Introduction

The assigned project comprises two primary objectives: firstly, identifying the points within the museum where individuals come to a halt, based on shapefile data delineating their trajectories; secondly, comparing these halt points with the stop times of other trajectories to ascertain if individuals were in close proximity. Prior to commencing the initial task, it was decided to utilize QGIS for visualizing the trajectories of individuals, as well as the layout of tables and exhibitions within the museum. This facilitated a foundational understanding of the representation of such data. To achieve this visualization, data importation into QGIS can be executed through three methods: direct layer importation, utilization of a Python function (via the built-in plugin), or incorporation of a plugin.

```
fT_68 = "../Data/Trajectories/person_68.shp"
layer_68 = iface.addVectorLayer(fT_68,'','')
fT_67 = "../Data/Trajectories/person_67.shp"
layer_67 = iface.addVectorLayer(fT_67,'','')
fT_57 = "../Data/Trajectories/person_57.shp"
layer_57 = iface.addVectorLayer(fT_57,'','')
```

- load it from a postgres DB after it has been connected to QGis via PostGis.

```
def load_table(uri, table_name, geometry_col):
    uri.setDataSource("public", table_name, geometry_col)
    vlayer = QgsVectorLayer(uri.uri(False), table_name, "postgres")
    QgsProject.instance().addMapLayer(vlayer)

# Database settings
db_settings = {
    "host": "localhost",
    "port": "5432",
    "database": "project_db",
    "user": "postgres",
    "password": "****",
}

geometry_col = "geom"
tables = ["outside_57", "outside_67", "outside_68", "person_57", "person_67", "person_68"]

# Construct the connection URI
uri = QgsDataSourceUri()
uri.setConnection(**db_settings)

# Load tables
for table in tables:
    load_table(uri, table, geometry_col)
```

Upon loading the trajectories of three individuals, tables, and exhibitions, the following outcomes were observed: Each trajectory layer is color-coded, enabling clear differentiation between individuals.

Task One

Analyze the data and creating new layer

After visualizing all the data, we observed that certain parts of the trajectories intersected with the tables and exhibitions. These points were identified as noise, likely artifacts obtained during the recovery of geospatial data. In order to improve the data quality, we opted to clean the data prior to analysis to assess its impact on the results compared to the original dataset.

To accomplish this, we imported our trajectories into PostgreSQL as tables, allowing us to manipulate each trajectory as a separate table. We determined that the most effective method for reducing noise involved removing all records from the trajectories that fell within the boundaries of the tables. Initially, we created another table named 'outside' by duplicating the relevant table 'person' to ensure that both tables shared identical attributes. Below is the query utilized for this operation:

```
-- Clone a table structure(column name and type) but no data
SELECT *
INTO outside_57
FROM person_57
WHERE 1=0;
```

After creating the table, we proceeded to insert all the points of the trajectory into it as follows:

```
-- insert in outside all the data from person
INSERT INTO outside_57
SELECT  gid,timestamp,geom
FROM person_57;
```

Next, we utilized the PostGIS function ST_contains() to identify all the points that were located inside the tables within the museum. These points were then saved into a table named 'inside', which shared the same structure as the 'person' and 'outside' tables.

```
-- Get point inside of objet in the trajectory
INSERT INTO inside_57(gid,geom,timestamp)
SELECT person_57.gid,person_57.geom, person_57.timestamp
FROM tables,person_57
WHERE ST_Contains(tables.geom,person_57.geom);
```

Following the definition of the ST_contains function, which determines if one geometry is completely inside another, we proceeded to remove from the 'outside' table all rows that were identical to those in the 'inside' table. This was accomplished using the following query:

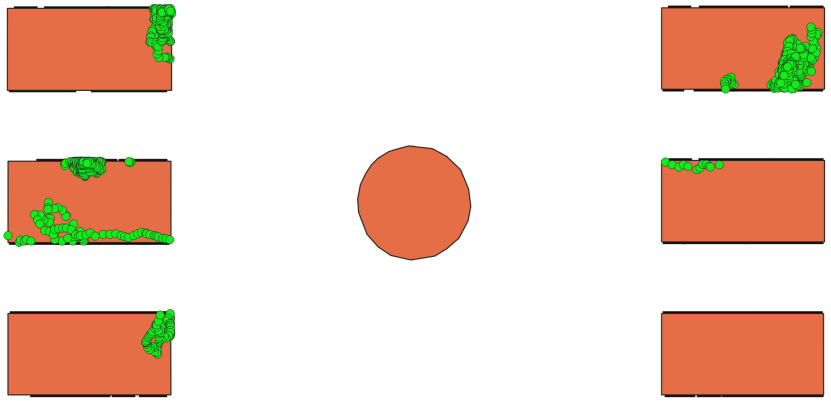
```
-- Delete extra data
delete from outside_68
where exists(select *
             from inside_68
             where outside_68.gid = inside_68.gid);
```

Now, we have divided the 'person' layer into two new layers: 'inside' and 'outside'. The latter could be useful for comparison with the non-filtered layer 'person'.

