Documentation of the script “synthetic gps data”

The structure of this documentation consist of three chapters: 1) General overview and logic of the script 2) Step by step description of the script 3) Evaluation of the current state of the script

General overview:

By now this script follows such conditions and plot:

All users have three main types of locations: home, work and event (bars, parks, museums, etc.). Event could be either one of the regular locations of a user or random location (amenity) in the area.

All locations are derived from OpenStreetMap's conceptual data model of the physical world using osmnx library.

Each location has its id. Using location’s id we can derive three main values that are used in generating gps data: id of nearest node, x and y of centroid point of the location

Each movement of a user through a day can be plotted/created using two types of activity: 1) stay activity - synthetic gps points are densely located within small area (location), moving activity - synthetic gps points create a path (LineString) from one location to another (e.g. from event1 to event2). Thus, whole movement history of a user can be described as a repeated consecutive process of stay and moving activity that replace each other

GPS points of moving activity are plotted along the road network of the area derived from osmnx library

All users walk with mean velocity is about 4 km/h (but some minor deviations possible)

Plot of weekdays: a user stays in the morning at home till: 7-9 pm. Then go to work, stay at work till: 17-19pm. After work a user can go to some events (the number_of_events varies from 1 to 3) but most probably will go straight to the home (number_of_events=0). If a user goes to an event(s) they stay there 1-3 hours. If a number of events is big a user may stay till late night away from home but nevertheless he will return to home and will get up in the morning of next day (between 7-9 p.m) and will go to work

Plot of weekends: On Saturday and Sunday a user gets up between 10-14 p.m. A user can stay all weekend day at home (number_of_events=0) and go to bed at 22:00 p.m. - 02:00 a.m. but most probably a user will go to some events (the number of events varies from 1 to 4). After a user visits all events he always goes home. No matter when a user comes to home on Sunday he will get up between 7-9 p.m.

Another day starts when a user comes back to home no matter at what time and from what place (when a user stays at home all day we assume that he has already come to home)

The final DataFrame consists of four columns: user_id, timestamp, lon, lat

To easily analyze created synthetic gps data the GPS_PLOTS text document is written that reflects all movements of a user with their respective time

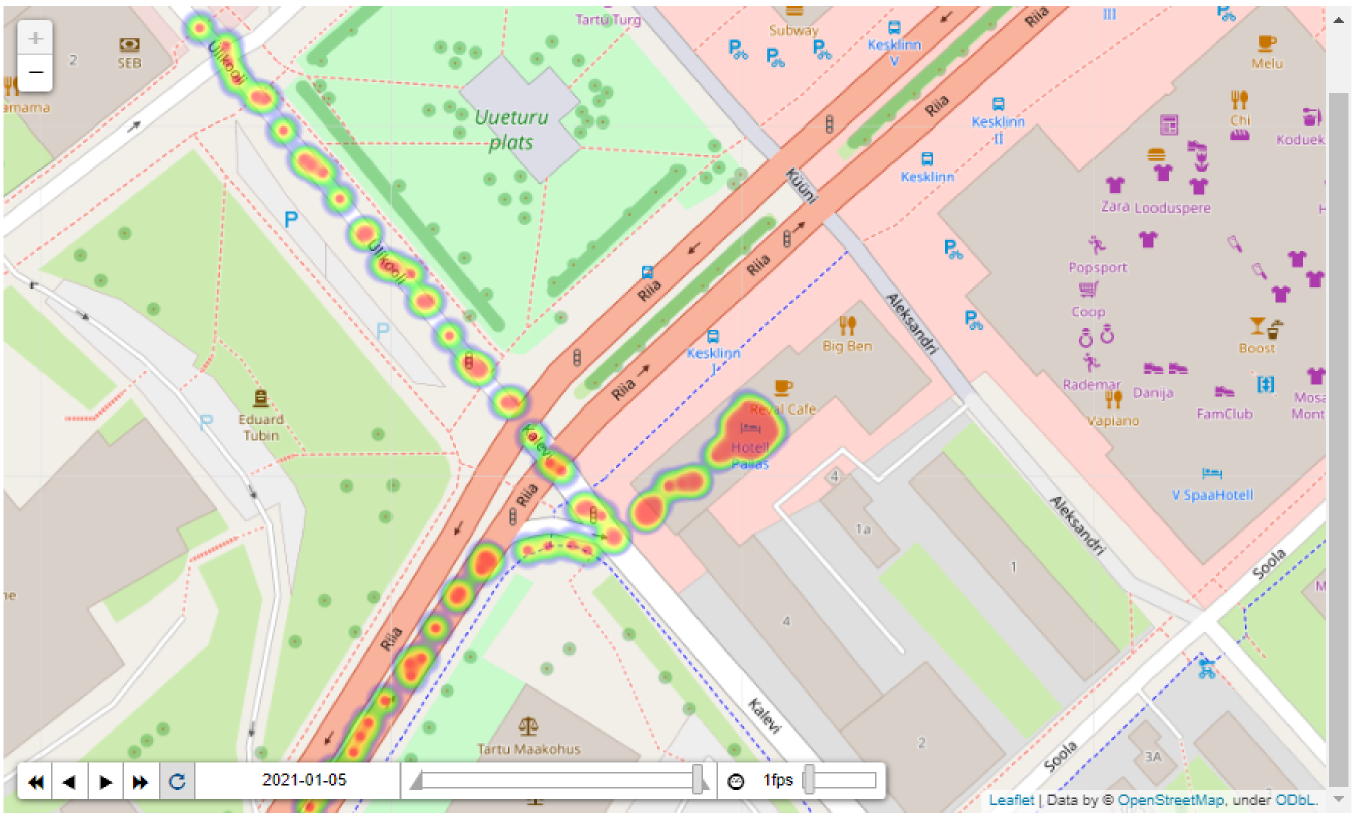
Step by step description of the script:

