# GeoSpatialTools

## *Release 0.11.2*

## NOC Surface Processes

**Mar 05, 2025**

# CONTENTS:

# INTRODUCTION

Python library containing useful functions and classes for Spatial Analysis.

Tested on Python versions 3.9 to 3.13.

# GETTING STARTED

## 2.1 Installation

### 2.1.1 Via Pip

GeoSpatialTools is not available on PyPI, however it can be installed via pip with the following command:

```
pip install git+ssh://git@git.noc.ac.uk/nocsurfaceprocesses/geospatialtools.git
```

### 2.1.2 From Source

Alternatively, you can clone the repository and install using pip (or conda if preferred).

```
git clone git@git.noc.ac.uk/nocsurfaceprocesses/geospatialtools.git
cd geospatialtools
python -m venv venv
source venv/bin/activate
pip install -e .
```

# CREDITS

## 3.1 Development Lead

- Joseph T. Siddons <josidd@noc.ac.uk> @josidd

## 3.2 Contributoring Developers

- Richard C. Cornes <rcornes@noc.ac.uk> @ricorne

# USERS GUIDE

## 4.1 Neighbours

Functions for finding nearest neighbours using bisection.

**exception** `GeoSpatialTools.neighbours.`**`SortedError`**

Error class for Sortedness

**exception** `GeoSpatialTools.neighbours.`**`SortedWarning`**

Warning class for Sortedness

`GeoSpatialTools.neighbours.`**`find_nearest`**(*vals*, *test*, *check_sorted=True*)

Find the nearest value in a list of values for each test value.

Uses bisection for speediness!

> **Parameters**
>
> - **vals** (`list[Numeric]`) – List of values - this is the pool of values for which we are looking for a nearest match. This list MUST be sorted. Sortedness is not checked, nor is the list sorted.
>
> - **test** (`list[Numeric] | Numeric`) – List of query values
>
> - **check_sorted** (`bool`) – Optionally check that the input vals is sorted. Raises an error if set to True (default), displays a warning if set to False.
>
> **Return type**
> Union[List[int], int]
>
> **Returns**
>
> - *A list containing the index of the nearest neighbour in vals for each value*
>
> - *in test. Or the index of the nearest neighbour if test is a single value.*

## 4.2 QuadTree

Constructors for QuadTree classes that can decrease the number of comparisons for detecting nearby records for example. This is an implementation that uses Haversine distances for comparisons between records for identification of neighbours.

**class** `GeoSpatialTools.quadtree.`**`Ellipse`**(*lon*, *lat*, *a*, *b*, *theta*)

A simple Ellipse Class for an ellipse on the surface of a sphere.

> **Parameters**

- **lon** (*float*) – Horizontal centre of the ellipse

- **lat** (*float*) – Vertical centre of the ellipse

- **a** (*float*) – Length of the semi-major axis

- **b** (*float*) – Length of the semi-minor axis

- **theta** (*float*) – Angle of the semi-major axis from horizontal anti-clockwise in radians

**contains**(*point*)

Test if a point is contained within the Ellipse

> **Return type**
>> bool

**nearby_rect**(*rect*)

Test if a rectangle is near to the Ellipse

> **Return type**
>> bool

**class** GeoSpatialTools.quadtree.**QuadTree**(*boundary*, *capacity=5*, *depth=0*, *max_depth=None*)

A Simple QuadTree class for PyCOADS

> **Parameters**
>
> - **boundary** ([*Rectangle*](#)) – The bounding Rectangle of the QuadTree
>
> - **capacity** (*int*) – The capacity of each cell, if max_depth is set then a cell at the maximum depth may contain more points than the capacity.
>
> - **depth** (*int*) – The current depth of the cell. Initialises to zero if unset.
>
> - **max_depth** (*int | None*) – The maximum depth of the QuadTree. If set, this can override the capacity for cells at the maximum depth.