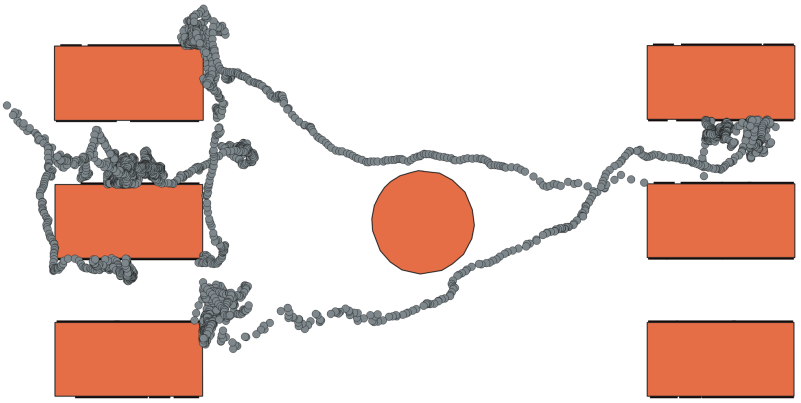
The following images represent the three layers we just discussed using the dataset 'person_67':

The original layer 'person_67':

The inside the tables layer ‘Inside_67’:



And the noise-free layer ’outside_67’:



We also repeated this process with our other 2 dataset ‘person_57’ and ‘person_68’.

StayPoint Detection Algorithm

A stay point refers to a geographic region where an individual has remained for a certain period of time. This algorithm serves as a reference to the paper titled "Mining Individual Life Pattern Based on Location History" by Y. Ye et al. (IEEE MDM 2009).

For the implementation of this algorithm, we utilized PyQGIS, which interfaces with various QGIS libraries and Python. Specifically, we imported the datetime library to convert the timestamp attribute column into the standard datetime format, and accessed the QGIS environment using qgis.utils and qgis.core. Since the timestamp column in our datasets was in varchar format, it was necessary to convert it to the datetime format in Python to facilitate comparison between timestamps. The function calculateTimeSpan(QgsFeature, QgsFeature) takes two features as input and casts their timestamp attributes into datetime format. The time span interval between the two features' points is then calculated through subtraction, yielding the only output of the function.

```
# Constants (Adjust as needed)
TIMESTAMP_FORMAT = '%Y/%m/%d %H:%M:%S.%f'

# ------------------ Helper Functions ------------------
def calculate_time_span(feature_i, feature_j):
    time_i = datetime.strptime(feature_i.attributes()[0], TIMESTAMP_FORMAT)
    time_j = datetime.strptime(feature_j.attributes()[0], TIMESTAMP_FORMAT)
    return time_j - time_i
```

We call this function wherever we need to compute the time difference between two points and compare it with a specified time threshold. The function meanCoord(QgsFeature: QgsPoint, QgsFeature: QgsPoint) takes two geometry features as input and calculates the mean coordinates between them by averaging their latitude and longitude values. The output of the function is a QgsPoint representing the mean coordinates.

```
def calculate_mean_coord(feature_i, feature_j):
    geom_i = feature_i.geometry().asPoint()
    geom_j = feature_j.geometry().asPoint()
    return QgsPoint((geom_i.x() + geom_j.x()) / 2, (geom_i.y() + geom_j.y()) / 2)
```

Additionally, we utilize this function when determining two points as a stay point. Lastly, we proceed with the implementation of the stayPointDetection algorithm.

Algorithm StayPoint_Detection(*P*, *distThreh*, *timeThreh*)

Input: A GPS log *P*, a distance threshold *distThreh*
and time span threshold *timeThreh*
Output: A set of stay points $SP=\{S\}$

```
1. i=0, pointNum = |P|; //the number of GPS points
2. while i < pointNum do,
3.     j:=i+1; Token:=0;
4.     while j < pointNum do,
5.         dist:=Distance(pi, pj);  //calculate the distance between points
6.         if dist > distThreh then
7.             ΔT:=pj.T-pi.T;   //calculate the time span between two points
8.             if ΔT>timeThreh then
9.                 S.coord:=ComputMeanCoord({pk | i<=k<=j})
10                S.arvT:= pi.T;  S.levT=pj.T ;
11.               SP.insert(S);
12.               i:=j; Token:=1;
13.            break;
14.        j:=j+1;
15.    if Token!=1  then i:=i+1;
16. return SP.
```

Fig. 3 Stay points detection Algorithm

The algorithm referenced is from the paper: [Y. Ye, Y. Zheng, Y. Chen, J. Feng, and X. Xie, "Mining Individual Life Pattern Based on Location History," 2009 Tenth International Conference on Mobile Data Management: Systems, Services and Middleware, 2009, pp. 1-10, doi: 10.1109/MDM.2009.11.]

The algorithm function is invoked with three inputs: the trajectory data, a distance threshold, and a time threshold. For each individual, the algorithm stores three pieces of information: the arrival time, the departure time, and the mean coordinate between two points to identify the stay point region.

```
stay_points = stay_point_detection(trajectory_layer, dist_thresh, time_thresh)
```

In the implementation, we first access the QGIS layers using PyQGIS and store the layer in a variable defined as p. We initialize a list called sp, which will return the result of the set of stay points {S}. To count the number of points, which are essentially features, we utilize a PyQGIS method to count them.