1. Derive a road graph of walk network type using place name (in current version Tartu)
2. Project the graph in local UTM zone , for Tartu it is 36N, units - meters (needed to calculate distance in meters)
3. From the already projected graph get nodes (one of the parts of the road network) that is stored in GeoDataFrame. For the script only geometry column is needed (for more details about the OSMnx library and it's structure please refer to https://github.com/gboeing/osmnx,)
4. Create the projection transformer from UTM zone to World Geodetic System 1984 (the final dataframe will have coordinates in WGS 84 since this projection seems to be the most comfortable to work with in future analyze or visualization)
5. For home and work locations the osm data tagged 'buildings' will be used, for event locations - tags 'amenities', 'leisure' and 'tourism services' (by now the script takes all tagged data in the area in order to make the final data various and extensive, however, you can tune the model to your needs and choose for example only apartments for buildings, or only restaurants for event locations for further research please refer to https://wiki.openstreetmap.org/wiki/Tags).
6. The data for home/work locations and event places are stored in two different GeoDataFrames, gdf_hw and gdf_event respectively. The geometry column is essential, the name column is additional. The data type of the geometry column of both GeoDataFrames is predominantly Polygon (but it can be point or MultiPolygon, to understand why, please refer to elements of OSM). For each GeoDataFrame the following process is done: project to local UTM zone, get coordinates of centroid point for each location, find the nearest road node id to the centroid and derives the node's coordinates, calculate the distance between the centroid point and its nearest node (in meters since the graph is projected), store all found data in respective GeoDataFrame.
7. To randomly derive work, home and regular locations for a user the function get_meaningful_locations is used. First it randomly chooses the index of gdf_hw GeoDataFrame (let's call it id of location) for work and home and their respective nodes, then calculates the distance between home and work (including the distance between each of two locations to their respective node). If the distance satisfies the condition the function goes to regular locations. The number of regular locations ids varies from 3 to 5, regular locations also have distance condition and take their ids from the gdf_event dataframe.
   The function returns home_id, work_id, list of regular location ids. The distance conditions can be discard, it is done only to make the final trajectories more sparse

