Queuing at a revolving door

Identifying queues in trajectory data using speed profiles

by

Daniël Rijnders

Student number:	4528786	
Project duration:	September 4, 2023 – October 23, 2023	
Faculty:	Civil Engineering and Geosciences	
Department:	Transport & Planning	
Thesis committee:	Yufei Yuan	Supervisor / Examiner
	Shadi Sharif	Supervisor / Examiner
	Kuldeep Kavta	Supervisor

TU Delft

Preface

This report is the Bachelor Final Project (course code CTB3000) for the bachelor program of Civil Engineering at Delft University of Technology. The general topic was offered by Winnie Daamen from the department of Transport & Planning at the TU Delft, who was interested in what could be learned about queues at the entrance of the Industrial Design faculty from trajectory data generated by a smart-sensor.

For those interested in the methodological approach to detect a queue using the speed profile of an individual object, the method used in this report can be found in chapter 4 and the results with verification in chapter 5. The specifics of the Python code used in this report to process the trajectory data and perform the queue detection can be found in appendix A .

A special thanks is given to Yufei Yuan, Kuldeep Kavta and Shadi Sharif for their continuous guidance during the writing of this report. I would also like to thank the other students of the same supervision group: Tieme van Hijum, Antoon Poelmans and Arend-Jan Timmermans. Their combined efforts during the weekly meetings with discussions, ideas and peer-reviewing greatly improved the quality of this report. Lastly I would like to thank Winnie Daamen for providing the topic and necessary data.

Daniël Rijnders
Delft, October 2023

Summary

Background
No explicit studies have been performed for the queue type specific for a revolving door. Smart-sensors providing trajectory data can be used to study this queue type. Before this can be studied, one has to find the moments of queue formation from the provided trajectory data. Various methods are used in previous studies but still without a definitive answer and often computationally demanding. Speed has not been directly used to identify or describe a queue, but showed interesting properties for this queue type.

Goal
This report investigates how queues can be identified using the speed and direction information from trajectory data from the smart smart-sensor at the revolving door at the entrance of the Industrial Design faculty.

Method
A specific 'stop and go' motion was observed for objects in a queue, where objects first slow down (1), then speed up to advance towards the entrance after other objects entered the door (2), and then slow down again as there was not enough room or time to enter the door (3). This was used to formulate a definition of being in a queue before entering the door. Each phase (1 to 3) was assigned to a time interval based on the time between door openings. An object complying with all three interval requirements was defined as a queue. A typical example of such an object is shown in Figure 1.

Results
Out of 69013 objects, 1109 were identified as being in a queue. Four of these were false positives caused by unexpected pedestrian behaviour and could not have been avoided using this method. A 100 objects unidentified out of 3846 viable objects were visually checked using animations, showing 4 missed queues, with 3 coincidentally of the same instance due to a long door delay.

Conclusion
The method proves a useful tool to quickly identify queues in large sets of trajectory data with low computational demand and fair reliability. Fine-tuning of the used parameters could improve the results. Further study is needed to check viability during other circumstances.

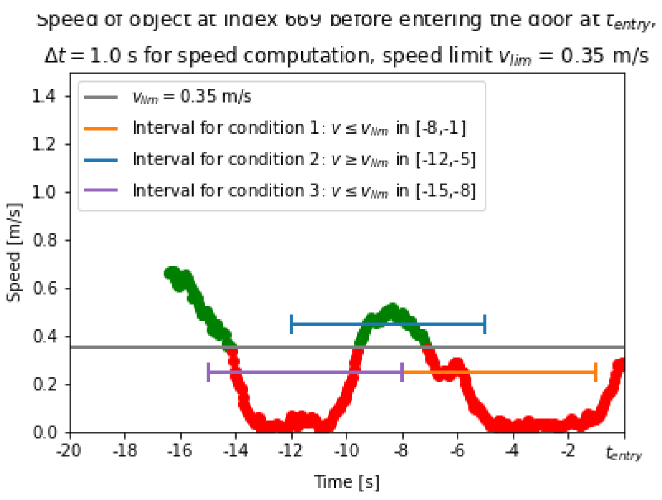


Figure 1: Typical 'stop and go' speed profile of an object in a queue

Contents

1 Introduction

Revolving doors are one of many different types of infrastructure used to facilitate and control access between partitioned areas. The capacity of such doors are determined by many factors and queuing will occur when demand exceeds maximum capacity. For the interest in queuing behaviour, a smart-sensor capable of object tracking was placed at the revolving door at the entrance of the Industrial Design faculty on the TU Delft campus (Figure 1.1). The most common way to identify and describe queues found in literature is using pedestrian density, but even though numerous ways have been proposed to quantify it, a complete answer is still missing and solutions rely on situation specific assumptions (Duives et al., 2015). Furthermore, pedestrians lining up in front of a revolving door due to unfortunate timing would be indistinguishable from a queue based on a capacity limit using this method. A study by Kneidl (2016) also reports that no explicit study has been performed concerning the specific queue type matching the situation at the entrance door.

