GeoTrees

Release 0.11.2

NOC Surface Processes

May 13, 2025

CONTENTS:

CHAPTER

ONE

INTRODUCTION

1.1 geotrees

`geotrees` is a python3 library developed at NOC (National Oceanography Centre, Southampton, UK) for identifying neighbours in a geo-spatial context. This is designed to solve problems where one needs to identify data within a spatial range on the surface of the Earth. The library provides implementations of standard tools for neighbourhood searching, such as k-d-tree and Quadtree that have been adapted to account for spherical geometry, using a haversine distance metric.

The tool allows for spatial look-ups with $O(\log(n))$ complexity in time. Additionally, a simple 1-d nearest neighbours look-up is provided for sorted data using bisection search.

`geotrees` also provides functionality for working with great-circle objects, for example intersecting great-circles.

1.2 Credits

1.2.1 Development Lead

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

1.2.2 Contributoring Developers

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

CHAPTER

TWO

INSTALLATION

geotrees is not currently available on PyPI so must be installed either from source or directly from the GitLab repository. Versions of python between 3.9 and 3.13, inclusive, are supported, however the recommended version of python is 3.12.

We recommend the installation of geotrees using the uv package manager, however it can be installed using pip.

The only required dependency of the project is NumPy. Additional dependency polars is required to run the Jupyter notebooks.

2.1 Via UV

You can install the library directly from the GitLab repository, adding the library to your current uv virtual environment. This will add the library as a dependency in your current project.

```
uv add git+ssh://git@git.noc.ac.uk/nocsurfaceprocesses/geotrees.git
```

2.1.1 Development mode

If you wish to contribute to geotrees you can install the library in development mode. This will require cloning the repository and creating a new uv environment.

```
# Get the code
git clone git@git.noc.ac.uk/nocsurfaceprocesses/geotrees.git
cd geotrees

# Install with all dependencies and create an environment with python 3.12
uv sync --all-extras --dev --python 3.12

# Load the environment
source .venv/bin/activate

# Run the unit tests
uv run pytest test
```

Note

The recommended python version is python 3.12. By default, uv creates a virtual environment in .venv.

2.2 Via Pip

The library can be installed via pip with the following command:

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

2.2.1 From Source

Alternatively, you can clone the repository and install using pip (or conda if preferred). This installs in `editable` mode.

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

CHAPTER

THREE

BISECTION SEARCH

Bisection can be used to find the nearest neighbour in a sorted one-dimensional list of search values in $O(\log(n))$ time complexity.

The implementation in *geotrees* makes use of the *bisect* library, which is part of the Python standard library. The input types are numeric types, which can include `int`, `float`, or `datetime.datetime` values.

The bisection approach repeatedly splits the list of search values in two at the mid-index. The query value is compared to the search value at the mid-index. If the query value is larger than the search value at the mid-index, then the search values after the mid-index become the new search values. If the query value is smaller than the search value at the mid-index then the search values before the mid-index become the new search values. This bisecting is repeated (succesively halving the number of search values) until one values remain. The nearest neighbour is either the value at the remaining index, or the value at the index one above.

Note

The above assumes that the list of search values is sorted in increasing order. The opposite applies if the list is sorted in reverse.

Warning

The input values must be sorted

3.1 Example

```
from geotrees import find_nearest
import numpy as np

search_values: list[float] = list(np.random.randn(50))
search_values.sort()

query_value: float = 0.45
neighbour_index: int = find_nearest(
     vals=search_values,
     test=query_value,
)
neighbour_value: float = search_values[neighbour_index]
```

3.2 neighbours Module

Functions for finding nearest neighbours using bisection. Nearest neighbours can be found with $O(\log(n))$ time-complexity.

Data for these functions must be sorted, otherwise incorrect values may be returned.

exception `geotrees.neighbours.`**`SortedError`**

Error class for Sortedness

exception `geotrees.neighbours.`**`SortedWarning`**

Warning class for Sortedness

`geotrees.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!

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.

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** (*Numeric | list[Numeric]*) – Query value(s)
- **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.

Returns

Index, or list of indices, of nearest value, or values.

Return type

int | list[int]

CHAPTER

FOUR

RECORD CLASSES

Record classes in geotrees form the back-bone of the data structures within the library. They represent a consistent input data-type across all classes in the library.