8. Function get_regular_or_random_loc first creates id of random location taken again from gdf_event then with likelihood 60 vs 40 chooses either list of regular location ids or random location id, and if the list of regular location ids is chosen the function randomly takes one regular id from it. The global variable within function is essential to make sure that the current id and previous id is not the same (otherwise the LineString /path will not be created, since LineString needs at least two different points)
   Based on event id the function returns the id of nearest node to the event location, coordinates of event location (its centroid point) and event id itself (the last variable is needed only to be written in GPS_PLOTS text document)
9. Function get_static_points - one of the main functions, generates the nearby points around some coordinate (in the script - around the centroid point of a user's location). The function employed when we want a user to stay in some location (at home, at work, at event) in other words produce stay activity (please see Overview chapter)
   First it defines time_start (the time when the previous activity has ended, namely moving activity) and adds one minute (to surely meet velocity condition). Then it transforms the coordinates of the start point to WGS 84 projection (will be explained a little later).
   After that a while loop begins. It is important to define time_end since the function can generate data endlessly. Time_end can be defined in absolute way for example till 2022-07-25 10:00:00 or in relative way based on time_start e.g. time_start +timedelta(hours=2).
   Crucial part of the function and a while loop is the built-in function pyproj.Geod forward transformation that with given coordinates of start point, forward azimuth (angel), and distance gives the coordinate of the end point (and backward azimuth but it does not matter for this model). Unfortunately, it does not work with UTM zones projection but works great with WGS 84 this is why we need first to transform the input coordinates into WGS 84 first otherwise pyproj.Geod will raise an error.
   Since a user stays (does not move a lot) we assume that she/he moves not far away from a centroid point of the location namely from 0 to 5 meters (possible_distance=random.randint(0,5)) and let's presume that a user can move in any direction (possible_forward_azimuth = random.randint(0,360))
   Before appending the data to data_array we pass the current value of time_start to time_gps it is needed because we will transform the time_start variable and want to hold its new value to the next iteration.
   At the end of each iteration we append to a data array four things: user (user_id), time_gps (time when a new pair of coordinates were created) and coordinates themselves.
   After that we add to time_start some minutes and actually this is where the creation of a new set of data begins since in the next iteration this value will be assigned to the next pair of coordinates. You can add to time_start some seconds or even use distance between start and end points (possible_distance variable) and calculate time (based on your preferred velocity). However, since the main purpose of this function is to make a user's movements look static it may be easier to add some minutes and avoid any calculation or appending rows with timestamps of second precision (saves time and memory).

The function will produce data within itself till time_start < time_end and returns the first value of time_start variable when this condition is not met. From this time value moving activity starts.
The output data of this function (if you use heatmap visualization) will look like a dense red dot since created points are located near to each other at small-sized area:

Out[36]:



Picture 1. Example of the data produced by function get_static_points, there is a dense red dot around the centroid point of the building. In this model it means a user stayed at this location (Hotel Pallas).

