvector is a suite of tools written in Python to solve geographical position calculations like:

- 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.

1.2 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 here:

1.3 Documentation and code

Official documentation:

http://www.navlab.net/nvector/

http://nvector.readthedocs.io/en/latest/

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

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

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

1.4 Installation

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

1.5 Unit tests

To test if the toolbox is working 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.

1.6 Acknowledgement

The nvector package for Python was written by Per A. Brodtkorb at FFI (The Norwegian Defence Research Establishment) based on the nvector toolbox for Matlab written by the navigation group at FFI.

Most of the content is based on the following article:

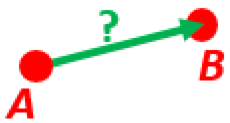
Kenneth Gade (2010): A Nonsingular Horizontal Position Representation, The Journal of Navigation, Volume 63, Issue 03, pp 395-417, July 2010.

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

1.7 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 here.

1.7.1 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. 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()
>>> valtxt = '{0:8.2f}, {1:8.2f}, {2:8.2f}'.format(x, y, z)
>>> 'Ex1: delta north, east, down = {}'.format(valtxt)
'Ex1: delta north, east, down = 331730.23, 332997.87, 17404.27'
```

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

```
>>> azimuth = p_AB_N.azimuth_deg[0]
>>> 'azimuth = {0:4.2f} deg'.format(azimuth)
'azimuth = 45.11 deg'
```

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()
>>> valtxt = '{0:8.2f}, {1:8.2f}, {2:8.2f}'.format(*p_AB_N)
>>> 'Ex1: delta north, east, down = {}'.format(valtxt)
'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(p_AB_N[1], p_AB_N[0])
>>> 'azimuth = {0:4.2f} deg'.format(deg(azimuth))
'azimuth = 45.11 deg'
```

See also Example 1 at www.navlab.net

1.7.2 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 nvector as nv
>>> import numpy as np
>>> 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()
```

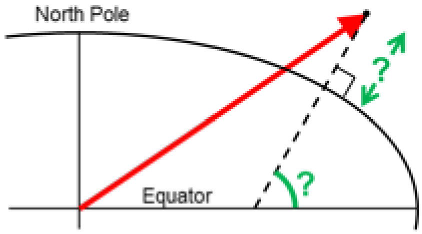
Step 4: Find point C by adding delta BC to EB

```
>>> 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 = {:4.2f}, {:4.2f} deg,  height = {:4.2f} m'
>>> msg.format(lat[0], lon[0], -z[0])
'Ex2: PosC: lat, lon = 53.33, 63.47 deg,  height = 406.01 m'
```

See also Example 2 at www.navlab.net

1.7.3 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.

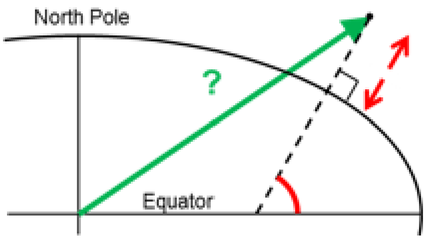
Solution:

```
>>> import numpy as np
>>> import nvector as nv
>>> 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
>>> msg = 'Ex3: Pos B: lat, lon = {:4.2f}, {:4.2f} deg, height = {:9.2f} m'
>>> msg.format(lat[0], lon[0], -z[0])
'Ex3: Pos B: lat, lon = 39.38, -48.01 deg, height = 4702059.83 m'
```

See also Example 3 at www.navlab.net

1.7.4 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.

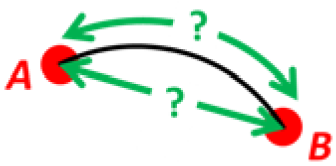
Solution:

```
>>> 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

1.7.5 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
>>> 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[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'
```

Alternative sphere solution:

```
>>> path = nv.GeoPath(positionA, positionB)
>>> s_AB2 = path.track_distance(method='greatcircle').ravel()
>>> d_AB2 = path.track_distance(method='euclidean').ravel()
>>> msg.format(s_AB2[0] / 1000, d_AB2[0] / 1000)
'Ex5: Great circle and Euclidean distance = 332.46 km, 332.42 km'
```

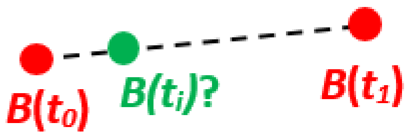
Exact solution for the WGS84 ellipsoid:

```
>>> 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[0]
>>> 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

1.7.6 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
>>> 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)
```

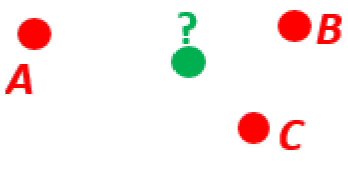
```
>>> t0 = 10.
>>> t1 = 20.
>>> ti = 16.  # time of interpolation
>>> ti_n = (ti - t0) / (t1 - t0) # normalized time of interpolation
```

```
>>> 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[0], lon_ti[0])
'Ex6, Interpolated position: lat, lon = 89.8 deg, 180.0 deg'
```

See also Example 6 at www.navlab.net

1.7.7 Example 7: "Mean position"



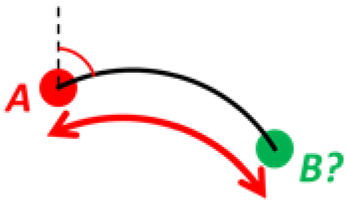
Three positions A, B, and C are given as n-vectors n_EA_E, n_EB_E, and n_EC_E. Find the mean position, M, given as n_EM_E. Note that the calculation is independent of the depths of the positions.

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 = {:4.2f}, {:4.2f} deg'
>>> msg.format(lat[0], lon[0])
'Ex7: Pos M: lat, lon = 67.24, -6.92 deg'
```

See also Example 7 at www.navlab.net

1.7.8 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, 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 and 5.)

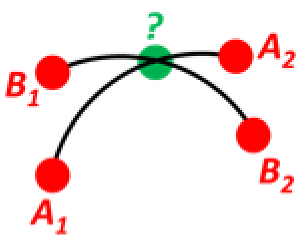
Solution:

```
>>> 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 = {:4.2f} deg, {:4.2f} deg'
>>> msg.format(lat, lon)
'Ex8, Destination: lat, lon = 79.99 deg, -90.02 deg'
```

See also Example 8 at www.navlab.net

1.7.9 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.

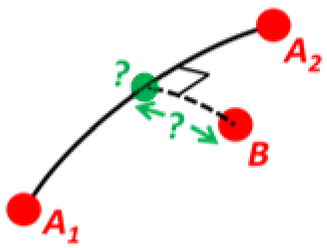
Solution:

```
>>> 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)
>>> np.allclose(pathA.on_path(pointC), pathB.on_path(pointC))
True
>>> np.allclose(pathA.on_great_circle(pointC),
...             pathB.on_great_circle(pointC))
True
>>> pointC = pointC.to_geo_point()
>>> lat, lon = pointC.latitude_deg, pointC.longitude_deg
>>> msg = 'Ex9, Intersection: lat, lon = {:4.2f}, {:4.2f} deg'
>>> msg.format(lat[0], lon[0])
'Ex9, Intersection: lat, lon = 40.32, 55.90 deg'
```

See also Example 9 at www.navlab.net

1.7.10 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 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').ravel()
>>> d_xt = pathA.cross_track_distance(pointB, method='euclidean').ravel()
```

```
>>> 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 = {}'.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

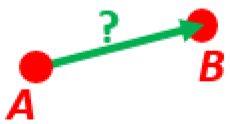
1.8 See also

geographiclib

1.9 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 here.