**divide**()

Divide the QuadTree

**insert**(*point*)

Insert a point into the QuadTree

> **Return type**
>> bool

**len**(*_current_len=0*)

Get the number of points in the OctTree

> **Return type**
>> int

**nearby_points**(*point*, *dist*, *points=None*)

Get all points that are nearby another point

> **Return type**
>> List[[*Record*](#)]

**query**(*rect*, *points=None*)

Get points that fall in a rectangle

> **Return type**
>> List[[*Record*](#)]

**query_ellipse**(*ellipse*, *points=None*)

    Get points that fall in an ellipse.

        **Return type**
            List[*Record*]

**remove**(*point*)

    Remove a Record from the QuadTree if it is in the QuadTree.

    Returns True if the Record is removed.

        **Return type**
            bool

**class** GeoSpatialTools.quadtree.**Record**(*lon*, *lat*, *datetime=None*, *uid=None*, *fix_lon=True*, *\*\*data*)

    Record class

    This is a simple instance of a record, it requires position data. It can optionally include datetime, a UID, and extra data passed as keyword arguments.

    Equality is checked only on the required fields + UID if it is specified.

        **Parameters**

            - **lon** (*float*) – Horizontal coordinate

            - **lat** (*float*) – Vertical coordinate

            - **datetime** (*datetime | None*) – Datetime of the record

            - **uid** (*str | None*) – Unique Identifier

            - **fix_lon** (*bool*) – Force longitude to -180, 180

            - **\*\*data** – Additional data passed to the Record for use by other functions or classes.

    **distance**(*other*)

        Compute the Haversine distance to another Record

            **Return type**
                float

**class** GeoSpatialTools.quadtree.**Rectangle**(*west*, *east*, *south*, *north*)

    A simple Rectangle class

        **Parameters**

            - **west** (*float*) – Western boundary of the Rectangle

            - **east** (*float*) – Eastern boundary of the Rectangle

            - **south** (*float*) – Southern boundary of the Rectangle

            - **north** (*float*) – Northern boundary of the Rectangle

    **contains**(*point*)

        Test if a point is contained within the Rectangle

            **Return type**
                bool

    **property edge_dist: float**

        Approximate maximum distance from the centre to an edge

**intersects**(*other*)

>   Test if another Rectangle object intersects this Rectangle

>>      **Return type**
>>          bool

**property lat: float**

>   Centre latitude of the Rectangle

**property lat_range: float**

>   Latitude range of the Rectangle

**property lon: float**

>   Centre longitude of the Rectangle

**property lon_range: float**

>   Longitude range of the Rectangle

**nearby**(*point*, *dist*)

>   Check if point is nearby the Rectangle

>>      **Return type**
>>          bool

## 4.3 OctTree

Constructors for OctTree classes that can decrease the number of comparisons for detecting nearby records for example. This is an implementation that uses Haversine distances for comparisons between records for identification of neighbours.

**class** GeoSpatialTools.octtree.**OctTree**(*boundary*, *capacity=5*, *depth=0*, *max_depth=None*)

>   A Simple OctTree class for PyCOADS.

>   Acts as a space-time OctTree on the surface of Earth, allowing for querying nearby points faster than searching a full DataFrame. As SpaceTimeRecords are added to the OctTree, the OctTree divides into 8 children as the capacity is reached. Additional SpaceTimeRecords are then added to the children where they fall within the child OctTree's boundary.

>   SpaceTimeRecords already part of the OctTree before divided are not distributed to the children OctTrees.

>   Whilst the OctTree has a temporal component, and was designed to utilise datetime / timedelta objects, numeric values and ranges can be used. This usage must be consistent for the boundary and all SpaceTimeRecords that are part of the OctTree. This allows for usage of pentad, timestamp, Julian day, etc. as datetime values.

>       **Parameters**

>> - **boundary** (*SpaceTimeRectangle*) – The bounding SpaceTimeRectangle of the QuadTree
>> - **capacity** (*int*) – The capacity of each cell, if max_depth is set then a cell at the maximum depth may contain more points than the capacity.
>> - **depth** (*int*) – The current depth of the cell. Initialises to zero if unset.
>> - **max_depth** (*int | None*) – The maximum depth of the QuadTree. If set, this can override the capacity for cells at the maximum depth.