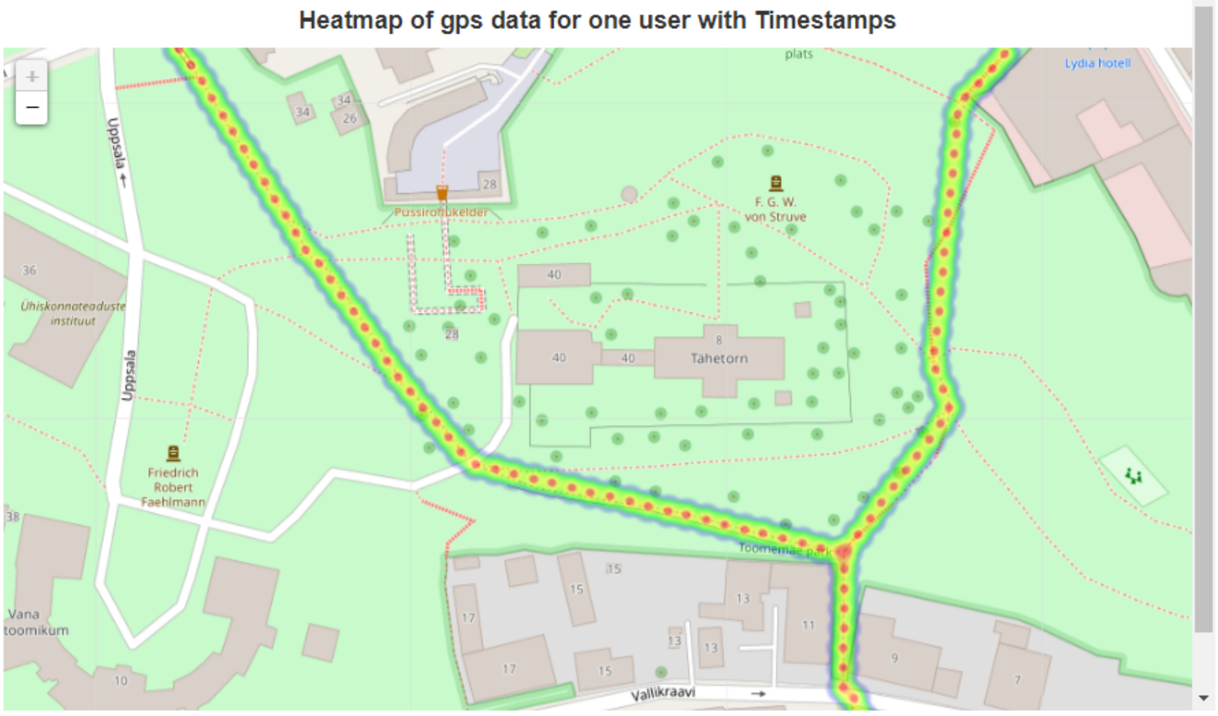
10. Next function get_points_on_path generates points on LineString between two locations (e.g. home and work, work and event1, etc.). This LineString will be called a path. The function uses a shapely interpolate function that returns a point at the specified distance along a linear geometric object (distance is calculated always from the starting point of the geometry object).
First to get these distances we employ np.linspace function that returns evenly spaced numbers over a specified interval. A start parameter will always be 0 since we want a user to start moving from the beginning of the path, the end parameter is the length of the path, num parameter (number_of_points) defines how many points we want the path to be interpolated. For example, we have path from home to work which length equals 100 meters (path.length) we want to get three points on this path (number_of_points=3) so we apply np.linspace function with such parameters: distances = np.linspace(0, 100, 3) -> the output will be: [0. 50. 100].

After we get distances, with list comprehension we interpolate our path with points that will be placed on the path at defined distances. In our example three points will be located in 0, 50, 100 meters distances from the starting point of the path respectively. So basically we have the first point in the beginning of the path, second point in the middle of the path and third point in the end of the path.
The function returns the list of these interpolated points and it should be mentioned that the length of the list = number_of_points.

The idea is not mine and maybe it will be easier to understand the content of this function with the post from Stack Overflow.