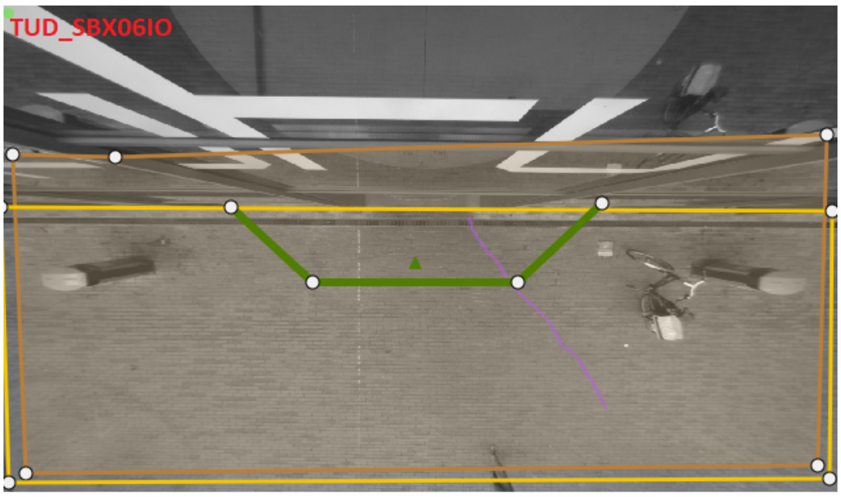


Figure 1.1: Detection area (orange) of the smart-sensor in front of the Industrial Design faculty (Daamen, 2023)

The tracking data provided by the smart-sensor can be used for countless fields of research and practical applications. Movement patterns and queuing behaviour resulting in specific queue shapes can be studied, as well as the factors that influence them. Not only during regular use, but also for safety concerns in extreme situations. Studies in this field can help to create measures to improve waiting times and door capacity, but also to increase safety during these extreme situations. Research can be expanded to compare different types of doors, even as far as to improve sustainability, for example by using revolving doors that prevent unwanted ventilation for situations found otherwise unfit.

Before queuing behaviour and shapes can be investigated, a method has to be created to identify queues in the smart-sensor tracking information data-set. A specific characteristic movement pattern during queue formation was observed at the entrance door that was not found to be used in literature for queue identification. As speed has not yet been used explicitly to identify queues, the following research questions was formed:

How can queues at the revolving door of the Industrial Design faculty entrance be identified using the speed profile of objects from smart-sensor trajectory data?

This is answered using the following sub-questions:

- What is currently known in literature about queue identification?
- What information does the smart-sensor data provide?
- How can a queue be defined based on the speed profile of objects?
- What are the required steps to identify queues in the smart-sensor data using this definition?
- How can the identified queues be verified?

The smart-sensor tracking data is provided by external expert Winnie Daamen from the department of Transport & Planning at the TU Delft. All required coding to process and visualise the information in the smart-sensor data is done using Python. To reduce the data-set into an appropriate size for this report, the workdays of the first two weeks of the academic year 2023/2024 at TU Delft are used as data-set samples. The study only examines normal every day use of the revolving door, extreme situations like evacuations are not included. Computation time considerations are not included in this report.

To answer the research question, a literature study was done in chapter 2. A description of the smart-sensor data-set is given in chapter 3. In chapter 4, the used methodology for this report is worked out. The results are shown in chapter 5 and discussed in chapter 6. The conclusions are given in chapter 7. Finally, recommendations concerning further research that were encountered in the creation of this report are given in chapter 8.

2
Literature study

The literature study was performed to find relevant information regarding queues and object trace data. The information was used for the definition of a queue in this report and to find the variables that can be used to describe one. It was found that the standard work by Fruin (1971) was cited often, even in very recent literature such as by van den Heuvel (2022). A review by Buchmüller and Weidmann (2006) shows that the studies performed since then are still relevant as their results have consistently been confirmed by more recent studies and still serve as the basis for currently applied techniques.

Some research involving required computations on the data-set is also discussed.