1.9.1 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. 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()
>>> valtxt = '{0:8.2f}, {1:8.2f}, {2:8.2f}'.format(*p_AB_N)
>>> 'Ex1: delta north, east, down = {}'.format(valtxt)
'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(p_AB_N[1], p_AB_N[0])
>>> 'azimuth = {0:4.2f} deg'.format(deg(azimuth))
'azimuth = 45.11 deg'
```

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

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

```
>>> azimuth = p_AB_N.azimuth_deg[0]
>>> 'azimuth = {0:4.2f} deg'.format(azimuth)
'azimuth = 45.11 deg'
```

See also Example 1 at www.navlab.net

1.9.2 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):

```
>>> 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)
```

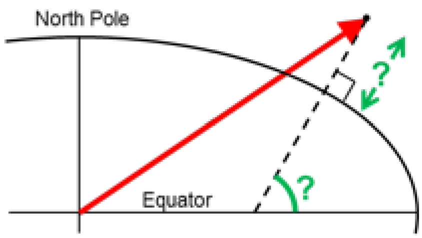
Step 6: Find the position of C, using the functions that goes from one

```
>>> 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 = {:4.2f}, {:4.2f} deg,  height = {:4.2f} m'
>>> msg.format(lat[0], lon[0], -z[0])
'Ex2: PosC: lat, lon = 53.33, 63.47 deg,  height = 406.01 m'
```

See also Example 2 at www.navlab.net

1.9.3 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
>>> wgs84 = dict(a=6378137.0, f=1.0/298.257223563)
>>> p_EB_E = 6371e3 * np.vstack((0.9, -1, 1.1))  # m
```

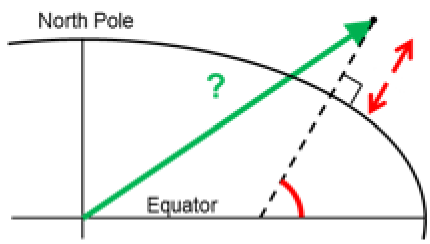
```
>>> n_EB_E, z_EB = nv.p_EB_E2n_EB_E(p_EB_E, **wgs84)
```

```
>>> 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 = {:4.2f}, {:4.2f} deg, height = {:9.2f} m'
>>> msg.format(lat[0], lon[0], h[0])
'Ex3: Pos B: lat, lon = 39.38, -48.01 deg, height = 4702059.83 m'
```

See also Example 3 at www.navlab.net

1.9.4 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.

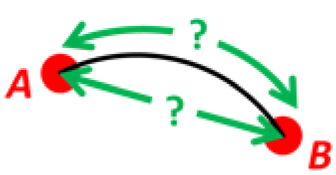
Solution:

```
>>> 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

1.9.5 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[0]
>>> 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

1.9.6 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))
```

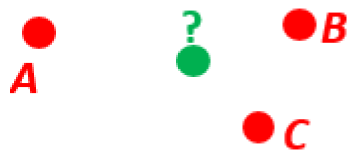
```
>>> t0 = 10.
>>> t1 = 20.
>>> ti = 16.  # time of interpolation
>>> ti_n = (ti - t0) / (t1 - t0) # normalized time of interpolation
```

```
>>> 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'
```

See also Example 6 at www.navlab.net

1.9.7 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 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)
```

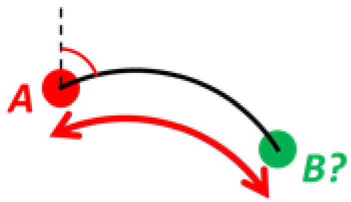
or

```
>>> 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 = {:4.2f}, {:4.2f} deg'
>>> msg.format(lat[0], lon[0])
'Ex7: Pos M: lat, lon = 67.24, -6.92 deg'
```

See also Example 7 at www.navlab.net

1.9.8 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, 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 and 5.)

Solution:

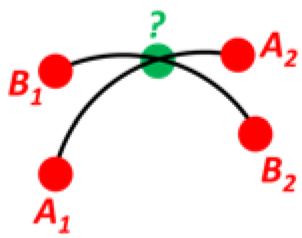
```
>>> import nvector as nv
>>> from nvector import rad, deg
>>> lat, lon = rad(80), rad(-90)
```

```
>>> 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 = {:4.2f} deg, {:4.2f} deg'
>>> msg.format(lat[0], lon[0])
'Ex8, Destination: lat, lon = 79.99 deg, -90.02 deg'
```

See also Example 8 at www.navlab.net

1.9.9 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)
```