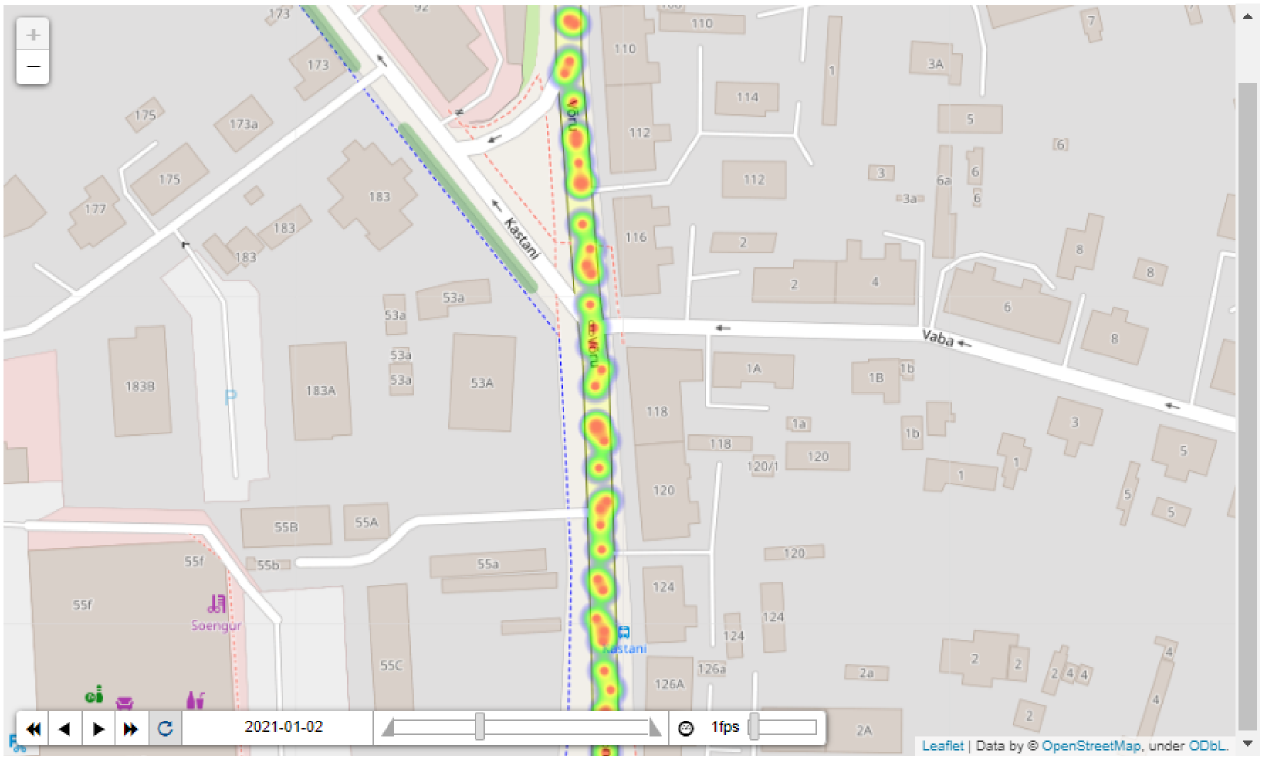
11. Previous function will get equally spaced points along the path but gps points can not be so ‘clean’ and ‘precise’ there are always some points that deviate from the path line. Function get_chaotic_point creates such points which produce one chaotic point between two interpolated points (see section 10) meaning that with very high likelihood it will not be located on the path but near to it.
Function takes three parameters: point_start, end_point, radius_of_buffer, the last parameter will be defined with the length_of_distance_m variable (see in Section 12). Firstly, it finds the intersection result of two buffers created around point_start and point_end respectively. Secondly, it generates a path between these points. Thirdly, it finds the intersection result of points_intersection with path buffer with radius equals 2 meters (you can change the radius of the path_between_points but it should be kept in mind that if the radius will be too big the chaotic point may be placed too much far from the main path and the whole movement might look too ‘clumsy’ or ‘too chaotic’).
The function returns one chaotic point within final_intersection.
The following pictures show the movement with and without get_chaotic_point:

Out[38]:


Heatmap of gps data for one user with Timestamps
Uppsala
Pussirohukelder
F. G. W. von Struve
Tähetorn
Friedrich Robert Faehlmann
Ühiskonnateaduste instituut
Vana toomikum
Toomemäe park
Vallikraavi
Lydia hotell

Picture 2. The visualization of movement from one location to another with only get_points_on_path function. The points are equally spaced and placed on the path.

Out[43]:



Picture 3. The visualization of movement from one location to another with get_points_on_path and get_chaotic_point functions. The movement looks not so

straightforward but at the same the direction and trajectory of the movement is vivid. Taking a closer look you may notice the trend: there are always two equally spaced points on path and one point between them not exactly located on LineString

12. To acquire data while a user is moving between locations, a function get_moving_points is employed. It is also one of the most important functions like get_static_points and responsible for so called moving activity (please, see Overview chapter).
To better understand function first let's go through each parameter:
G - projected graph
start_node, end_node - needed to create a path along the actual roads of the area, both can be derived using ids which are returned in function get_meaningful_locations (Section 7)
start_coords, end_coords - needed to add to the final path's coordinate list since the OSMnx library only provides the path from one node to another and nodes are just parts of roads to get a path connecting two locations we need to pass their coordinates. Both parameters can be derived using ids which are returned in function get_meaningful_locations (Section 7)
user - user_id
data_array - array within which all data is collected
time_start - the same as in get_static_points function, time when the previous activity has ended
length_of_distance_m - the length between two interpolated points on the path, keyword argument, equals 10 meters
mean_velocity_ms - since we have the velocity condition we need define mean_velocity, keyword argument, equals 1.1 ms (4 km/h)

There are three main parts of the function:
    1. Creating path
First, with built-in function distance.shortest_path we get a list of node IDs constituting the shortest path. Since nodes are stored in GeoDataFrame and we have their ids with property .loc we derive respective rows from the GeoDataFrame. Next step is to get a list of Point.geometries that create the corresponding path using their geometry column (route_list) . Then insert in the beginning of the route_list coordinates of start location and append coordinates of end locations (basically the coordinates of their centroid point). Finally, get a path passing list of points to a LineString.

