geolib

3.3.1 • Public • Published 9 months ago

Readme

Explore BETA

0 Dependencies

264 Dependents

31 Versions

Install

```
npm i geolib
```

Weekly Downloads

149,165

Version	License
3.3.1	MIT
Unpacked Size	**Total Files**
102 kB	91
Issues	**Pull Requests**
26	13

Homepage

github.com/manuelbieh/geolib#readme

Repository

github.com/manuelbieh/geolib

Last publish

9 months ago

Collaborators

Try on RunKit

Report malware

Geolib

Library to provide basic geospatial operations like distance calculation, conversion of decimal coordinates to sexagesimal and vice versa, etc. This library is currently **2D**, meaning that altitude/elevation is not yet supported by any of its functions!

Changelog

A detailed changelog can be found in **CHANGELOG.md**

Install

```
npm install geolib
```

```
yarn add geolib
```

Usage

There is a **UMD** build and an **ES Module** build. You can either use the UMD build in Node like any other library:

```
const geolib = require('geolib');
```

or in the browser by using a simple script element:

```
<script src="lib/geolib.js"></script>
```

If you load it in the browser, you can access all the functions via `window.geolib`.

If you're working with a bundler (like Webpack or Parcel) or have an environment that supports ES Modules natively, you can either import certain functions from the package directly:

```
import { getDistance } from 'geolib';
```

or load the whole library:

```
import * as geolib from 'geolib';
```

or you can import single functions directly to potentially make use of treeshaking (recommended):

```
import getDistance from 'geolib/es/getDistance';
```

General

This library is written in TypeScript. You don't have to know TypeScript to use Geolib but the **type definitions** give you valuable information about the general usage, input parameters etc.

Supported values and formats

All methods that are working with coordinates accept either an object with a `lat` / `latitude` **and** a `lon` / `lng` / `longitude` property, **or** a GeoJSON coordinates array, like: `[lon, lat]`. All values can be either in decimal (`53.32055555555556`) or sexagesimal (`53° 21' 16"`) format

values can be either in decimal (`53.471`) or sexagesimal (`53° 21' 16"`) format.

Distance values are **always** floats and represent the distance in **meters**.

Functions

`getDistance(start, end, accuracy = 1)`

Calculates the distance between two geo coordinates.

This function takes up to 3 arguments. First 2 arguments must be valid `GeolibInputCoordinates` (e.g. `{latitude: 52.518611, longitude: 13.408056}`). Coordinates can be in sexagesimal or decimal format. The third argument is accuracy (in meters). By default the accuracy is 1 meter. If you need a more accurate result, you can set it to a lower value, e.g. to `0.01` for centimeter accuracy. You can set it higher to have the result rounded to the next value that is divisible by your chosen accuracy (e.g. `25428` with an accuracy of `100` becomes `25400`).

```
getDistance(
    { latitude: 51.5103, longitude: 7.49347 },
    { latitude: "51° 31' N", longitude: "7° 28' E" }
);
```

```
// Working with W3C Geolocation API
navigator.geolocation.getCurrentPosition(
    (position) => {
        console.log(
            'You are ',
            geolib.getDistance(position.coords, {
                latitude: 51.525,
                longitude: 7.4575,
            }),
            'meters away from 51.525, 7.4575'
        );
    },
    () => {
        alert('Position could not be determined.');
    }

);
```

Returns the distance in meters as a numeric value.

`getPreciseDistance(start, end[, int accuracy])`

Calculates the distance between two geo coordinates. This method is more accurate then `getDistance`, especially for long distances but it is also slower. It is using the Vincenty inverse formula for ellipsoids.

It takes the same (up to 3) arguments as `getDistance`.

```
geolib.getPreciseDistance(
    { latitude: 51.5103, longitude: 7.49347 },
    { latitude: "51° 31' N", longitude: "7° 28' E" }
);
```

`getCenter(coords)`

Calculates the geographical center of all points in a collection of geo coordinates. Takes an array of coordinates and calculates the center of it.

```
geolib.getCenter([
    { latitude: 52.516272, longitude: 13.377722 },
    { latitude: 51.515, longitude: 7.453619 },
    { latitude: 51.503333, longitude: -0.119722 },
]);
```

Returns an object:

```
{
    "latitude": centerLat,
    "longitude": centerLon
}
```

`getCenterOfBounds(coords)`

Calculates the center of the bounds of geo coordinates.

Takes an array of coordinates, calculate the border of those, and gives back the center of that rectangle.

