|  |  |  |  |
| --- | --- | --- | --- |
| Sr. | Ambition | ETA | Dependency? |
| 1. | **Finish dissertation with distinction.** | 30 | Focus, planning, coding, drafting |
| 2. | I have a nice paying job. |  |  |
| 3. | I own a good car. |  |  |
| 4. | I repay my education loan. |  |  |
| 5. | Parents UK trip. Sisters are welcome! |  |  |
| 6. | Vacations Two-gether. |  |  |

ChatGPT Prompt:

I'm working on a project to build a fleet scheduler for pothole repairing. The aim is to help strategize in terms of path-planning, time-taken, number of bots required, energy/battery consumption, cost evaluation.

I shall have several bots that can be deployed to cover a neighbourhood (postcode) and they shall go to every road defect to repair them.

The starting point can be configured.

Each bot shall have a load capacity.

The road defects include cracks, dents, and potholes.

The repair time shall comprise of a variable portion proportional to the size of the defect, and a constant portion.

Programming shall be done in Python.

from openrouteservice.directions import directions

from openrouteservice.distance\_matrix import distance\_matrix

from openrouteservice.elevation import elevation\_point

from openrouteservice.elevation import elevation\_line

from openrouteservice.isochrones import isochrones

from openrouteservice.geocode import pelias\_search

from openrouteservice.geocode import pelias\_autocomplete

from openrouteservice.geocode import pelias\_structured

from openrouteservice.geocode import pelias\_reverse

from openrouteservice.places import places

from openrouteservice.optimization import optimization

|  |  |  |
| --- | --- | --- |
| directions | directions | Routes between given locations [w/ criteria incl. vehicle details] |
| distance-matrix | distance-matrix |  |
| elevation | elevation-point |  |
| elevation-line |  |
| isochrones | isochrones |  |
| geocode | pelias-search |  |
| pelias-autocomplete |  |
| pelias-structured |  |
| pelias-reverse |  |
| places | places |  |
| optimization | optimization |  |

Describe the project:

Title: Data Driven Tool for Strategic Road Repairing using a fleet of bots.

1. Task: Strategize Pothole repairing in a neighbourhood.
2. Input:
   1. UK Postcode,
   2. pothole details
      1. location coordinates,
      2. road defect type,
      3. defect dimensions.
3. Evaluate:
   1. estimated filling material required,
   2. time to travel,
   3. time to repair.
   4. available material (in the vehicle) at any time. (Hence, need to refill).
4. Output:
   1. Travel routes (JSON) for n=5,10,15 bots in the given area.
   2. Compare different estimated timelines.
   3. Show comparison charts (in addition to matrix).
5. Fleet of ‘n’ bots shall accomplish the task.
6. Bots have vehicle profile: driving-car for now, then it can be changed to a custom profile.

Directions Module

Directions Method

def directions(client,

coordinates,

profile='driving-car',

format\_out=None,

format='json',

preference=None,

units=None,

language=None,

geometry=None,

geometry\_simplify=None,

instructions=None,

instructions\_format=None,

alternative\_routes=None,

roundabout\_exits=None,

attributes=None,

maneuvers=None,

radiuses=None,

bearings=None,

skip\_segments=None,

continue\_straight=None,

elevation=None,

extra\_info=None,

suppress\_warnings=None,

optimized=None,

optimize\_waypoints=None,

options=None,

validate=True,

dry\_run=None):

"""Get directions between an origin point and a destination point.

For more information, visit https://go.openrouteservice.org/documentation/.

:param coordinates: The coordinates tuple the route should be calculated

from. In order of visit.

:type origin: list or tuple of coordinate lists or tuples

:param profile: Specifies the mode of transport to use when calculating

directions. One of ["driving-car", "driving-hgv", "foot-walking",

"foot-hiking", "cycling-regular", "cycling-road","cycling-mountain",

"cycling-electric",]. Default "driving-car".

:type profile: string

:param format: Specifies the response format. One of ['json', 'geojson', 'gpx']. Default "json".

Geometry format for "json" is Google's encodedpolyline. The GPX schema the response is validated

against can be found here:

https://raw.githubusercontent.com/GIScience/openrouteservice-schema/master/gpx/v1/ors-gpx.xsd.

:type format: str

:param format\_out: DEPRECATED.

:type format: str

:param preference: Specifies the routing preference. One of ["fastest, "shortest",

"recommended"]. Default "fastest".

:type preference: string

:param units: Specifies the distance unit. One of ["m", "km", "mi"]. Default "m".

:type units: string

:param language: Language for routing instructions. One of ["en", "de", "cn",

"es", "ru", "dk", "fr", "it", "nl", "br", "se", "tr", "gr"]. Default "en".

:type language: string

:param language: The language in which to return results.