    2. Interpolating the path
The main question of this part is to what value variable number_of_points should be equal to. For this purpose we need the length_of_distance_m parameter. We divide the length of the path generated from the first part by length_of_distance_m and round it up. In the current version of the script length_of_distance_m equals 10 meters. There are two reasons why. Most important - since we have velocity condition if the distance between interpolated points is too high the time needed to reach from start

point to end point will be also high (e.g. 30 seconds, if distance equals 33 m and velocity 1.1 m/s) so the time gaps of consecutive gps data rows may be quite enormous, which is not realistic when talking about gps data (usually frequency of gps tracker is even 1 second). Of course a chaotic point between two interpolated points makes the time gap between rows less drastic but nevertheless I would not recommend making length_of_distance_m too big. Another reason for the small length_of_distance_m concerning the validation process (will be discussed more in detail in Section 15). In order to check the velocity condition the distance between each pair of consecutive points will be calculated and divided by the time gap between them. The distance will be the length of the straight line between these points. Imagine a really curved path and for example between start point and end point there will be a lot of turns so while the length of straight line between these points will be low the actual distance including all the turns may be high. Respectively it can lead to miscalculation in the validation process.
There is another thing to consider while deriving a number of points. If the length of the path <= the length_of_distance_m, the number_of_points will equal 1 and in the get_points_on_path function we will get only one point in the beginning of the path. Thus the path is not created in this case we need to manually assign number_of_points to 2 (will get the start and end point of the path). This condition will probably never be used in this model since it considers movements in cities where the distance between locations are higher than 10 meters, but for example in the Neom project where distances are much less that condition will be essential. In other cases the number_of_points derived in formula mentioned above - math.ceil(path.length/length_of_distance_m). However, it should be considered that the actual distance between consecutive interpolated points will always be a little bit bigger than length_of_distance . However, with rising number_of_points the difference between actual distance of two consecutive interpolated points and length_of_distance_m will shrink to thousandths and less.
Finally when we find the number_of_points parameter with the get_points_on_path function we can produce a list of interpolated points on the path

3. Creating gps data and appending to data_array

We start looping through each point of the list of interpolated points. Right away we transform point coordinates to WGS 84 projection. Then we find a chaotic point between the current interpolated point and its next “neighbor”. After, we calculate the distance between the current interpolated point and chaotic point and since we have a velocity condition we find time dividing a distance between them by a defined mean_velocity keyword argument. As in get_static_point we first assign the current value of time_start to time_gps and then modify time_start variable adding time_to_chaotic_point
Next step - append to data_array user_id, time_gps (time when the current interpolated point was taken), coordinates of the first interpolated point. The same process is repeated but between a chaotic point and the next interpolated point. Again we assign to time_gps value of time_start (but now it equals time_start+time_to_chaotic_point). In the second step we append the coordinates of a chaotic point and time when they were created. The coordinates of the next interpolated point and its time will be appended in the next iteration of the for loop. Chaotic points are generated till the user gets to the end point of the path (after this

point there is no next point) so the function just appends the last point of the path in the else section. The condition statements distance_to_chaotic_point < 2.2 and distance_to_next_point < 2.2 area initiated because when the distance is really small between points each milliseconds in time matters however in final data frame we want timestamps with second precision but with rounding the time to seconds the final velocity maybe sometimes 7.8 km/h which is not appropriate in our model. So we mainly change the time_to_chaotic_point and time_to_next_point e.g. to 2 seconds while their actual value can be 1.342.

As a get_static_points function, the function get_moving_points also creates data within itself and returns the last value of time_start (time when a user reached the place) , which we need to pass to the get_static_function since stay activity starts from this time.

13. The final function is random_plot. This function creates random gps data with different plots (please refer to Overview chapter). Its basis is two main functions of the script - get_static_points and get_moving_points. Before analyzing randome_plot function it is better to see what happens before it.

First we define the number of users (e.g. 3), and date range. Second, create a dictionary user_locations: 1) for each user in the loop the function get_meaningful_locations generates home_id, work_id, list of regular ids and add it to the dictionary where key is user_id and value is the list of mentioned ids. Thus we have a dictionary where for each user we can derive meaningful locations by his/her id.