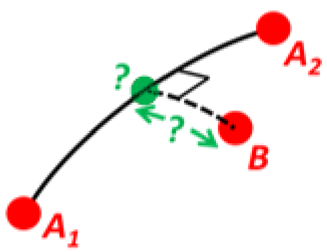
```
>>> lat_EC, lon_EC = nv.n_E2lat_lon(n_EC_E)
```

```
>>> lat, lon = deg(lat_EC), deg(lon_EC)
>>> msg = 'Ex9, Intersection: lat, lon = {:4.2f}, {:4.2f} deg'
>>> msg.format(lat[0], lon[0])
'Ex9, Intersection: lat, lon = 40.32, 55.90 deg'
```

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

See also Example 9 at www.navlab.net

1.9.10 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
>>> n_EA1_E = nv.lat_lon2n_E(rad(0), rad(0))
```

(continues on next page)

(continued from previous page)

```
>>> 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)
```

See also Example 10 at www.navlab.net

1.10 License

```
=======
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, 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.
```

(continues on next page)

(continued from previous page)

```
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.
```

1.11 Developers

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

1.12 Changelog

1.12.1 Version 0.7.0, June 02, 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.png.math with sphinx.ext.img.math
- Added pngmath 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.

1.12.2 Version 0.6.0, December 09, 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

1.12.3 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.

1.12.4 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

1.12.5 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.

1.12.6 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
- Small refactoring Renamed distance_rad_bearing_rad2point to n_EA_E_distance_and_azimuth2n_EB_E updated tests
- 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

1.12.7 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.

1.13 Modules

Release 0.7

Date Jun 03, 2019

This reference manual details functions, modules, and objects included in nvector, describing what they are and what they do.

1.13.1 nvector package

1.13.1.1 Geodesic functions

lat_lon2n_E(latitude, longitude[, R_Ee])	Converts latitude and longitude to n-vector.
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[, ...])	Return the delta vector from position A to B.
n_EA_E_and_p_AB_E2n_EB_E(n_EA_E, p_AB_E[, ...])	Return position B from position A and delta.
n_EA_E_and_n_EB_E2azimuth(n_EA_E, n_EB_E[, ...])	Return azimuth from A to B, relative to North:
n_EA_E_distance_and_azimuth2n_EB_E(n_EA_E, ...)	Return position B from azimuth and distance from position A
great_circle_distance(n_EA_E, n_EB_E[, radius])	Return great circle distance between positions A and B
euclidean_distance(n_EA_E, n_EB_E[, radius])	Return Euclidean distance between positions A and B
cross_track_distance(path, n_EB_E[, method, ...])	Return cross track distance between path A and position B.
closest_point_on_great_circle(path, n_EB_E)	Return closest point C on great circle path A to position B.
intersect(path_a, path_b)	Return the intersection(s) between the great circles of the two paths
mean_horizontal_position(n_EB_E)	Return the n-vector of the horizontal mean position.
on_great_circle(path, n_EB_E[, radius, ...])	True if position B is on great circle through path A.
on_great_circle_path(path, n_EB_E[, radius, ...])	True if position B is on great circle and between endpoints of path A.

1.13.1.1.1 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

1.13.1.1.2 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

1.13.1.1.3 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.

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 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.

1.13.1.1.4 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)

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.

1.13.1.1.5 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*)

Return the delta vector from position A to 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.

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.

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.

1.13.1.1.6 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=0.0033528106647474805*, *R_Ee=None*)

Return position B from position A and delta.

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.

1.13.1.1.7 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*)

Return 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.

1.13.1.1.8 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*)

Return 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.

1.13.1.1.9 nvector._core.great_circle_distance

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

Return 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.

Formulae is given by equation (16) in Gade (2010) and is well conditioned for all angles.

1.13.1.1.10 nvector._core.euclidean_distance

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

Return 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.

1.13.1.1.11 nvector._core.cross_track_distance

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

Return 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)

1.13.1.1.12 nvector._core.closest_point_on_great_circle

closest_point_on_great_circle (*path*, *n_EB_E*)
Return 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

1.13.1.1.13 nvector._core.intersect

intersect (*path_a*, *path_b*)
Return 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.

1.13.1.1.14 nvector._core.mean_horizontal_position

mean_horizontal_position (*n_EB_E*)

Return 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.

1.13.1.1.15 nvector._core.on_great_circle

on_great_circle (*path*, *n_EB_E*, *radius=6371009.0*, *rtol=1e-06*, *atol=1e-08*)
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)

rtol, atol: real scalars defining relative and absolute tolerance

Returns

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

1.13.1.1.16 nvector._core.on_great_circle_path

`on_great_circle_path` (*path*, *n_EB_E*, *radius=6371009.0*, *rtol=1e-06*, *atol=1e-08*)
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)

rtol, atol: real scalars defining relative and absolute tolerance

Returns

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

1.13.1.2 Rotation matrices and angles

E_rotation([axes])	Return 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.

1.13.1.2.1 nvector._core.E_rotation

`E_rotation` (*axes='e'*)
Return 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. Options are: 'e':

z-axis points to the North Pole along the Earth's rotation axis,

x-axis points towards the point where latitude = longitude = 0. This choice is very common in many fields.

'E': x-axis points to the North Pole along the Earth's rotation axis, y-axis points towards longitude +90deg (east) and latitude = 0. (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. The functions in this library originally used this option.

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)

R_Ee controls the axes of the coordinate frame E (Earth-Centred,

Earth-Fixed, ECEF) used by the other functions in this library

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
```