On polygons like political borders (eg. states), this may gives a closer result to human expectation, than `getCenter`, because that function can be disturbed by uneven distribution of point in different sides.

point in different sides.

Imagine the US state Oklahoma: `getCenter` on that gives a southern point, because the southern border contains a lot more nodes, than the others.

```
geolib.getCenterOfBounds([
    { latitude: 51.513357512, longitude: 7.45574331 },
    { latitude: 51.515400598, longitude: 7.45518541 },
    { latitude: 51.516241842, longitude: 7.456494328 },
    { latitude: 51.516722545, longitude: 7.459863183 },
    { latitude: 51.517443592, longitude: 7.463232037 },
]);
```

Returns an object:

```
{
    "latitude": centerLat,
    "longitude": centerLng
}
```

`getBounds(points)`

Calculates the bounds of geo coordinates.

```
geolib.getBounds([
    { latitude: 52.516272, longitude: 13.377722 },
    { latitude: 51.515, longitude: 7.453619 },
    { latitude: 51.503333, longitude: -0.119722 },
]);
```

It returns minimum and maximum latitude and minimum and maximum longitude as an object:

```
{
    "minLat": minimumLatitude,
    "maxLat": maximumLatitude,
    "minLng": minimumLongitude,
    "maxLng": maximumLongitude,
}
```

`isPointInPolygon(point, polygon)`

Checks whether a point is inside of a polygon or not.

```
geolib.isPointInPolygon({ latitude: 51.5125, longitude: 7.485 }, [
    { latitude: 51.5, longitude: 7.4 },
    { latitude: 51.555, longitude: 7.4 },
    { latitude: 51.555, longitude: 7.625 },
    { latitude: 51.5125, longitude: 7.625 },
]);
```

Returns `true` or `false`

`isPointWithinRadius(point, centerPoint, radius)`

Checks whether a point is inside of a circle or not.

```
// checks if 51.525/7.4575 is within a radius of 5 km from 51.5175/7.4678
geolib.isPointWithinRadius(
    { latitude: 51.525, longitude: 7.4575 },
    { latitude: 51.5175, longitude: 7.4678 },
    5000
);
```

Returns `true` or `false`

`getRhumbLineBearing(origin, destination)`

Gets rhumb line bearing of two points. Find out about the difference between rhumb line and great circle bearing on Wikipedia. Rhumb line should be fine in most cases:

http://en.wikipedia.org/wiki/Rhumb_line#General_and_mathematical_description

Function is heavily based on Doug Vanderweide's great PHP version (licensed under GPL 3.0)
http://www.dougv.com/2009/07/13/calculating-the-bearing-and-compass-rose-direction-between-two-latitude-longitude-coordinates-in-php/

```
geolib.getRhumbLineBearing(
    { latitude: 52.518611, longitude: 13.408056 },
    { latitude: 51.519475, longitude: 7.46694444 }
);
```

Returns calculated bearing as number.

`getGreatCircleBearing(origin, destination)`

Gets great circle bearing of two points. This is more accurate than rhumb line bearing but also slower.

```
geolib.getGreatCircleBearing(
    { latitude: 52.518611, longitude: 13.408056 },
    { latitude: 51.519475, longitude: 7.46694444 }
);
```

Returns calculated bearing as number.

`getCompassDirection(origin, destination, bearingFunction = getRhumbLineBearing)`

Gets the compass direction from an origin coordinate to a destination coordinate. Optionally a function to determine the bearing can be passed as third parameter. Default is `getRhumbLineBearing`.

```
geolib.getCompassDirection(
    { latitude: 52.518611, longitude: 13.408056 },
    { latitude: 51.519475, longitude: 7.46694444 }
);
```

Returns the direction (e.g. `NNE`, `SW`, `E`, ...) as string.

`orderByDistance(point, arrayOfPoints)`

Sorts an array of coords by distance to a reference coordinate.

```
geolib.orderByDistance({ latitude: 51.515, longitude: 7.453619 }, [
    { latitude: 52.516272, longitude: 13.377722 },
    { latitude: 51.518, longitude: 7.45425 },
    { latitude: 51.503333, longitude: -0.119722 },
]);
```

Returns an array of points ordered by their distance to the reference point.