Next step - create an empty gps_data_array where all data will be stored. Then produce nested for loop. In the first loop for each user we will get id of nearest node, x and y coordinates of home and work locations derived by their own ids and stored these three variables in two separate lists - home_list and work_list. After that time_start variable is initiated which at the beginning will be equal to starting time of the analyzed period (but then will constantly change it in the second for loop). Then, the second for loop will go through all defined days in date_range and : 1) assigns time start to a new another_day_start variable 2) initiates day variable which always be equal to starting time of current day during which the gps data is created 3) makes a day string - needed only for GPS_PLOT text document 4) gets a day of week with built-in function .isoweekday()

As it previously described in the Overview chapter the user may visit some events, the number_of_events is defined randomly with stated likelihood. We assume that on weekdays a user surely will be at home and go to work thus the list_of locations should append both home_list and work_list (the sequence does matter). However, on weekends a user does not go to work so the list_of_location appends only the home_list. Finally the function random_plot is called.

Firstly, within the function we randomly generate event_node_id, event_x, event_y using another function get_regular_or_random_loc (please, see Section 8). We pass these variables to event_list and this list we append to list_of_locations. The number of event lists is equal to the number of events. The length of list_of_locations equals to number_of_events + 1 (just home_list) or 2 (home_list and work_list)

Then three possible startings of plot are called based on condition:

- if it is weekend and number_of_events != 0 the user stays at home between 10-14 p.m.
- elif (or) it is weekday the user stays at home between 7-9 p.m., goes to work and remains there till 17 -19
- if none of mentioned conditions are not met it means it is weekend and the user stay at all day and the function will produce only static points within home location till 22 p.m. - 2 a.m and return the final value of time which first assign to time_start and then to another_day_start variable when next iteration (day) starts

In terms of the first two startings of the plot the user will surely go home soon or later (last if section in the function) and then the time of reaching home will be returned and used as a start time of another day.

However, if the first starting of the plot (weekends) is called it means that a user will surely go to some events. Since all movement in the model can be presented as a consecutive process of replacing moving and stay activity with each other we create a while loop (can be called an event loop) that will iterate and produce activities between events till it reaches the last location in list_of_locations because from the last location user goes to home.

In case of the second starting of the plot a user can either go straight to home or initially proceed to the event loop and then go home, this depends on the number_of_events variable.

14. The visualization part is done using folium library and HeatMapWithTime plugin. The main goal is to see how the synthetic gps data for some random user looks like. Also using GPS_PLOTS text document you can actually check if a user for example visited all defined locations in each day
15. The validation part is done using three functions. The main goal is to check if the velocity condition is satisfied. The function getDistanceFromLatLonInKm calculates the distance between consecutive points that are in WGS 84 projection (please, refer to the post from Stack OverFlow). The function calc_velocity is self-explainable. The function get_dustance_and_velocity takes the gps_data DataFrame and modifies it adding new columns - lon_end, lat_end, timestamps_end, dist_km, velocity_kmh, the last two are that used for velocity condition evaluation. Make sure that before passing gps_data DataFrame to the function get_dustance_and_velocity you sort it first by user and then by timestamp.

Evaluation of the script:

Advantages:

- The model finally looks like gps with highly noticeable trajectories
- User gets home, work, regular locations that are mostly meaningful
- Velocity condition is met

- The process is random and automated but at the same time follow real life plot
- The model is flexible and there are many parameters that can be tuned

Disadvantages:

- The path between the location and its nearest node can intersect buildings or water bodies (lakes). A user goes through them, it is not realistic
- While moving users go in one direction but they may turn back
- No gps like errors in the data (e.g. sudden termination of ‘acquiring’ data, or some points that do not make sense)

Challenges (points to discuss):

- Plot different movements to other social groups (e.g. children)
- The event locations can be tuned using tags in OSM based on social background of users
- Possible reduction of length_of_distance_m variable to 2 meters to make time frequency of the data close to one second. However, time to produce data may take much longer also velocity condition should be checked again
- Implement different kinds of modes
- Load testing

If you have any questions or suggestions do not hesitate to contact me in person or in Slack/by email (nikolajkozlovskij73@gmail.com)