Next, the iteration over the geometry points begins to find the distance between two sequential points. To calculate the distance, we first access the features of the trajectories, which are points. Then, we use the

geometry() method of PyQGIS to retrieve the geom attribute of each feature. Finally, we use the distance method to determine the distance between two geometries.

```
# ------------------ Main Algorithm ------------------
def stay_point_detection(layer_name, distance_threshold, time_threshold):
    layer = QgsProject.instance().mapLayersByName(layer_name)[0]
    stay_points = []

    features = layer.getFeatures()
    for i, feature_i in enumerate(features):
        for feature_j in itertools.islice(features, i + 1, None):
            distance = feature_i.geometry().distance(feature_j.geometry())
            if distance > distance_threshold:
                time_span = calculate_time_span(feature_i, feature_j)
                if time_span.seconds > time_threshold:
                    mean_coord = calculate_mean_coord(feature_i, feature_j)
                    arrival_time = feature_i.attributes()[0]
                    departure_time = feature_j.attributes()[0]
                    stay_points.append((mean_coord, arrival_time, departure_time))  # Store as tuple
                    break

    return stay_points
```

After computing the distance between two feature points, we proceed to check whether this distance exceeds the given distance threshold. If the distance is greater than the specified threshold, we calculate the delta time to determine the time span interval between the two points. This calculation is achieved by invoking the previously mentioned function, calculateTimeSpan, which returns the delta time as the subtraction of the two points in the datetime library format. We access the seconds property and explicitly cast it as the delta time to a float type.

Additionally, we compare the time span interval with the given time threshold to determine if the delta time is less than or greater than the time threshold. In the latter case, the mean coordinate of the two points is calculated using the meanCoord() function mentioned earlier. The resulting QgsPoint is then stored in a variable along with the arrival time and departure time.

If a stay point is detected, the token is set to 1 to control the flow and skip further iterations, starting from the previous point. In case of successive failures to meet either the distance or time span interval conditions, the internal loops break. If the condition for the token is not met, the algorithm continues iterating to the next point to check.

These operations are repeated for the entire list of points until the list is exhausted, and then the list of stay points is returned. For example, the function is called with the following threshold values: distance threshold = 1.2 meters, time threshold = 2 minutes.

After this, we inspect the list to determine if any stay points were found. If any stay points are identified, their geometry (without timestamps) is extracted, and a GeometryLayer is created for each geometry of a stay point found in the list. These layers are then added to the graphical display, showcasing all the identified stay point areas.

```
# ------------------ Usage Example ------------------
trajectory_layer = "person_67"  # Replace with your layer name
dist_thresh = 1.2
time_thresh = 30

stay_points = stay_point_detection(trajectory_layer, dist_thresh, time_thresh)

if not stay_points:  # Check if any stay points were found
    print("No stay point detected ")
    print(datetime.now())
else :
    # Display stay points on QGIS
    for geometry, arrival_time, departure_time in stay_points:
        geometry_layer = QgsVectorLayer(f"?query=SELECT ST_GeomFromText('{geometry.asWkt()}')",
                                        f"trajectory {trajectory_layer}, dist_thresh {dist_thresh} , time_thresh {time_thresh}",
                                        "virtual")
        geometry_layer.renderer().symbol().setSize(6)
        QgsProject.instance().addMapLayer(geometry_layer)
        print(datetime.now())
```

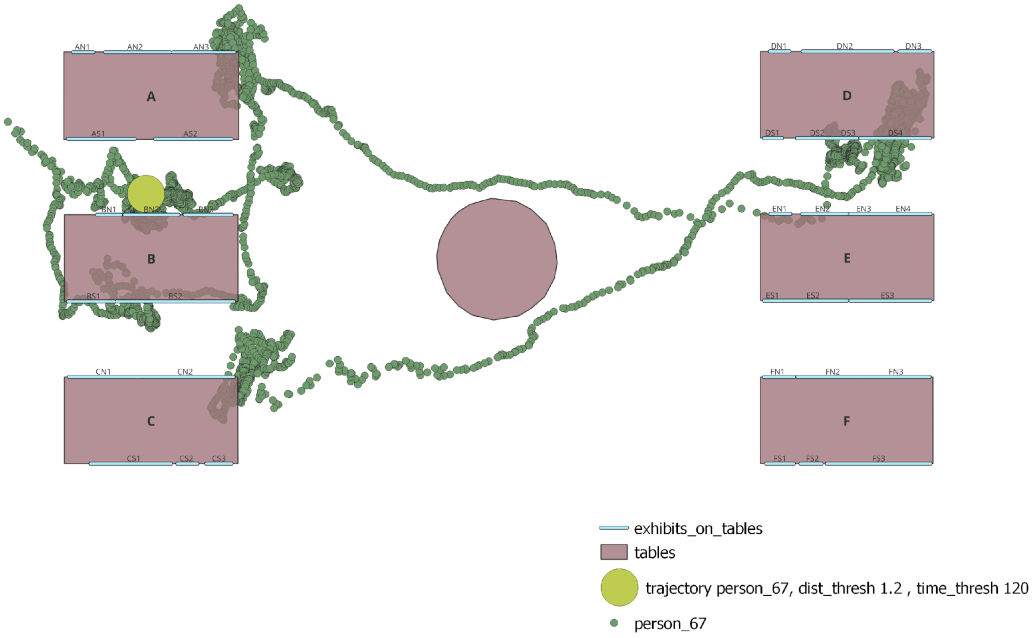
As an example, the following image illustrates the graphical outcomes generated by running our algorithm on the datasets 'person_57', 'person_67', and 'person_68', with the designated threshold values: distance threshold = 1.2 meters, time threshold = 2 minutes.