`findNearest(point, arrayOfPoints)`

Finds the single one nearest point to a reference coordinate. It's actually just a convenience method that uses `orderByDistance` under the hood and returns the first result.

```
geolib.findNearest({ latitude: 52.456221, longitude: 12.63128 }, [
    { latitude: 52.516272, longitude: 13.377722 },
    { latitude: 51.515, longitude: 7.453619 },
    { latitude: 51.503333, longitude: -0.119722 },
    { latitude: 55.751667, longitude: 37.617778 },
    { latitude: 48.8583, longitude: 2.2945 },
    { latitude: 59.3275, longitude: 18.0675 },
    { latitude: 59.916911, longitude: 10.727567 },
]);
```

Returns the point nearest to the reference point.

`getPathLength(points, distanceFunction = getDistance)`

Calculates the length of a collection of coordinates. Expects an array of points as first argument and optionally a function to determine the distance as second argument. Default is `getDistance`.

```
geolib.getPathLength([
    { latitude: 52.516272, longitude: 13.377722 },
    { latitude: 51.515, longitude: 7.453619 },
    { latitude: 51.503333, longitude: -0.119722 },
]);
```

Returns the length of the path in meters as number.

`getDistanceFromLine(point, lineStart, lineEnd)`

Gets the minimum distance from a point to a line of two points.

```
geolib.getDistanceFromLine(
    { latitude: 51.516, longitude: 7.456 },
    { latitude: 51.512, longitude: 7.456 },
    { latitude: 51.516, longitude: 7.459 }
);
```

Returns the shortest distance to the given line as number.

`getBoundsOfDistance(point, distance)`

Computes the bounding coordinates of all points on the surface of the earth less than or equal to the specified great circle distance

to the specified great circle distance.

```
geolib.getBoundsOfDistance(
    { latitude: 34.090166, longitude: -118.276736555556 },
    1000
);
```

Returns an array with the southwestern and northeastern coordinates.

`isPointInLine(point, lineStart, lineEnd)`

Calculates if given point lies in a line formed by start and end.

```
geolib.isPointInLine(
    { latitude: 0, longitude: 10 },
    { latitude: 0, longitude: 0 },
    { latitude: 0, longitude: 15 }
);
```

`sexagesimalToDecimal(value)`

Converts a sexagesimal coordinate into decimal format

```
geolib.sexagesimalToDecimal(`51° 29' 46" N`);
```

Returns the new value as decimal number.

`decimalToSexagesimal(value)`

Converts a decimal coordinate to sexagesimal format

```
geolib.decimalToSexagesimal(51.49611111); // -> 51° 29' 46`
```

Returns the new value as sexagesimal string.

`geolib.getLatitude(point, raw = false)`

`geolib.getLongitude(point, raw = false)`

Returns the latitude/longitude for a given point **and** converts it to decimal. If the second argument is set to true it does **not** convert the value to decimal.

```
geolib.getLatitude({ lat: 51.49611, lng: 7.38896 }); // -> 51.49611
geolib.getLongitude({ lat: 51.49611, lng: 7.38896 }); // -> 7.38896
```

Returns the value as decimal or in its original format if the second argument was set to true.

`toDecimal(point)`

Checks if a coordinate is already in decimal format and, if not, converts it to. Works with single values (e.g. `51° 32' 17"`) and complete coordinates (e.g. `{lat: 1, lon: 1}`) as long as it in a **supported format**.

```
geolib.toDecimal(`51° 29' 46" N`); // -> 51.59611111
geolib.toDecimal(51.59611111); // -> 51.59611111
```

Returns a decimal value for the given input value.

`computeDestinationPoint(point, distance, bearing, radius = earthRadius)`

Computes the destination point given an initial point, a distance (in meters) and a bearing (in degrees). If no radius is given it defaults to the mean earth radius of 6,371,000 meters.

Attention: this formula is not *100%* accurate (but very close though).

```
geolib.computeDestinationPoint(
    { latitude: 52.518611, longitude: 13.408056 },
    15000,
    180
);
```

```
geolib.computeDestinationPoint(
    [13.408056, 52.518611]
    15000,
    180
);
```

Returns the destination in the same format as the input coordinates. So if you pass a GeoJSON point, you will get a GeoJSON point.

`getAreaOfPolygon(points)`

Calculates the surface area of a polygon.

```
geolib.getAreaOfPolygon([
    [7.453635617650258, 51.49320556213869],
    [7.454583481047989, 51.49328893754685],
    [7.454778172179346, 51.49240881084831],
    [7.453832678225655, 51.49231619246726],
    [7.453635617650258, 51.49320556213869],
]);
```

Returns the result as number in square meters.