1.13.1.2.2 nvector._core.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*

1.13.1.2.3 nvector._core.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.

R_EL = n_E_and_wa2R_EL(n_E,wander_azimuth) Calculates the rotation matrix (direction cosine matrix) R_EL using n-vector (n_E) and the wander azimuth angle. When wander_azimuth=0, we have that N=L (See Table 2 in Gade (2010) for details)

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*

1.13.1.2.4 nvector._core.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*

1.13.1.2.5 nvector._core.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*

1.13.1.2.6 nvector._core.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.

1.13.1.2.7 nvector._core.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.

1.13.1.2.8 nvector._core.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.

1.13.1.2.9 nvector._core.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.

1.13.1.3 Misc functions

nthroot(x, n)	Return the n'th root of x to machine precision
deg(rad_angle)	Converts angle in radians to degrees.
rad(deg_angle)	Converts angle in degrees to radians.

Continued on next page

Table 3 – continued from previous page

select_ellipsoid(name)	Return semi-major axis (a), flattening (f) and name of ellipsoid
unit(vector[, norm_zero_vector])	Convert input vector to a vector of unit length.

1.13.1.3.1 nvector._core.nthroot

nthroot (*x, n*)
Return 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
```

1.13.1.3.2 nvector._core.deg

deg (*rad_angle*)
Converts angle in radians to degrees.

Parameters

rad_angle: angle in radians

Returns

deg_angle: angle in degrees

See also:

rad

1.13.1.3.3 nvector._core.rad

rad (*deg_angle*)
Converts angle in degrees to radians.

Parameters

deg_angle: angle in degrees

Returns

rad_angle: angle in radians

See also:

deg

1.13.1.3.4 nvector._core.select_ellipsoid

select_ellipsoid (*name*)
Return semi-major axis (a), flattening (f) and name of ellipsoid

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.select_ellipsoid(name='wgs84')
(6378137.0, 0.0033528106647474805, 'GRS80 / WGS84  (NAD83)')
>>> nv.select_ellipsoid(name='GRS80')
(6378137.0, 0.0033528106647474805, 'GRS80 / WGS84  (NAD83)')
>>> nv.select_ellipsoid(name='NAD83')
(6378137.0, 0.0033528106647474805, 'GRS80 / WGS84  (NAD83)')
>>> nv.select_ellipsoid(name=18)
(6378137.0, 0.0033528106647474805, 'GRS80 / WGS84  (NAD83)')
```