There are two classes of Record:

- geotrees.record.Record for two-dimensional data structures defined by lon (longitude) and lat (latitude). Optionally, one can pass datetime and uid, as well as additional data attributes with keyword arguments.
- geotrees.record.SpaceTimeRecord for three-dimensional data structures defined by lon (longitude), lat (latitude), and datetime. Optionally, one can pass uid, as well as additional data attributes with keyword arguments.

Only the positional, datetime, and uid attributes are used for equality tests. Record objects are used for QuadTree and KDTree objects, whereas SpaceTimeRecord objects must be used for OctTree.

Note

Record and SpaceTimeRecord are exposed at the geotrees level.

4.1 Example

```
from geotrees import Record

record: Record = Record(lon=-151.2, lat=42.7, uid="foo")
dist: float = record.distance(Record(-71.1, -23.2, uid="bar"))
```

4.2 record Module

Record objects used for containing data passed to QuadTree, OctTree and KDTree classes. Require positions defined by "lon" and "lat", SpaceTimeRecord objects also require "datetime". Optional fields are "uid", other data can be passed as keyword arguments. Only positional, temporal, and uid values are used for equality checks.

Distances between records is calculated using Haversine distance.

Classes prefixed by "SpaceTime" include a temporal dimension and should be used with OctTree classes.

class geotrees.record.**Record**(*lon*, *lat*, *datetime=None*, *uid=None*, *fix_lon=True*, ***data*)

Record class

This is a simple instance of an 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 geotrees.record.SpaceTimeRecord`(*lon*, *lat*, *datetime*, *uid=None*, *fix_lon=True*, ***data*)

ICOADS Record class.

This is a simple instance of an ICOARDS 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, timestamp, 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`

CHAPTER

FIVE

SHAPE CLASSES

The geotrees.shape module defines various classes that can be used to define the boundary for QuadTree and OctTree classes, or query regions for the same.

Rectangle and SpaceTimeRectangle classes are used to define the boundaries for QuadTree and OctTree classes respectively. They are defined by the bounding box in space (and time for a SpaceTimeRectangle).

Ellipse and SpaceTimeEllipse classes are defined by lon and lat indicating the centre of the ellipse, a and b indicating the length of the semi-major and semi-minor axes respectively, and theta indicating the angle of the ellipse. SpaceTimeEllipse classes also require start and end datetime values. The SpaceTimeEllipse is an elliptical cylinder where the height is represented by the time dimension.

Rectangle and Ellipse classes can be used to define a query shape for a QuadTree, using QuadTree.query and QuadTree.query_ellipse respectively.

SpaecTimeRectangle and SpaceTimeEllipse classes can be used to define a query shape for a OctTree, using OctTree.query and OctTree.query_ellipse respectively.

5.1 Example

```
from geotrees.shape import Rectangle, SpaceTimeEllipse
from datetime import datetime
from math import pi

rectangle: Rectangle = Rectangle(
    west=-180,
    east=180,
    south=-90,
    north=90,
)

ellipse: SpaceTimeEllipse = SpaceTimeEllipse(
    lon=23.4,
    lat=-17.9,
    a=103,
    b=71,
    theta=pi/3,
    start=datetime(2009, 2, 13, 19, 30),
    end=datetime(2010, 7, 2, 3, 45),
)
```

5.2 shape Module

Shape objects for defining QuadTree or OctTree classes, or for defining a query region for QuadTree and OctTree classes.

Distances between shapes, or between shapes and Records uses the Haversine distance.

All shape objects account for spherical geometry and the wrapping of longitude at -180, 180 degrees.

Classes prefixed by "SpaceTime" include a temporal dimension and should be used with OctTree classes.

class `geotrees.shape.`**`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 Record is contained within the Ellipse

Return type

`bool`

nearby_rect(*rect*)

Test if a Rectangle is near to the Ellipse

Return type

`bool`

class `geotrees.shape.`**`Rectangle`**(*west*, *east*, *south*, *north*)

A simple Rectangle class for GeoSpatial analysis. Defined by a bounding 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

contains(*point*)

Test if a Record 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 Record is nearby the Rectangle

Return type

bool

class geotrees.shape.**SpaceTimeEllipse**(*lon*, *lat*, *a*, *b*, *theta*, *start*, *end*)

A simple SpaceTimeEllipse Class for an ellipse on the surface of a sphere with an additional time dimension.