Person_57

sp = [<QgsPoint: Point (3.3881540205120082 8.78701860300969528)>, 2021-12-13 12:08:55.352000, 2021-12-13 12:14:23.198000]

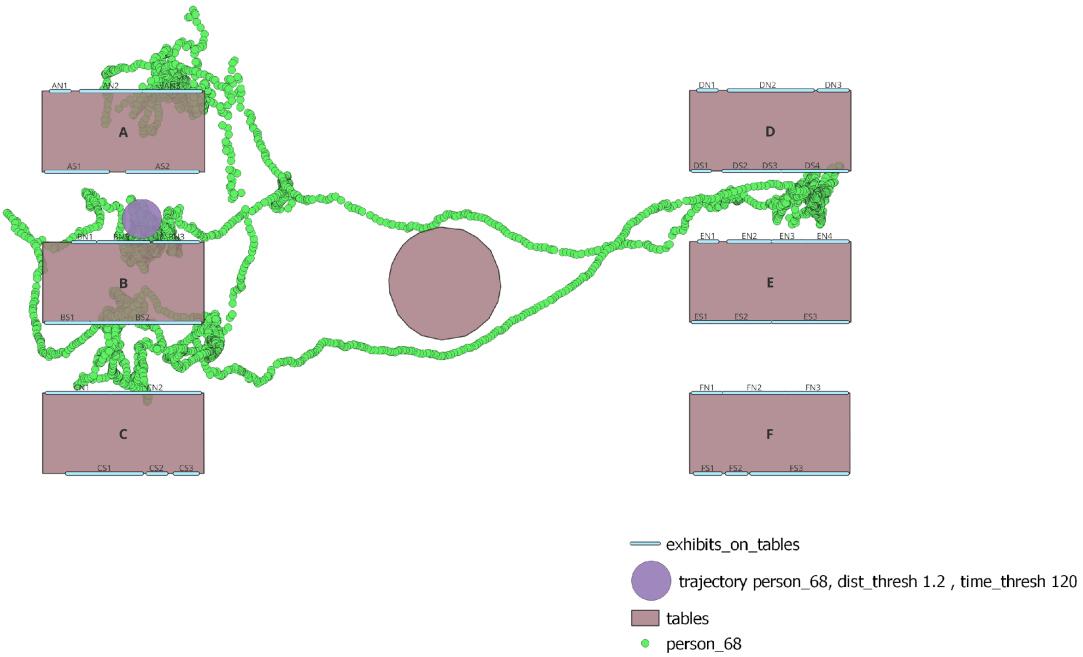
Person_67

sp = [<QgsPoint: Point (3.92595060325195444 8.62342855146957987)>, 2021-12-13 12:09:09.427000, 2021-12-13 12:14:21.277000]

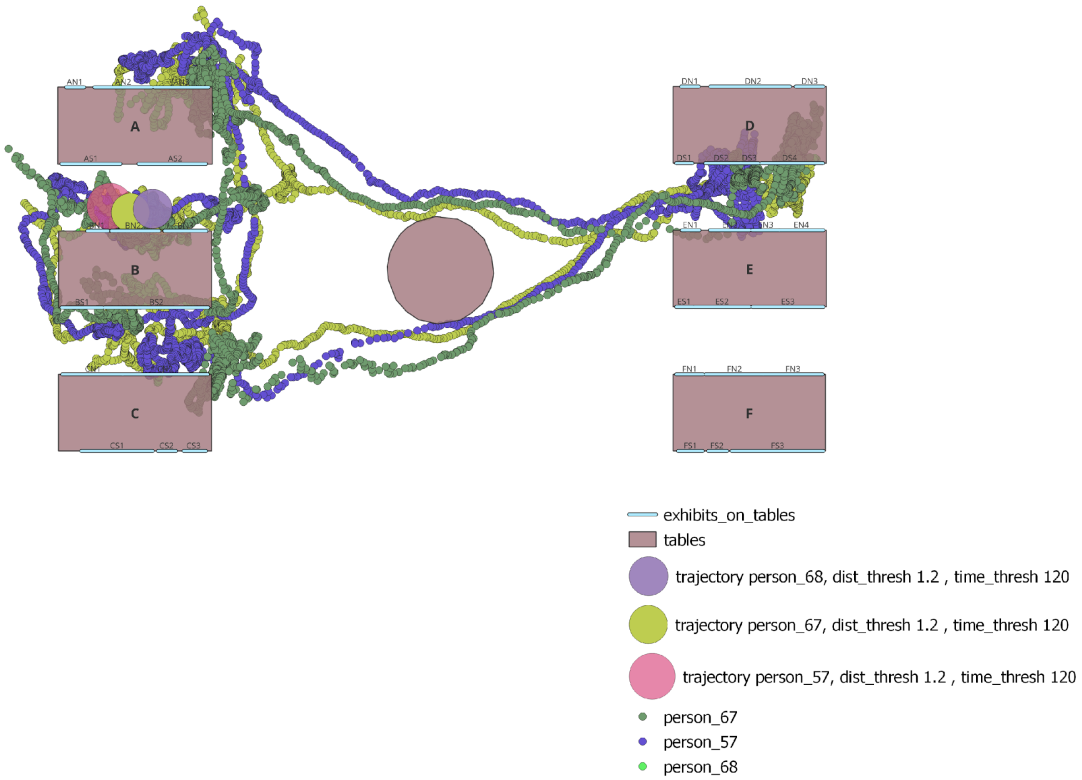


Person_68

sp = [<QgsPoint: Point (4.50414626777582505 8.71712140294116544)>, 2021-12-13 12:09:00.793000, 2021-12-13 12:14:25.462000]



Below, the figure depicts all of the datasets together:



As observed, the algorithm with these parameters has identified only one stay point for each of our datasets. Interestingly, in every dataset, these stay points are located in nearly the same position.