`getCoordinateKeys(point)`

Gets the property names of that are used in the point in a normalized form:

```
geolib.getCoordinateKeys({ lat: 1, lon: 1 });
// -> { latitude: 'lat', longitude: 'lon' }
```

Returns an object with a `latitude` and a `longitude` property. Their values are the property names for latitude and longitude that are used in the passed point. Should probably only be used internally.

`getCoordinateKey(point, keysToLookup)`

Is used by `getCoordinateKeys` under the hood and returns the property name out of a list of possible names.

```
geolib.getCoordinateKey({ latitude: 1, longitude: 2 }, ['lat', 'latitude']);
// -> latitude
```

Returns the name of the property as string or `undefined` if no there was no match.

`isValidCoordinate(point)`

Checks if a given point has at least a **latitude** and a **longitude** and is in a supported format.

```
// true:
geolib.isValidCoordinate({ latitude: 1, longitude: 2 });

// false, longitude is missing:
```

```
geolib.isValidCoordinate({ latitude: 1 });

// true, GeoJSON format:
geolib.isValidCoordinate([2, 1]);
```

Returns `true` or `false`.

`getSpeed(startPointWithTime, endPointWithTime)`

Calculates the speed between two points within a given time span.

```
geolib.getSpeed(
    { latitude: 51.567294, longitude: 7.38896, time: 1360231200880 },
    { latitude: 52.54944, longitude: 13.468509, time: 1360245600880 }
);
```

Return the speed in meters per second as number.

`convertSpeed(value, unit)`

Converts the result from `getSpeed` into a more human friendly format. Currently available units are `mph` and `kmh`.

Units

`unit` can be one of:

- kmh (kilometers per hour)
- mph (miles per hour)

```
geolib.convertSpeed(29.8678, 'kmh'));
```

Returns the converted value as number.

`convertDistance(value, unit)`

Converts a given distance (in meters) into another unit.

Units

`unit` can be one of:

- m (meter)
- km (kilometers)
- cm (centimeters)

- mm (millimeters)
- mi (miles)
- sm (seamiles)
- ft (feet)
- in (inches)
- yd (yards)

```
geolib.convertDistance(14200, 'km'); // 14.2
geolib.convertDistance(500, 'km'); // 0.5
```

Returns the converted distance as number.

`convertArea(value, unit)`

Converts the result from `getAreaForPolygon` into a different unit.

Units

`unit` can be one of:

- m2, sqm (square meters)
- km2, sqkm (square kilometers)
- ha (hectares)
- a (ares)
- ft2, sqft (square feet)
- yd2, sqyd (square yards)
- in2, sqin (square inches)

```
geolib.convertArea(298678, 'km2'));
```

Returns the converted area as number.

`wktToPolygon(wkt)`

Converts the Well-known text (a.k.a WKT) to polygon that Geolib understands.
https://en.wikipedia.org/wiki/Well-known_text_representation_of_geometry#Geometric_Objects

```
geolib.wktToPolygon('POLYGON ((30 10.54321, 40 40, 20 40, 10 20, 30 10))');
// [
//     { latitude: 10.54321, longitude: 30 },
//     { latitude: 40, longitude: 40 },
//     { latitude: 40, longitude: 20 },
```

```
//     { latitude: 20, longitude: 10 },
//     { latitude: 10, longitude: 30 },}
// ]
```

Returns the array of coordinates.

Breaking Changes in 3.0.0 and migration from 2.x.x

In version 3.0.0 I'm trying to get a little bit *back to the roots*. **Geolib** was once started because I needed a handful of methods to perform very specific geo related tasks like getting the distance or the direction between two points. Since it was one of the very first libraries on npm back then to do these kind of things in a very simple way it became very popular (with more than 300k downloads per month as of April 2019!) and as a consequence got a lot of contributions over the years. Many of which I just merged as long as they had accompanying tests, without looking at consistency, conventions, complexity, coding style or even the overall quality of the functions that I sometimes didn't even fully understand.

I have now cleaned up the codebase completely, rebuilt the entire library "from scratch", unified all the function arguments, removed a few functions where I wasn't sure if they should be in here (feel free to add them back of you're using them!) or if they were even used (did a few searches on GitHub for the function names, turned out there are zero results).

Elevation support was dropped, as well as a few functions that unnecessarily made the library really large in size (e.g. `isPointInsideRobust` alone was over 700[!] lines of code and was basically taken from a **different library**).

I removed Grunt from the build process, added "modern" tools like ESLint and Prettier. I switched from Travis CI to Circle CI and I am in the process of further automating the release process of new versions using `semantic-release` and `conventional-commits`. I also switched from pure JavaScript to TypeScript because I think it does have some benefits.