The representation of the shape is an elliptical cylinder, with the time dimension representing the height of the cylinder.

Parameters

- **lon** (*float*) – Horizontal centre of the SpaceTimeEllipse
- **lat** (*float*) – Vertical centre of the SpaceTimeEllipse
- **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 SpaceTimeEllipse
- **end** (*datetime.datetime*) – Send date of the SpaceTimeEllipse

contains(*point*)

Test if a SpaceTimeRecord is contained within the SpaceTimeEllipse

Return type

bool

nearby_rect(*rect*)

Test if a SpaceTimeRectangle is near to the SpaceTimeEllipse

Return type

bool

class geotrees.shape.**SpaceTimeRectangle**(*west*, *east*, *south*, *north*, *start*, *end*)

A simple SpaceTimeRectangle class for GeoSpatial analysis. Defined by a bounding box in space and time.

Parameters

- **west** (*float*) – Western boundary of the SpaceTimeRectangle
- **east** (*float*) – Eastern boundary of the SpaceTimeRectangle
- **south** (*float*) – Southern boundary of the SpaceTimeRectangle
- **north** (*float*) – Northern boundary of the SpaceTimeRectangle

- **start** (*datetime.datetime*) – Start datetime of the SpaceTimeRectangle
- **end** (*datetime.datetime*) – End datetime of the SpaceTimeRectangle

property centre_datetime: datetime

The midpoint time of the SpaceTimeRectangle

contains(*point*)

Test if a SpaceTimeRecord 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 SpaceTimeRectangle object intersects this SpaceTimeRectangle.

Return type
`bool`

property lat: float

Centre latitude of the SpaceTimeRectangle

property lat_range: float

Latitude range of the SpaceTimeRectangle

property lon: float

Centre longitude of the SpaceTimeRectangle

property lon_range: float

Longitude range of the SpaceTimeRectangle

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

Check if SpaceTimeRecord is nearby the SpaceTimeRectangle

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 SpaceTimeRecords.

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

CHAPTER

SIX

QUADTREE

A Quadtree is a data-structure where each internal node has exactly four branches, and are used to recursively partition a two-dimensional spatial domain. Each branch note is itself a Quadtree, whose spatial domain represents one of the quadrants (north-west, north-east, south-west, south-east) of its parent's domain. The partitioning of data in this way is dependent on the spatial density of data inserted into the Quadtree. The Quadtree is typically initialised with a capacity value, once the capacity is reached (by inserting data points), the Quadtree divides and subsequent data points are added to the appropriate branch-node.

Quadtree structures allow for fast identification of data within some query region. The structure of the tree ensures that only nodes whose domain boundary intersects (or contains or is contained by) the query region are evaluated. The time-complexity of these query operations is $O(\log(n))$, the space-complexity of a Quadtree is $O(n)$.

Typically, it is assumed that the data uses a cartesian coordinate system, so comparisons between boundaries and query shapes utilise cartesian geometry and euclidean distances. The implementation of Quadtree within this library, the `QuadTree` class, utilises the Haversine distance as a metric for identifying records within the queried region. This allows the Quadtree to account for the spherical geometry of the Earth. Boundary checks with query regions also account for the wrapping of longitude at -180, 180 degrees.

The `QuadTree` object is defined by a bounding box, i.e. boundaries at the western, eastern, southern, and northern edges of the data that will be inserted into the `QuadTree`. Additionally, a capacity and maximum depth can be provided. If the capacity is exceeded whilst inserting records the `QuadTree` will divide and new records will be inserted into the appropriate branch `QuadTree`. The maximum depth is the maximum height of the `QuadTree`, if capacity is also specified then this will be overridden if the `QuadTree` is at this depth, and the `QuadTree` will not divide.

6.1 Documentation

6.1.1 Inserting Records

A `Record` can be added to an `QuadTree` with `QuadTree.insert` which will return `True` if the operation was successful, `False` otherwise. The `QuadTree` is modified in place.

6.1.2 Removing Records

A `Record` can be removed from an `QuadTree` with `QuadTree.remove` which will return `True` if the operation was successful, `False` otherwise. The `QuadTree` is modified in place.

6.1.3 Querying

The `QuadTree` class defined in `geotrees.quadtree` can be queried in the following ways:

- with a `Record`, a spatial range with `QuadTree.nearby_points`. All points within the spatial range of the `Record` will be returned in a list. The `Record` can be excluded from the results if the `exclude_self` argument is set.