:type language: string

:param geometry: Specifies whether geometry should be returned. Default True.

:type geometry: boolean

:param geometry\_simplify: Specifies whether to simplify the geometry.

Default False.

:type geometry\_simplify: boolean

:param instructions: Specifies whether to return turn-by-turn instructions.

Default True.

:type instructions: boolean

:param instructions\_format: Specifies the the output format for instructions.

One of ["text", "html"]. Default "text".

:type instructions\_format: string

:param alternative\_routes: Specifies whether alternative routes are computed,

and parameters for the algorithm determining suitable alternatives. Expects

3 keys: share\_factor (float), target\_count (int), weight\_factor (float).

More on https://openrouteservice.org/dev/#/api-docs/v2/directions/{profile}/geojson/post.

:type alternative\_routes: dict[int|float]

:param roundabout\_exits: Provides bearings of the entrance and all passed

roundabout exits. Adds the 'exit\_bearings' array to the 'step' object

in the response. Default False.

:type roundabout\_exits: boolean

:param attributes: Returns route attributes on ["detourfactor", "percentage"].

Must be a list of strings. Default None.

:type attributes: list or tuple of strings

:param maneuvers: Specifies whether the maneuver object is included into the step object or not. Default: false.

:type maneuvers bool

:param radiuses: A list of maximum distances (measured in

meters) that limit the search of nearby road segments to every given waypoint.

The values must be greater than 0, the value of -1 specifies no limit in

the search. The number of radiuses must correspond to the number of waypoints.

Default None.

:type radiuses: list or tuple

:param bearings: Specifies a list of pairs (bearings and

deviations) to filter the segments of the road network a waypoint can

snap to. For example bearings=[[45,10],[120,20]]. Each pair is a

comma-separated list that can consist of one or two float values, where

the first value is the bearing and the second one is the allowed deviation

from the bearing. The bearing can take values between 0 and 360 clockwise

from true north. If the deviation is not set, then the default value of

100 degrees is used. The number of pairs must correspond to the number

of waypoints. Setting optimized=false is mandatory for this feature to

work for all profiles. The number of bearings corresponds to the length

of waypoints-1 or waypoints. If the bearing information for the last waypoint

is given, then this will control the sector from which the destination

waypoint may be reached.

:type bearings: list or tuple or lists or tuples

:param skip\_segments: Specifies the segments that should be skipped in the route calculation.

A segment is the connection between two given coordinates and the counting starts with 1

for the connection between the first and second coordinate.

:type skip\_segments: list[int]

:param continue\_straight: Forces the route to keep going straight at waypoints

restricting u-turns even if u-turns would be faster. This setting

will work for all profiles except for driving-\*. In this case you will

have to set optimized=false for it to work. True or False. Default False.

:type continue\_straight: boolean

:param elevation: Specifies whether to return elevation values for points.

Default False.

:type elevation: boolean

:param extra\_info: Returns additional information on ["steepness", "suitability",

"surface", "waycategory", "waytype", "tollways", "traildifficulty", "roadaccessrestrictions"].

Must be a list of strings. Default None.

:type extra\_info: list or tuple of strings

:param suppress\_warnings: Tells the system to not return any warning messages and corresponding extra\_info.

For false the extra information can still be explicitly requested by adding it with the extra\_info parameter.

:type suppress\_warnings: bool

:param optimized: If set False, forces to not use Contraction Hierarchies.

:type optimized: bool

:param options: Refer to https://openrouteservice.org/dev/#/api-docs/v2/directions/{profile}/geojson/post for detailed documentation. Construct your own dict() following the example of the minified options object. Will be converted to json automatically.

:type options: dict

:param optimize\_waypoints: If True, a `Vroom <https://github.com/VROOM-Project/vroom>`\_ instance (ORS optimization endpoint) will optimize the `via` waypoints, i.e. all coordinates between the first and the last. It assumes the first coordinate to be the start location and the last coordinate to be the end location. Only requests with a minimum of 4 coordinates, no routing options and fastest weighting. Default False.

:type optimize\_waypoints: bool

:param validate: Specifies whether parameters should be validated before sending the request. Default True.

:type validate: bool

:param dry\_run: Print URL and parameters without sending the request.

:param dry\_run: boolean

:raises ValueError: When parameter has wrong value.

:raises TypeError: When parameter is of wrong type.

:returns: sanitized set of parameters

:rtype: call to Client.request()

"""

Future scope:

1. update the Maps services in real time or in advance about a planned downtime such as road closure.
2. Integrate cost factor.
3. Integrate and leverage sensors on the bots.
   1. Collision avoidance
   2. SLAM for real time environment observation
   3. Relate to Boston Dynamics’ dog-robot, with a weight carrying capacity.
4. Real time decision making from feedback mechanism to continuously update the path planning with new data from UGV – UML diag for framework