1.13.1.3.5 nvector._core.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
```

1.13.1.4 OO interface to Geodesic functions

FrameE([a, f, name, axes])	Earth-fixed frame
FrameN(position)	North-East-Down frame
FrameL(position[, wander_azimuth])	Local level, Wander azimuth frame
FrameB(position[, yaw, pitch, roll, degrees])	Body frame
ECEFvector(pvector[, frame])	Geographical position given as Cartesian position vector in frame E

Continued on next page

Table 4 – continued from previous page

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
GeoPath(positionA, positionB)	Geographical path between two positions in Frame E
Pvector(pvector, frame)	Cartesian position vector in another frame
diff_positions(*args, **kwds)	*diff_positions* is deprecated!

1.13.1.4.1 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__(*self*, *a=None*, *f=None*, *name='WGS84'*, *axes='e'*)

x.__init__(…) initializes x; see help(type(x)) for signature

Methods

ECEFvector(self, *args, **kwds)	Geographical position given as Cartesian position vector in frame E
GeoPoint(self, *args, **kwds)	Geographical position given as latitude, longitude, depth in frame E
Nvector(self, *args, **kwds)	Geographical position given as n-vector and depth in frame E
__init__(self[, a, f, name, axes])	x.__init__(…) initializes x; see help(type(x)) for signature
direct(self, lat_a, lon_a, azimuth, distance)	Return position B computed from position A, distance and azimuth.
inverse(self, lat_a, lon_a, lat_b, lon_b[, …])	Return ellipsoidal distance between positions as well as the direction.

1.13.1.4.2 nvector.objects.FrameN

class FrameN(*position*)

North-East-Down frame

Parameters

position: 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).

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.

`__init__`(*self*, *position*)
x.__init__(...) initializes x; see help(type(x)) for signature

Methods

`Pvector`(self, pvector)	
`__init__`(self, position)	x.__init__(...) initializes x; see help(type(x)) for signature

Attributes

`R_EN`

1.13.1.4.3 nvector.objects.FrameL

class FrameL(*position*, *wander_azimuth=0*)
Local level, Wander azimuth frame

Parameters

position: 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__`(*self*, *position*, *wander_azimuth=0*)
x.__init__(…) initializes x; see help(type(x)) for signature

Methods

`Pvector`(self, pvector)	
`__init__`(self, position[, wander_azimuth])	x.__init__(…) initializes x; see help(type(x)) for signature

Attributes

`R_EN`

1.13.1.4.4 nvector.objects.FrameB

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

Parameters

position: 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

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.

`__init__`(*self*, *position*, *yaw=0*, *pitch=0*, *roll=0*, *degrees=False*)
x.__init__(…) initializes x; see help(type(x)) for signature

Methods

`Pvector`(self, pvector)	
`__init__`(self, position[, yaw, pitch, roll, …])	x.__init__(…) initializes x; see help(type(x)) for signature

Attributes

`R_EN`

1.13.1.4.5 nvector.objects.ECEFvector

class ECEFvector(*pvector*, *frame=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.

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.

__init__(*self*, *pvector*, *frame=None*)
x.__init__(...) initializes x; see help(type(x)) for signature

Methods

__init__(self, pvector[, frame])	x.__init__(...) initializes x; see help(type(x)) for signature
change_frame(self, frame)	Converts to Cartesian position vector in another frame
delta_to(self, other)	Return cartesian delta vector from positions A to B decomposed in N.
to_ecef_vector(self)	
to_geo_point(self)	Converts ECEF-vector to geo-point.
to_nvector(self)	Converts ECEF-vector to n-vector.

Attributes

azimuth
azimuth_deg
elevation
elevation_deg
length

1.13.1.4.6 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')
```

The geodesic inverse problem

```
>>> positionA = wgs84.GeoPoint(-41.32, 174.81, degrees=True)
>>> positionB = wgs84.GeoPoint(40.96, -5.50, degrees=True)
>>> s12, az1, az2 = positionA.distance_and_azimuth(positionB, 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