- with a `Rectangle` using `QuadTree.query`. All points within the specified `Rectangle` will be returned in a list.
- with a `Ellipse` using `QuadTree.query_ellipse`. All points within the specified `Ellipse` will be returned in a list.

6.2 Example

```
from geotrees import QuadTree, Record, Rectangle
from random import choice

lon_range = list(range(-180, 180))
lat_range = list(range(-90, 90))

N_samples = 1000

# Construct Tree
boundary = Rectangle(
    west=-180,
    east=180,
    south=-90,
    north=90,
)  # Full domain
quadtree = QuadTree(boundary)

# Populate the tree
records: list[Record] = [
    Record(
        choice(lon_range),
        choice(lat_range),
    ) for _ in range(N_samples)
]
for record in records:
    quadtree.insert(record)

dist: float = 340  # km

# Find all Records that are 340km away from test_value
neighbours: list[Record] = quadtree.nearby_points(test_value, dist)
```

6.3 quadtree Module

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 `geotrees.quadtree.`**`QuadTree`**(*boundary*, *capacity=5*, *depth=0*, *max_depth=None*)

Acts as a Geo-spatial QuadTree on the surface of Earth, allowing for querying nearby points faster than searching a full DataFrame. As Records are added to the QuadTree, the QuadTree divides into 4 branches as the capacity is reached, points contained within the QuadTree are not distributed to the branch QuadTrees. Additional Records are then added to the branch where they fall within the branch QuadTree's boundary.

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.

Parameters
point (Record) – The point to insert

Returns
True if the point was inserted into the QuadTree

Return type
bool

len(*current_len=0*)

Get the number of points in the QuadTree

Return type
int

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

Get all Records contained in the QuadTree that are nearby another query Record.

Query the QuadTree to find all Records within the QuadTree that are nearby to the query Record. This search should be faster than searching through all records, since only QuadTree branch whose boundaries are close to the query Record are evaluated.

Parameters

- **point** (Record) – The query point.
- **dist** (*float*) – The distance for comparison. Note that Haversine distance is used as the distance metric as the query Record and QuadTree are assumed to lie on the surface of Earth.
- **points** (*Records | None*) – List of Records already found. Most use cases will be to not set this value, since it's main use is for passing onto the branch QuadTrees.
- **exclude_self** (*bool*) – Optionally exclude the query point from the results if the query point is in the OctTree

Returns
A list of Records whose distance to the query Record is <= dist, and the datetimes of the Records fall within the datetime range of the query Record.

Return type
list[*Record*]

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

Get Records contained within the QuadTree that fall in a Rectangle

Parameters
: `rect` (`Rectangle`)

Returns
: The Record values contained within the QuadTree that fall within the bounds of rect.

Return type
: list[*Record*]

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

Get Records contained within the QuadTree that fall in a Ellipse

Parameters
: `ellipse` (`Ellipse`)

Returns
: The Record values contained within the QuadTree that fall within the bounds of ellipse.

Return type
: list[*Record*]

`remove`(*point*)

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

Parameters
: `point` (`Record`) – The point to remove

Returns
: True if the point is removed

Return type
: bool

CHAPTER

SEVEN

OCTTREE

An Octtree is an extension of the Quadtree into a third dimension. In standard Octtree implementations the third dimension is treated as another spatial dimension, in that distance checks are performed using Euclidean distances. Here, the third dimension is considered to be a time dimension. Any look-ups using the Octtree require a timedelta to be provided, so that any records falling within the spatial range are returned only if they also fall within the time range defined by the timedelta.

Whilst the Quadtree divides into 4 branches after the capacity is reached, the Octtree divides into 8 branches. The divisions are at the longitude midpoint, the latitude midpoint, and the datetime midpoint of the boundary.

The implementation of Octtree within this library, the `OctTree` class, utilises the Haversine distance as a metric for identifying records within the queried region. This allows the Octtree to account for the spherical geometry of the Earth. Boundary checks with query regions also account for the wrapping of longitude at -180, 180 degrees.