**divide**()

>   Divide the QuadTree

**insert**(*point*)

Insert a SpaceTimeRecord into the QuadTree.

Note that the SpaceTimeRecord can have numeric datetime values if that is consistent with the OctTree.

> **Return type**
> > bool

**len**(*_current_len=0*)

Get the number of points in the OctTree

> **Return type**
> > int

**nearby_points**(*point*, *dist*, *t_dist*, *points=None*)

Get all points that are nearby another point.

Query the OctTree to find all SpaceTimeRecords within the OctTree that are nearby to the query Space-TimeRecord. This search should be faster than searching through all records, since only OctTree children whose boundaries are close to the query SpaceTimeRecord are evaluated.

> **Parameters**
>
> - **point** (`SpaceTimeRecord`) – The query point.
> - **dist** (`float`) – The distance for comparison. Note that Haversine distance is used as the distance metric as the query SpaceTimeRecord and OctTree are assumed to lie on the surface of Earth.
> - **t_dist** (`datetime.timedelta`) – Max time gap between SpaceTimeRecords within the OctTree and the query SpaceTimeRecord. Can be numeric if the OctTree boundaries, SpaceTimeRecords, and query SpaceTimeRecord have numeric datetime values and ranges.
> - **points** (`SpaceTimeRecords` | `None`) – List of SpaceTimeRecords already found. Most use cases will be to not set this value, since it's main use is for passing onto the children OctTrees.
>
> **Return type**
> > *SpaceTimeRecords*
>
> **Returns**
>
> - **SpaceTimeRecords** (*A list of SpaceTimeRecords whose distance to the*)
> - *query SpaceTimeRecord is <= dist, and the datetimes of the*
> - *SpaceTimeRecords fall within the datetime range of the query*
> - *SpaceTimeRecord.*

**query**(*rect*, *points=None*)

Get points that fall in a SpaceTimeRectangle

> **Return type**
> > *SpaceTimeRecords*

**query_ellipse**(*ellipse*, *points=None*)

Get points that fall in an ellipse.

> **Return type**
> > *SpaceTimeRecords*

**remove**(*point*)

Remove a SpaceTimeRecord from the OctTree if it is in the OctTree.

Returns True if the SpaceTimeRecord is removed.

> **Return type**
> bool

**class** GeoSpatialTools.octtree.**SpaceTimeEllipse**(*lon*, *lat*, *a*, *b*, *theta*, *start*, *end*)

A simple Ellipse Class for an ellipse on the surface of a sphere.

> **Parameters**
>
> - **lon** (*float*) – Horizontal centre of the ellipse
> - **lat** (*float*) – Vertical centre of the ellipse
> - **a** (*float*) – Length of the semi-major axis
> - **b** (*float*) – Length of the semi-minor axis
> - **theta** (*float*) – Angle of the semi-major axis from horizontal anti-clockwise in radians
> - **start** (*datetime.datetime*) – Start date of the Ellipse
> - **end** (*datetime.datetime*) – Send date of the Ellipse

**contains**(*point*)

Test if a point is contained within the Ellipse

> **Return type**
> bool

**nearby_rect**(*rect*)

Test if a rectangle is near to the Ellipse

> **Return type**
> bool

**class** GeoSpatialTools.octtree.**SpaceTimeRecord**(*lon*, *lat*, *datetime*, *uid=None*, *fix_lon=True*, *\*\*data*)

SpaceTimeRecord class.

This is a simple instance of a record, it requires position and temporal data. It can optionally include a UID and extra data.

The temporal component was designed to use *datetime* values, however all methods will work with numeric datetime information - for example a pentad, time-stamp, Julian day, etc. Note that any uses within an OctTree and SpaceTimeRectangle must also have timedelta values replaced with numeric ranges in this case.

Equality is checked only on the required fields + UID if it is specified.