2.1. Queue definitions

Studies show multiple definitions and classifications of queues, depending on their specific situation. For the situation addressed this report, the following conceptual ideas for the definition of a queue were used (Fruin, 1971)(van den Heuvel, 2022)(Okazaki & Matsushita, 1993):

1. A spatial component: a line or group of pedestrians is involved
2. A time component: there is waiting involved
3. A cause: there is a reason for a pedestrian to wait for their turn

The definition of a queue for this report is the combination of these three factors that arises in the following situation: there are more pedestrians trying to enter the door than the capacity at that moment allows (van den Heuvel, 2022)(Kneidl, 2016).

Research by Okazaki and Matsushita (1993) shows a classification of queues in three types based on the movements of pedestrians in various public spaces:

- Type 1: in front of a counter, also known as a linear queue where the first person to arrive is the first to be served (van den Heuvel, 2022).
- Type 2: through multiple parallel gates such as tourniquets, causing parallel linear queues.
- Type 3: in front of doors of vehicles, involving possible situations where arriving passengers must exit before waiting passengers can enter.

As the visualisation of these types clearly shows (Figure 2.1), the third type is typical for the revolving door in this report. Pedestrians using the door as an exit use the same 'vehicle' as waiting pedestrians wish to enter. The formed queues are undisciplined (Fruin, 1971), a mass of pedestrians, where some join the end of the queue and others try to join at the bottleneck (van den Heuvel, 2022)(Daamen, 2004) but still more or less ordered (no pushing or jostling interaction between pedestrians) (Kneidl, 2016). The study by Kneidl (2016) reports that a no explicit study has been performed concerning this type of queue and no new literature was found during this literature study.

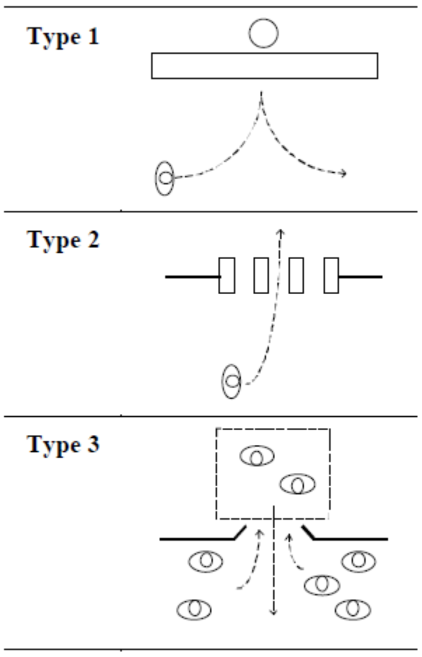


Figure 2.1: Three queue types according to Okazaki and Matsushita (1993)

A further distinction in queuing is made by Kneidl (2016) for bottlenecks (loose queue formation where a model with direction and velocity is used) and train boarding (bulk of people next to the entrance where a model with waiting zones is used). Factors of both can be identified for the revolving door in this report: a loose queue formation at a bottleneck (excluding emergency situations) combined with a waiting zone next to the 'opening' door.

2.2. Variables to describe queues

Based on the conceptual components of a queue described in section 2.1, the following variables were found to describe and/or measure queues in various studies:

- Amount [P] of pedestrians in a queue (spatial component) (Fruin, 1971) (van den Heuvel, 2022)
- Type of pedestrian, e.g. commuting or leisure (reason component) (Buchmüller & Weidmann, 2006) (Daamen, 2004)
- Area A [m^2] of a queue, also involving length and width (spatial component) (van den Heuvel, 2022)
- Density ρ [P/m^2] of a queue (spatial component) (Duives et al., 2015) (Daamen, 2004) (van den Heuvel, 2022) (Steffen & Seyfried, 2009) (Fruin, 1971)
- Flow [P/ms] of a queue (spatial and time component) (Daamen, 2004) (Buchmüller & Weidmann, 2006) (Steffen & Seyfried, 2009)
- Distance [m] to other pedestrians in a queue (spatial component) (Duives et al., 2015) (Kneidl, 2016)
- Waiting time [s] for pedestrians in a queue (time component) (Fruin, 1971) (van den Heuvel, 2022)
- Speed v [m/s] of pedestrians in a queue (spatial and time component) (Daamen, 2004) (Buchmüller & Weidmann, 2006) (Steffen & Seyfried, 2009)
- Direction $\vec{v}$ of pedestrians in a queue (spatial and time component) (Daamen, 2004) (Steffen & Seyfried, 2009)

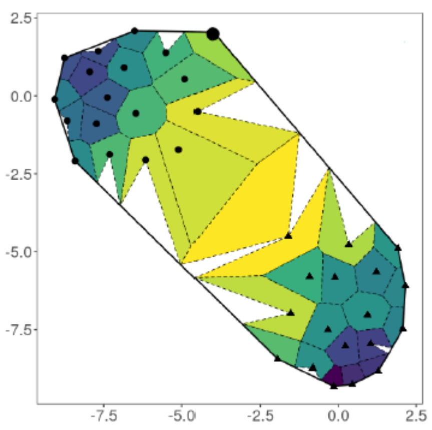


Figure 2.2: Example of a Voronoi diagram adjusted to compensate for lack of spatial boundaries (Mullick et al., 2022)

2.3. Queue density computation

Of all the variables to describe queues, density has been given the most attention by previous studies. But even though numerous ways have been proposed to quantify it, a complete answer is still missing. Each solution can only be applied for specific situations and more often than not, the parameters used in the solutions depend heavily on assumptions (Duives et al., 2015).