The `OctTree` object is defined by a bounding box in space and time, i.e. boundaries at the western, eastern, southern, and northern edges as well as the start and end datetimes of the data that will be inserted into the `OctTree`. Additionally, a capacity and maximum depth can be provided. If the capacity is exceeded whilst inserting records the `OctTree` will divide and new records will be inserted into the appropriate branch `OctTree`. The maximum depth is the maximum height of the `OctTree`, if capacity is also specified then this will be overridden if the `OctTree` is at this depth, and the `OctTree` will not divide.

7.1 Documentation

7.1.1 Inserting Records

A `SpaceTimeRecord` can be added to an `OctTree` with `OctTree.insert` which will return `True` if the operation was successful, `False` otherwise. The `OctTree` is modified in place. Records that fall outside the bounds of the `OctTree` will not be inserted as the boundary is fixed.

7.1.2 Removing Records

A `SpaceTimeRecord` can be removed from an `OctTree` with `OctTree.remove` which will return `True` if the operation was successful, `False` otherwise. The `OctTree` is modified in place.

7.1.3 Querying

The `OctTree` class defined in `geotrees.octtree` can be queried in the following ways:

- with a `SpaceTimeRecord`, a spatial range, and a time range (specified by a `datetime.timedelta`) with `OctTree.nearby_points`. All points within the spatial range and time range of the `SpaceTimeRecord` will be returned in a list. The `Record` can be excluded from the results if the `exclude_self` argument is set.
- with a `SpaceTimeRectangle` using `OctTree.query`. All points within the specified `SpaceTimeRectangle` will be returned in a list.

- with a `SpaceTimeEllipse` using `OctTree.query_ellipse`. All points within the specified `SpaceTimeEllipse` will be returned in a list.

7.2 Example

```
from geotrees import OctTree, SpaceTimeRecord, SpaceTimeRectangle
from datetime import datetime, timedelta
from random import choice
from polars import datetime_range

lon_range = list(range(-180, 180))
lat_range = list(range(-90, 90))

dates = datetime_range(
    start=datetime(2009, 1, 1, 0, 0),
    end=datetime(2009, 2, 1, 0, 0),
    interval=timedelta(hours=1),
    closed="left",
    eager=True,
)
N_samples = 1000

# Construct Tree
boundary = SpaceTimeRectangle(
    west=-180,
    east=180,
    south=-90,
    north=90,
    start=datetime(2009, 1, 1, 0),
    end=datetime(2009, 1, 2, 23),
)  # Full domain
octtree = OctTree(boundary)

# Populate the tree
records: list[SpaceTimeRecord] = [
    SpaceTimeRecord(
        choice(lon_range),
        choice(lat_range),
        choice(dates)
    ) for _ in range(N_samples)
]
for record in records:
    octtree.insert(record)

test_value: SpaceTimeRecord = SpaceTimeRecord(
    lon=47.6, lat=-31.1, datetime=datetime(2009, 1, 23, 17, 41)
)
dist: float = 340  # km
t_dist = timedelta(hours=4)

# Find all Records that are 340km away from test_value, and within 4 hours
# of test_value
```

(continues on next page)

(continued from previous page)

```
neighbours: list[SpaceTimeRecord] = octtree.nearby_points(
    test_value, dist, t_dist
)
```

7.3 octtree Module

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

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 branches as the capacity is reached, points within the OctTree are not distributed to the branch OctTrees. Additional SpaceTimeRecords are then added to the branch where they fall within the branch OctTree's boundary.

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 point into the OctTree.

Parameters
point (*SpaceTimSpaceTimeeRecord*) – The point to insert

Returns
True if the point was inserted into 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*, *exclude_self=False*)

Get all SpaceTimeRecords contained in the OctTree that are nearby another query SpaceTimeRecord.

Query the OctTree to find all SpaceTimeRecords within the OctTree that are nearby to the query SpaceTimeRecord. This search should be faster than searching through all records, since only OctTree branch 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** (*List[SpaceTimeRecord] | 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 branch OctTrees.
- **exclude_self** (*bool*) – Optionally exclude the query point from the results if the query point is in the OctTree

Returns

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.

Return type

list[*SpaceTimeRecord*]

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

Get SpaceTimeRecords contained within the OctTree that fall in a SpaceTimeRectangle

Parameters

rect (SpaceTimeRectangle)

Returns

The SpaceTimeRecord values contained within the OctTree that fall within the bounds of rect.