> **Parameters**
>
> - **lon** (*float*) – Horizontal coordinate (longitude).
> - **lat** (*float*) – Vertical coordinate (latitude).
> - **datetime** (*datetime.datetime*) – Datetime of the record. Can also be a numeric value such as pentad. Comparisons between Records with datetime and Records with numeric datetime will fail.
> - **uid** (*str | None*) – Unique Identifier.
> - **fix_lon** (*bool*) – Force longitude to -180, 180

- **\*\*data** – Additional data passed to the SpaceTimeRecord for use by other functions or classes.

**distance**(*other*)

    Compute the Haversine distance to another SpaceTimeRecord. Only computes spatial distance.

> **Return type**
>     float

**class** GeoSpatialTools.octtree.**SpaceTimeRecords**(*iterable=(), / (Positional-only parameter separator (PEP 570))*)

    List of SpaceTimeRecords

**class** GeoSpatialTools.octtree.**SpaceTimeRectangle**(*west*, *east*, *south*, *north*, *start*, *end*)

    A simple Space Time SpaceTimeRectangle class.

    This constructs a simple Rectangle object. The defining coordinates are the centres of the box, and the extents are the full width, height, and time extent.

    Whilst the rectangle is assumed to lie on the surface of Earth, this is a projection as the rectangle is defined by a longitude/latitude range.

    The temporal components are defined in the same way as the spatial components, that is that the *datetime* component (t) is the "centre", and the time extent (dt) is the full time range of the box.

> **Parameters**
>
> - **west** (`float`) – Western boundary of the Rectangle
> - **east** (`float`) – Eastern boundary of the Rectangle
> - **south** (`float`) – Southern boundary of the Rectangle
> - **north** (`float`) – Northern boundary of the Rectangle
> - **start** (`datetime.datetime`) – Start datetime of the Rectangle
> - **end** (`datetime.datetime`) – End datetime of the Rectangle

**property centre_datetime:  datetime**

    The midpoint time of the Rectangle

**contains**(*point*)

    Test if a point is contained within the SpaceTimeRectangle

> **Return type**
>     bool

**property edge_dist:  float**

    Approximate maximum distance from the centre to an edge

**intersects**(*other*)

    Test if another Rectangle object intersects this Rectangle

> **Return type**
>     bool

**property lat:  float**

    Centre latitude of the Rectangle

**property lat_range:  float**

    Latitude range of the Rectangle

**property lon: float**

Centre longitude of the Rectangle

**property lon_range: float**

Longitude range of the Rectangle

**nearby**(*point*, *dist*, *t_dist*)

Check if point is nearby the Rectangle

Determines if a SpaceTimeRecord that falls on the surface of Earth is nearby to the rectangle in space and time. This calculation uses the Haversine distance metric.

Distance from rectangle to point is challenging on the surface of a sphere, this calculation will return false positives as a check based on the distance from the centre of the rectangle to the corners, or to its Eastern edge (if the rectangle crosses the equator) is used in combination with the input distance.

The primary use-case of this method is for querying an OctTree for nearby Records.

> **Parameters**
>
> > - **point** (SpaceTimeRecord)
> >
> > - **dist** (*float,*)
> >
> > - **t_dist** (*datetime.timedelta*)
>
> **Returns**
> > bool
>
> **Return type**
> > True if the point is <= dist + max(dist(centre, corners))

**property time_range: timedelta**

The time extent of the Rectangle

## 4.4 KDTree

An implementation of KDTree using Haversine Distance for GeoSpatial analysis. Useful tool for quickly searching for nearest neighbours. The implementation is a K=2 or 2DTree as only 2 dimensions (longitude and latitude) are used.

Haversine distances are used for comparisons, so that the spherical geometry of the earth is accounted for.

**class** GeoSpatialTools.kdtree.**KDTree**(*points*, *depth=0*, *max_depth=20*)

A Haverine distance implementation of a balanced KDTree.

This implementation is a _balanced_ KDTree, each leaf node should have the same number of points (or differ by 1 depending on the number of points the KDTree is initialised with).