Task Two

Recap on stayPoint_Detection Algorithm

As previously demonstrated, upon calling the stayPoint_Detection algorithm, it returns a list of stay points, each containing their geometries, arrival time, and departure time.

SP = [Point, SarvT , SlevT , Point, SarvT, SlevT, ...]

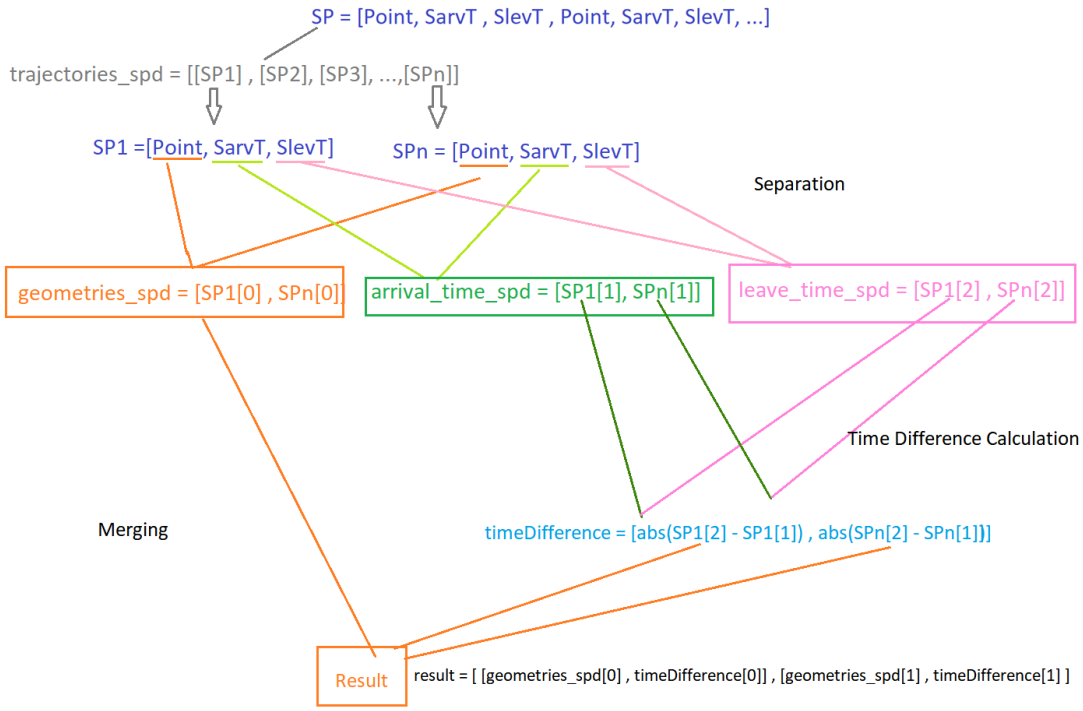
Using this approach, we can identify stay points that have identical arrival and departure times.

groupSPDBasedOnTime

We introduce the function groupSPDBasedOnTime(layerNames: List[str], dist_Thresh: float, time_Thresh: float, stayTimeThresh: float). The first parameter takes a list of strings representing the names of layers. The second parameter, similar to the one in the SPD algorithm, determines a distance-based threshold. The third parameter defines a time threshold for the trajectories. The last parameter, stayTimeThresh, sets a threshold for the time difference between trajectories to consider them as staying together. The function returns a list of point geometries in the QGIS point class type. If no points are detected with the given thresholds, the function returns 1 as the output.

Deeper on groupSPDBasedOnTime

As depicted in the figure below, after obtaining the stay points from the SPD algorithm, the result is stored as a 2D list. Subsequently, the algorithm iterates over this list, separating the geometry, arrival time, and departure time, and inserts them into another list. Next, the algorithm calculates the time differences using delta time and compares them with the specified threshold. The time differences are then merged with the corresponding geometry.



To call the SPD algorithm, we need to import it. Essentially, we import the SPD algorithm.

```
from datetime import datetime, timedelta
from qgis.core import *
import qgis.utils
import itertools

from spd import *
```

Next, we need to compute SPD for each layer and store each resulting list in a new list.

```
def group_spd_based_on_time(layers , distThresh, timeThresh,stay_time_thresh):
    # For each layer the SPD is calculated and store in a list
    trajectories_spd = []
    for i in range(0,len(layers)):
        trajectories_spd.append(stay_point_detection(layers[i],dist_thresh,time_thresh))

    # If not find any SPD
    if len(trajectories_spd) == 0:
        return 1
```