Return type

List[*SpaceTimeRecord*]

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

Get SpaceTimeRecords contained within the OctTree that fall in a SpaceTimeEllipse

Parameters

ellipse (SpaceTimeEllipse)

Returns

The SpaceTimeRecord values contained within the OctTree that fall within the bounds of ellipse.

Return type

List[*SpaceTimeRecord*]

remove(*point*)

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

Parameters

point (SpaceTimeRecord) – The point to remove

Returns

True if the point is removed

Return type
bool

CHAPTER

EIGHT

K-D-TREE

A K-D-Tree is a data structure that operates in a similar way to bisection or a binary tree, and can be used to find the nearest neighbour. For k-dimensional data (i.e. the data has k features), a binary tree is constructed by bisecting the data along each of the k dimensions in sequence. The first layer bisects the data along the first dimension, the second layer bisects each of the previous bisection results along the 2nd dimension (the data is now partitioned into 4), and so on. The pattern repeats after the k-th layer, until a single point of data remains in each leaf node. A K-D-Tree that bisects data and results in each leaf node containing a single value is called referred to as a balanced K-T-Tree.

To find the data point that is closest to a point in the tree, one descends the tree comparing the query point to the partition value in each dimension. The final leaf node should be the closest point, however there may be a point closer if the query point is close to a previous partition value, so some back tracking is performed to either confirm, or update, the closest point.

A K-D-Tree can typically find the nearest neighbour in $O(\log(n))$ time complexity, and the data structure has $O(n)$ space-complexity.

Most implementations of K-D-Tree assume that the coordinates use a cartesian geometry and therefore use a simple Euclidean distance to identify the nearest neighbour. The implementation in `geotrees.kdtree` assumes a spherical geometry on the surface of the Earth and uses the Haversine distance to identify neighbours. The implementation has been designed to account for longitude wrapping at -180, 180 degrees. The `geotrees.kdtree.KDTree` class is a 2-D-Tree, the dimensions are longitude and latitude. The object is initialised with data in the form of a list of `geotrees.quadtree.Record` objects. A maximum depth value (`max_depth`) can be provided, if this is set then the partitioning will stop after `max_depth` partitionings, the leaf nodes may contain more than one `Record`.

8.1 Example

```
from geotrees import KDTree, Record
from random import choice

lon_range = list(range(-180, 180))
lat_range = list(range(-90, 90))
N_samples = 1000

records: list[Record] = [Record(choice(lon_range), choice(lat_range)) for _ in range(N_
↪samples)]
# Construct Tree
kdtree = KDTree(records)

test_value: Record = Record(lon=47.6, lat=-31.1)
neighbours, dist = kdtree.query(test_value)
```

8.2 Documentation

> **Note**
>
> Insertion and deletion operations may cause the `KDTree` to become un-balanced.

8.2.1 Inserting Records

A `Record` can be inserted in to a `KDTree` with the `KDTree.insert` method. The method will return `True` if the `Record` was inserted into the `KDTree`, `False` otherwise. A `Record` will not be added if it is already contained within the `KDTree`, to add the `Record` anyway use the `KDTree._insert` method.

8.2.2 Removing Records

A `Record` can be removed from a `KDTree` with the `KDTree.delete` method. The method will return `True` if the `Record` was successfully removed, `False` otherwise (for example if the `Record` is not contained within the `KDTree`).

8.2.3 Querying

The nearest neighbour `Record` contained within a `KDTree` to a query `Record` can be found with the `KDTree.query` method. This will return a tuple containing the list of `Record` objects from the `KDTree` with minimum distance to the query `Record`, and the minimum distance.

8.3 kdtree Module

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 `geotrees.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 geotrees.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]`

CHAPTER

NINE

ADDITIONAL MODULES

9.1 GreatCircle

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

class `geotrees.great_circle.`**`GreatCircle`**(*lon0*, *lat0*, *lon1*, *lat1*, *radius=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

geotrees.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*

geotrees.great_circle.**polar_to_cartesian**(*lon*, *lat*, *radius=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)

9.2 Distance Metrics

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

`geotrees.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

`geotrees.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]

`geotrees.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

`geotrees.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

`geotrees.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

9.3 Utils

Utility functions. Including Error classes and Warnings.

exception `geotrees.utils.`**`DateWarning`**

Warning for Datetime Value

exception `geotrees.utils.`**`LatitudeError`**

Error for invalid Latitude Value

PYTHON MODULE INDEX

g

INDEX

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T