The KDTree partitions in each of the lon and lat dimensions alternatively in sequence by splitting at the median of the dimension of the points assigned to the branch.

> **Parameters**
>
> > - **points** (*list[Record]*) – A list of GeoSpatialTools.Record instances.
> >
> > - **depth** (*int*) – The current depth of the KDTree, you should set this to 0, it is used internally.
> >
> > - **max_depth** (*int*) – The maximum depth of the KDTree. The leaf nodes will have depth no larger than this value. Leaf nodes will not be created if there is only 1 point in the branch.

**delete**(*point*)

>   Delete a Record from the KDTree. May unbalance the KDTree

>   >   **Return type**
>   >   >   bool

**insert**(*point*)

>   Insert a Record into the KDTree. May unbalance the KDTree.

>   The point will not be inserted if it is already in the KDTree.

>   >   **Return type**
>   >   >   bool

**query**(*point*)

>   Find the nearest Record within the KDTree to a query Record

>   >   **Return type**
>   >   >   Tuple[List[*Record*], float]

## 4.5 GreatCircle

Constructors and methods for interacting with GreatCircle objects, including comparisons between GreatCircle objects.

**class** GeoSpatialTools.great_circle.**GreatCircle**(*lon0*, *lat0*, *lon1*, *lat1*, *R=6371*)

>   A GreatCircle object for a pair of positions.

>   Construct a great circle path between a pair of positions.

>   https://www.boeing-727.com/Data/fly%20odds/distance.html

>   >   **Parameters**
>   >   - **lon0** (*float*) – Longitude of start position.
>   >   - **lat0** (*float*) – Latitude of start position.
>   >   - **lon1** (*float*) – Longitude of end position.
>   >   - **lat1** (*float*) – Latitude of end position.
>   >   - **R** (*float*) – Radius of the sphere. Default is Earth radius in km (6371.0).

**dist_from_point**(*lon*, *lat*)

>   Compute distance from the GreatCircle to a point on the sphere.

>   >   **Parameters**
>   >   - **lon** (*float*) – Longitude of the position to test.
>   >   - **lat** (*float*) – Longitude of the position to test.

>   >   **Returns**
>   >   >   Minimum distance between point and the GreatCircle arc.

>   >   **Return type**
>   >   >   float

**intersection**(*other*, *epsilon=0.01*)

Determine intersection position with another GreatCircle.

Determine the location at which the GreatCircle intersects another GreatCircle arc. (To within some epsilon threshold).

Returns *None* if there is no solution - either because there is no intersection point, or the planes generated from the arc and centre of the sphere are identical.

> **Parameters**
>
> - **other** ([GreatCircle](#)) – Intersecting GreatCircle object
> - **epsilon** (*float*) – Threshold for intersection
>
> **Returns**
>    Position of intersection
>
> **Return type**
>    (float, float) | None

**intersection_angle**(*other*, *epsilon=0.01*)

Get angle of intersection with another GreatCircle.

Get the angle of intersection with another GreatCircle arc. Returns None if there is no intersection.

The intersection angle is computed using the normals of the planes formed by the two intersecting great circle objects.

> **Parameters**
>
> - **other** ([GreatCircle](#)) – Intersecting GreatCircle object
> - **epsilon** (*float*) – Threshold for intersection
>
> **Returns**
>    Intersection angle in degrees
>
> **Return type**
>    float | None

GeoSpatialTools.great_circle.**cartesian_to_lonlat**(*x*, *y*, *z*, *to_radians=False*)

Get lon, and lat from cartesian coordinates.

> **Parameters**
>
> - **x** (*float*) – x coordinate
> - **y** (*float*) – y coordinate
> - **z** (*float*) – z coordinate
> - **to_radians** (*bool*) – Return angles in radians. Otherwise return values in degrees.
>
> **Return type**
>    tuple[float, float]
>
> **Returns**
>
> - *(float, float)*
> - *lon, lat*

GeoSpatialTools.great_circle.**polar_to_cartesian**(*lon*, *lat*, *R=6371*, *to_radians=True*, *normalised=True*)

Convert from polars coordinates to cartesian.