The exception here is that if one or fewer trajectories are found, it returns 1 to halt the execution. Following that, the separation of the SPD information begins, with each piece of information stored in a separate list.

```
    # separate geometries arrival time and leave time
    # Initialize variables
    geometries_spd = []
    arrival_time_spd = []
    leave_time_spd = []


    # Iterates over the trajectories_spd list and put each category in
    # its correspond list
    for i in range(0,len(trajectories_spd)):
        geometries_spd.append(trajectories_spd[i][0])
        for j in range(0,len(trajectories_spd)):
            if isinstance(trajectories_spd[i][j],datetime):
                arrival_time_spd.append(trajectories_spd[i][j])
                break
        leave_time_spd.append(trajectories_spd[i][j+1])
```

Similar to the implementation in the SPD algorithm, the delta time, which is the subtraction between the departure time and arrival time, is calculated. The resulting values are then stored in a new list. The calculated result is stored as a datetime.timedelta object from the Python datetime library.

```
    # Compute the delta time in seconds for each trajectory
    # leave_time_spd - arrival_time_spd in corresponding list items
    delta_times = []
    for i in range(0,len(arrival_time_spd)):
        # The result will be a datetime.deltatime in python date time
        delta_times.append(leave_time_spd[i] -arrival_time_spd[i])
```

To compute it based on seconds, access the seconds property of the delta time, and insert these values into a list.

```
# We need to get the property 'seconds' of deltatimes
delta_times_seconds = []
for i in range(0,len(delta_times)):
    delta_times_seconds.append(delta_times[i].seconds)
```

In this step, the time differences for each trajectory, which have been stored in a list in seconds, need to be computed. To accomplish this, we utilize the itertools module in Python and the Cartesian product to calculate the time differences in seconds.

```
# For each trajectory we need to compute the difference time between them
timeDifference = []
counter = 0
sum = 0

# The algorithm bellow illustrates how we can compute the difference of time between trajectories
# Use cartesian product to subtract the time of each trajectory with the others in the list
# Store a result of each time difference of trajectory with others in new list
for p1,p2 in itertools.product(delta_times_seconds,repeat = 2):
    if counter == len(delta_times_seconds):
        counter = 0
    if counter < len(delta_times_seconds):
        sum = sum + abs(p1 - p2)
        counter = counter + 1
        if counter == len(delta_times_seconds):
            timeDifference.append(sum)
            sum = 0
```

In the next step, the geometry list is merged with the corresponding indices of the time differences list. To achieve this, we define a function called "merge" and call it to merge the two lists.

```
# Merge the correspond geometries with its time difference
# Call the merge function that the output will be a merged list
# The result will be a 2D List with the structure
# [[geom object, time difference],[...],[...],[...],...,[...]]
merge_g_t = []
merge_g_t = merge(geometries_spd , timeDifference)

# Extract the points with less than the stay time threshold
result = []
```

At the end of the function, the extraction of the geometries within the stay point threshold parameter is applied. The corresponding geometries are then stored in a result list, which is returned as the output of the function.

```
    # Extract the points with less than the stay time threshold
    result = []

    for i in range(0,len(merge_g_t)):
        for j in range(0,len(merge_g_t[i])):
            if (isinstance(merge_g_t[i][j],int)==True and merge_g_t[i][j] <= stay_time_thresh):
                    result.append(merge_g_t[i][j-1])
                    break

    return result
```

To execute the algorithm, the inputs are stored in a variable and passed to the function. The result of the algorithm is then stored in a list.

```
# Preliminaries to run the algorithm
trajectory_layers = ["person_57", "person_67", "person_68"]
dist_thresh = 1.2
time_thresh = 2 * 60
stay_time_thresh = 35
stp = []
stp = groupSPDBasedOnTime(trajectory_layers ,dist_thresh , time_thresh, stay_time_thresh )
```

Then we iterate over the result list to add the geometry points as layers in the QGIS interface using Python and PyQGIS methods.

```
# Check the result and show them as a layer in QGIS
if len(stp) <= 0:
    print("No stay point detected ")
else :
    # Extract only geometries, NOT timestamps and show them on QGIS
    for geometry in stp:

            geometry_Layer = QgsVectorLayer(f"?query=SELECT ST_GeomFromText('{geometry.asWkt()}')",
            f"trajectory {trajectory_layers}, dist_thresh {dist_thresh} ,
             time_thresh {time_thresh}, stay_time_thresh = {stay_time_thresh}", "virtual")
            geometry_Layer.renderer().symbol().setSize(6)
            QgsProject.instance().addMapLayer(geometry_Layer)
```

Conclusion

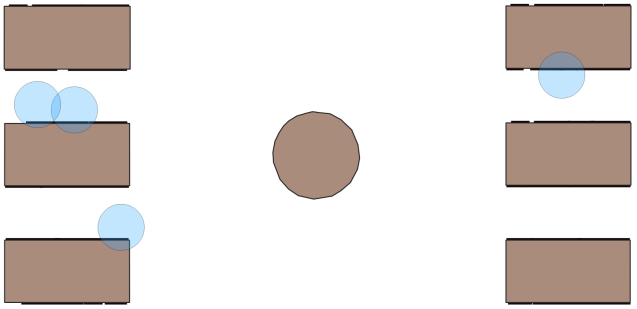
Quest One

In conclusion, regarding the first task, we observed that the original thresholds we initially set were too restrictive for our dataset, resulting in only one stay point being identified for each trajectory. To address this, we progressively lowered the thresholds until we obtained more than one stay point.