```
>>> positionA = wgs84.GeoPoint(40.6, -73.8, degrees=True)
>>> az1, distance = 45, 10000e3
>>> positionB, az2 = positionA.displace(distance, az1, degrees=True)
>>> lat2, lon2 = positionB.latitude_deg, positionB.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__`(*self, latitude, longitude, z=0, frame=None, degrees=False*)
  x.__init__(...) initializes x; see help(type(x)) for signature

Methods

`__init__`(self, latitude, longitude[, z, ...])	x.__init__(...) initializes x; see help(type(x)) for signature
`delta_to`(self, other)	Return cartesian delta vector from positions A to B decomposed in N.
`displace`(self, distance, azimuth[, ...])	Return position B computed from current position, distance and azimuth.
`distance_and_azimuth`(self, point[, ...])	Return ellipsoidal distance between positions as well as the direction.
`to_ecef_vector`(self)	Converts latitude and longitude to ECEF-vector.
`to_geo_point`(self)	Return geo-point
`to_nvector`(self)	Converts latitude and longitude to n-vector.

Attributes

`latitude_deg`
`latlon`
`latlon_deg`
`longitude_deg`

1.13.1.4.7 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.

__init__(*self*, *normal*, *z=0*, *frame=None*)
x.__init__(...) initializes x; see help(type(x)) for signature

Methods

__init__(self, normal[, z, frame])	x.__init__(...) initializes x; see help(type(x)) for signature
delta_to(self, other)	Return cartesian delta vector from positions A to B decomposed in N.
mean(self)	Return mean position of the n-vectors.
mean_horizontal_position(*args, **kwds)	*mean_horizontal_position* is deprecated!
to_ecef_vector(self)	Converts n-vector to Cartesian position vector ("ECEF-vector")
to_geo_point(self)	Converts n-vector to geo-point.
to_nvector(self)	
unit(self)	Normalizes self to unit vector(s)

1.13.1.4.8 nvector.objects.GeoPath

class GeoPath(*positionA*, *positionB*)

Geographical path between two positions in Frame E

Parameters

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

__init__(*self*, *positionA*, *positionB*)
x.__init__(...) initializes x; see help(type(x)) for signature

Methods

`__init__`(self, positionA, positionB)	x.__init__(. . .) initializes x; see help(type(x)) for signature
`closest_point_on_great_circle`(self, point)	
`closest_point_on_path`(self, point)	Returns closest point on great circle path segment to the point.
`cross_track_distance`(self, point[, method, . . .])	Return cross track distance from path to point.
`ecef_vectors`(self)	Return positionA and positionB as ECEF-vectors
`geo_points`(self)	Return positionA and positionB as geo-points
`interpolate`(self, ti)	Return the interpolated point along the path
`intersect`(self, path)	Return the intersection(s) between the great circles of the two paths
`intersection`(*args, **kwds)	*intersection* is deprecated!
`nvector_normals`(self)	
`nvectors`(self)	Return positionA and positionB as n-vectors
`on_great_circle`(self, point[, rtol, atol])	
`on_path`(self, point[, method, rtol, atol])	Return True if point is on the path between A and B
`track_distance`(self[, method, radius])	Return the distance of the path.

1.13.1.4.9 nvector.objects.Pvector

class Pvector (*pvector, frame*)
Cartesian position vector in another frame

__init__ (*self, pvector, frame*)
x.__init__(. . .) initializes x; see help(type(x)) for signature

Methods

`__init__`(self, pvector, frame)	x.__init__(. . .) initializes x; see help(type(x)) for signature
`delta_to`(self, other)	Return cartesian delta vector from positions A to B decomposed in N.
`to_ecef_vector`(self)	
`to_geo_point`(self)	
`to_nvector`(self)	

Attributes

`azimuth`
`azimuth_deg`
`elevation`
`elevation_deg`
`length`

1.13.1.4.10 nvector.objects.diff_positions

diff_positions (**args, **kwds*)
diff_positions is deprecated!

Deprecated use delta_E instead.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

n

Index

Symbols

C

D

E

F

G

I

L

M

N

O

P

R