Get cartesian coordinates from spherical polar coordinates. Default behaviour assumes lon and lat, so converts to radians. Set *to_radians=False* if the coordinates are already in radians.

> **Parameters**
>
> - **lon** (*float*) – Longitude.
>
> - **lat** (*float*) – Latitude.
>
> - **R** (*float*) – Radius of sphere.
>
> - **to_radians** (*bool*) – Convert lon and lat to radians.
>
> - **normalised** (*bool*) – Return normalised vector (ignore R value).
>
> **Returns**
>
> x, y, z cartesian coordinates.
>
> **Return type**
>
> (float, float, float)

## 4.6 Distance Metrics

Functions for computing navigational information. Can be used to add navigational information to DataFrames.

GeoSpatialTools.distance_metrics.**bearing**(*lon0*, *lat0*, *lon1*, *lat1*)

Compute the bearing of a track from (lon0, lat0) to (lon1, lat1).

Duplicated from geo-py

> **Parameters**
>
> - **lon0** (*float,*) – Longitude of start point
>
> - **lat0** (*float,*) – Latitude of start point
>
> - **lon1** (*float,*) – Longitude of target point
>
> - **lat1** (*float,*) – Latitude of target point
>
> **Returns**
>
> **bearing** – The bearing from point (lon0, lat0) to point (lon1, lat1) in degrees.
>
> **Return type**
>
> float

GeoSpatialTools.distance_metrics.**destination**(*lon*, *lat*, *bearing*, *distance*)

Compute destination of a great circle path.

Compute the destination of a track started from 'lon', 'lat', with 'bearing'. Distance is in units of km.

Duplicated from geo-py

> **Parameters**
>
> - **lon** (*float*) – Longitude of initial position
>
> - **lat** (*float*) – Latitude of initial position
>
> - **bearing** (*float*) – Direction of track

- **distance** (*float*) – Distance to travel

> **Returns**
> > **destination** – Longitude and Latitude of final position
>
> **Return type**
> > tuple[float, float]

GeoSpatialTools.distance_metrics.**gcd_slc**(*lon0*, *lat0*, *lon1*, *lat1*)

> Compute great circle distance on earth surface between two locations.
>
> > **Parameters**
> >
> > - **lon0** (*float*) – Longitude of position 0
> >
> > - **lat0** (*float*) – Latitude of position 0
> >
> > - **lon1** (*float*) – Longitude of position 1
> >
> > - **lat1** (*float*) – Latitude of position 1
> >
> > **Returns**
> > > **dist** – Great circle distance between position 0 and position 1.
> >
> > **Return type**
> > > float

GeoSpatialTools.distance_metrics.**haversine**(*lon0*, *lat0*, *lon1*, *lat1*)

> Compute Haversine distance between two points.
>
> > **Parameters**
> >
> > - **lon0** (*float*) – Longitude of position 0
> >
> > - **lat0** (*float*) – Latitude of position 0
> >
> > - **lon1** (*float*) – Longitude of position 1
> >
> > - **lat1** (*float*) – Latitude of position 1
> >
> > **Returns**
> > > **dist** – Haversine distance between position 0 and position 1.
> >
> > **Return type**
> > > float

GeoSpatialTools.distance_metrics.**midpoint**(*lon0*, *lat0*, *lon1*, *lat1*)

> Compute the midpoint of a great circle track
>
> > **Parameters**
> >
> > - **lon0** (*float*) – Longitude of position 0
> >
> > - **lat0** (*float*) – Latitude of position 0
> >
> > - **lon1** (*float*) – Longitude of position 1
> >
> > - **lat1** (*float*) – Latitude of position 1
> >
> > **Returns**
> > > Positions of midpoint between position 0 and position 1
> >
> > **Return type**
> > > lon, lat

# PYTHON MODULE INDEX

## g