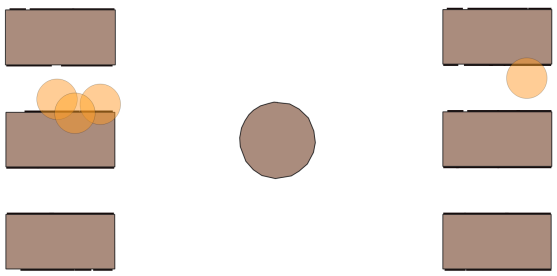
We found that the critical threshold adjustment occurred when we reduced the time threshold to below 50 seconds while maintaining a space distance of around 1.2 meters. However, after further experimentation, we determined that the optimal thresholds for achieving better algorithm performance are 1.2 meters for the space threshold and 30 seconds for the time threshold.

Below is a representation of the stop points with the aforementioned thresholds:

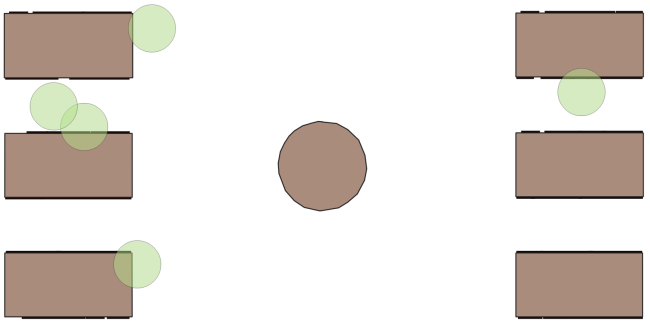
Person_57



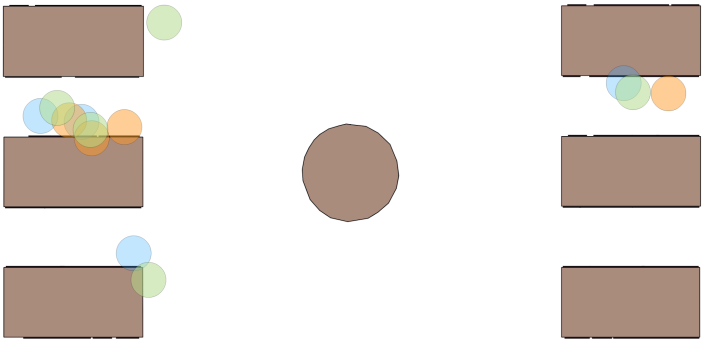
Person_68



Person_67

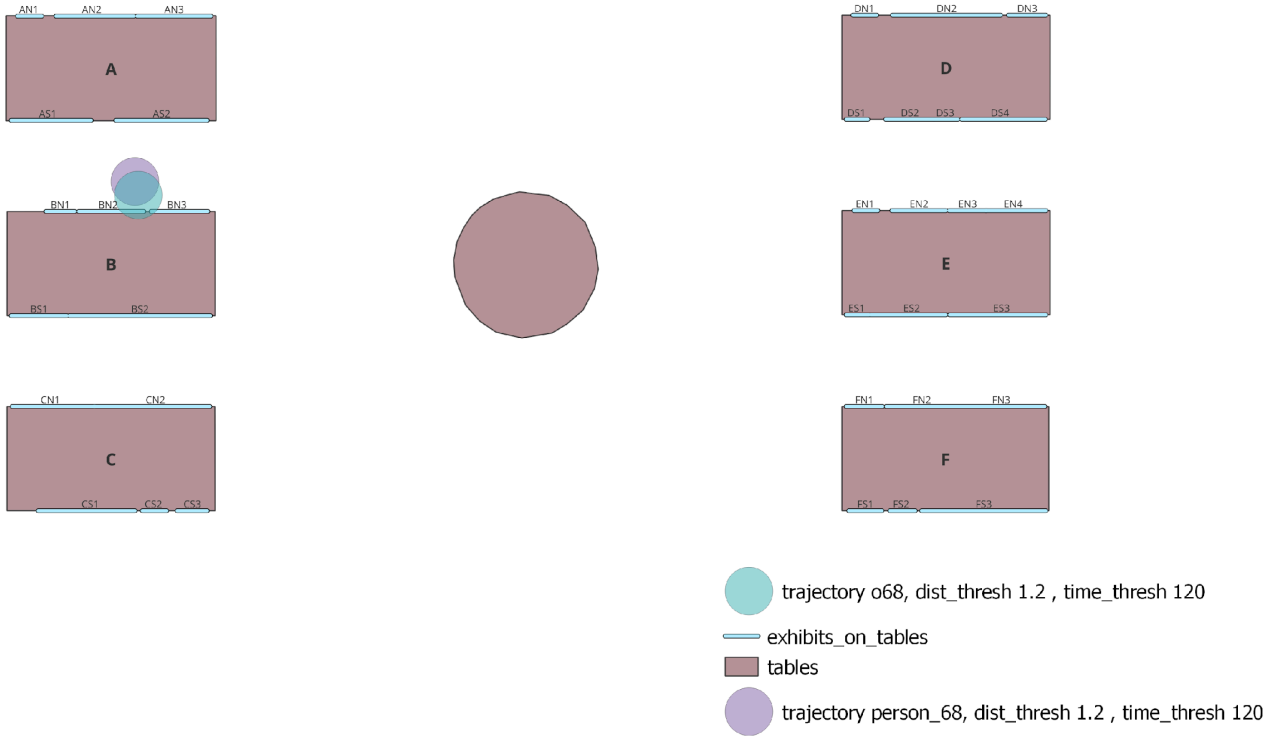


All together:



As we can observe, with these new thresholds, the algorithm is more effective and performs its task better.