- All functions are pure functions now. No input data is mutated anymore. You give the same input, you get the same output. No side effects or whatsoever.
- I changed the default `getDistance` function from being the slow, accurate one to being the fast, slightly inaccurate one. The old `getDistance` function is now named `getPreciseDistance` while the old `getDistanceSimple` function is now the default `getDistance` function. You can, however, pass `getPreciseDistance` as argument to any function that uses distance calculation internally.
- Artificial limitation to 8 decimal places in decimal coordinates was removed
- `getBoundsOfDistance()` now returns the *exact* coordinates due to the removal of the artificial 8 decimal place limitation

- `getCompassDirection()` does no longer return an object with an *exact* and a *rough* direction but only the exact direction as string
- third argument to `getCompassDirection()` is no longer a string ("circle", "line") but a function to determine the bearing (you can pass `getRhumbLineBearing` or `getGreatCircleBearing`). The function receives the origin and the destination as first and second argument. If no 3rd argument was given, `getRhumbLineBearing(origin, dest)` is used by default.
- There is now a new helper function `roughCompassDirection(exact)` if you *really* only need a very rough (and potentially inaccurate or inappropriate) direction. Better don't use it.
- `orderByDistance()` does no longer modify its input so does not add a `distance` and `key` property to the returned coordinates.
- The result of `getSpeed()` is now always returned as **meters per second**. It can be converted using the new convenience function `convertSpeed(mps, targetUnit)`
- Relevant value (usually point or distance) is now consistently the **first** argument for each function (it wasn't before, how confusing is that?)
- `findNearest()` does no longer take `offset` and `limit` arguments. It's only a convenience method to get the single one nearest point from a set of coordinates. If you need more than one, have a look at the implementation and implement your own logic using `orderByDistance`
- Whereever distances are involved, they are returned as meters or meters per second. No more inconsistent defaults like kilometers or kilometers per hour.
- The method how sexagesimal is formatted differs a little bit. It may now potentially return ugly float point units like `52° 46' 21.0004"` in rare cases but it is also more accurate than it was before.
- Dropped support for Meteor (feel free to add it back if you like)

Functions with the same name

- `computeDestinationPoint`
- `getBounds`
- `getBoundsOfDistance`
- `getCenter`
- `getCenterOfBounds`
- `getCompassDirection`
- `getDistanceFromLine`
- `getPathLength`
- `getRhumbLineBearing`
- `getSpeed`
- `isDecimal`
- `isPointInLine`
- `isPointNearLine`
- `isSexagesimal`
- `orderByDistance`

Renamed functions

- `getKeys` renamed to `getCoordinateKeys`
- `validate` renamed to `isValidCoordinate`
- `getLat` renamed to `getLatitude`
- `getLon` renamed to `getLongitude`
- `latitude` -> renamed to `getLatitude`
- `longitude` -> renamed to `getLongitude`
- `convertUnit` -> remamed to convertDistance, because name was too ambiguous
- `useDecimal` renamed to `toDecimal`
- `decimal2sexagesimal` renamed to `decimalToSexagesimal`
- `sexagesimal2decimal` renamed to `sexagesimalToDecimal`
- `getDistance` renamed to `getPreciseDistance`
- `getDistanceSimple` renamed to `getDistance`
- `isPointInside` renamed to `isPointInPolygon`
- `isPointInCircle` renamed to `isPointWithinRadius`
- `getBearing` renamed to `getGreatCircleBearing` to be more explicit

Removed functions

- `getElev` -> removed
- `elevation` -> removed
- `coords` -> removed (might be re-added as getCoordinate or getNormalizedCoordinate)
- `ll` -> removed (because wtf?)
- `preparePolygonForIsPointInsideOptimized` -> removed due to missing documentation and missing tests
- `isPointInsideWithPreparedPolygon` -> removed due to missing documentation
- `isInside` alias -> removed (too ambiguous) - use `isPointInPolygon` or `isPointWithinRadius`
- `withinRadius` -> removed, use `isPointWithinRadius`
- `getDirection` alias -> removed (unnecessary clutter) - use `getCompassDirection`

Added functions

- `getAreaOfPolygon` to calculate the area of a polygon
- `getCoordinateKey` to get a property name (e.g. `lat` or `lng` of an object based on an array of possible names)

Keywords

none

Support

Help

Community

Advisories

Status

Contact npm

Company

About

Blog

Press

Terms & Policies

Policies

Terms of Use

Code of Conduct

Privacy