Additionally, we tested the noise-filtered data against the non-filtered data to examine how the representation of the stop points would differ. In this example, we compared the stop points of 'person_68' and its noise-filtered version, 'o68', both with a time threshold of 2 minutes and a space threshold of 1.2 meters. As depicted in the image, the stop point of the filtered data is not in the same position, demonstrating how noise can affect the data representation.



Quest Two

After running the algorithm GroupSPDBasedOnTime() with the thresholds:

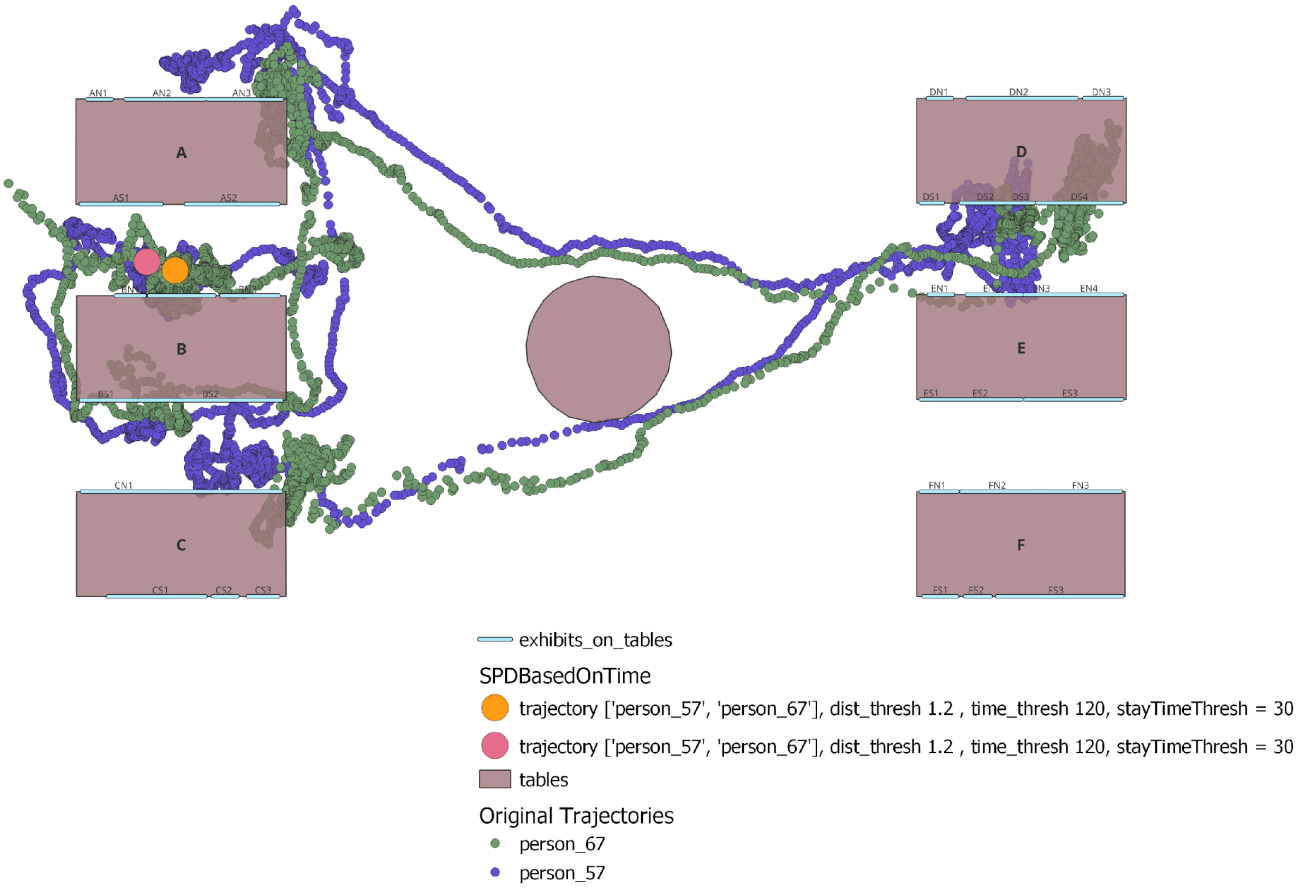
trajectory = ['person_57', 'person_67'],

dist_thresh = 1.2,

time_thresh = 120,

stayTimeThresh = 30

This means that the algorithm compared the trajectories of 'person_57' and 'person_67', seeking the stop points for each trajectory with a distance threshold of 1.2 meters and a stop time threshold of 2 minutes. It determined whether some stop points shared the same timestamp and geometry within a time window of 30 seconds (as defined by the 'stayTimeThresh' parameter), and here are the results:



The orange and pink points represent instances where the two people stopped together. They do not need to arrive and leave at the same time to be considered together; the 'stayTimeThresh' parameter is used to handle such discrepancies. Meanwhile, the small green and blue dots depict the trajectories of the two individuals, with each dot representing their movement inside the museum.