The research of van den Heuvel (2022) points out the X-T method and Voronoi diagram gave the best results for queue density computations in the overview study of Duives et al. (2015) for a bottleneck situation in his report that is very similar to the bottleneck in this report. Even so, two different methods of density computations were used as the X-T method and Voronoi diagram were too technically challenging to implement.

The first method is based on the research of Fruin (1971) with a classical partitioning in zones. Density is computed from the number of pedestrians in the area of the zone.

The second method is based on the research of Helbing (van den Heuvel, 2022) where exponentially weighted distance is used to create a normal distribution for density around a pedestrian. For a predefined grid, the combined weight of each pedestrian based on its distance to a cell defines its density. A couple of assumptions are made like the size of pedestrians (for which he used a simplification of dimensions used in Buchmüller and Weidmann (2006)) and the scale parameter of the distribution.

Since not only static pedestrian locations are known for this report, but also their traces with timestamps, the study of Mullick et al. (2022) gives multiple approaches to density computation along the traced path of objects. The Voronoi diagram gives the best results, but is especially problematic to apply in this report because of the wide open space before the entrance. This gives rise to possible infinitely low densities at the edges of a queue (Figure 2.2) that have to be corrected with new assumptions about maximum Voronoi cell size or boundaries.

2.4. Pedestrian parameters

The walking speed of pedestrians in free flow conditions appears to have a normal distribution with a mean $\mu = 1.34$ m/s and a standard deviation $\sigma = 0.37$ m/s according to Daamen (2004). Several factors can influence the walking speed, such as age, temperature and travel purpose (Buchmüller & Weidmann, 2006). The speed at which there is a 99% chance (P ≤0.01) for deviation from the free flow speed in the presented normal distribution is at $v \leq 0.47$ m/s. This corresponds with the observation of van den Heuvel (2022) that pedestrian speeds do not exceed 0.5 m/s when inside a queue.

In the study of Steffen and Seyfried (2009), methods for reducing scatter in measuring density, flow, speed and direction were presented for trajectory data. In case of speed, the scatter (variations) in momentary pedestrian speed caused by swaying through the walking motion can be compensated by finding moments of identical phase and interpolating between those points. It is unclear however if the data from the smart-sensor is accurate enough apply this.

An approximation of the size of a pedestrian in the horizontal plane is given by Buchmüller and Weidmann (2006) as 50 x 30 cm with an average body ellipse of 60 x 50 cm to account for elbowroom in different body positions. This was used as a reference for correct visual representation of pedestrians in animations and plots of the detection area.

2.5. Interpolation of data-points

For basic computations on the traces, like the numerical first derivative for speed, it is desirable to know an approximation of the location of an object at any given point in time instead of just the data-points at random moments. Interpolation of the data-points of a trace provides this information.

Linear interpolation The largest distances between data-points for a trace, using high walking speeds of 1.5 m/s (Daamen, 2004)(Buchmüller & Weidmann, 2006) and a rounded down average of 10 data-points per second, are in the order of 200 mm. Combined with the fact that change of direction of pedestrians is slower at higher speeds (inertia) and the scale of the detection area, linear interpolation provides a smooth enough trace.

This piece-wise linear interpolation between each data-point in a trace as a function of time can be described by the following equation:

$$f_{x,n}(t) = x_n + \left(\frac{x_{n+1} - x_n}{t_{n+1} - t_n}\right)(t - t_n), \quad \text{for } t \in [t_n, t_{n+1}), \quad n = 0, 1, \dots, m-2 \tag{2.1}$$

where: $f_{x,n}(t)$	[mm] =	the x-coordinate at timestamp t between data-point n and $n+1$
x_n	[mm] =	the x-coordinate of data-point n
t_n	[mm] =	the timestamp of data-point n
m	[$-$] =	the amount of data-points in the trace

Higher-order interpolation and smoothing It could be convenient to have a smooth function through all data-points in a trace with techniques like higher-order Lagrange interpolation. However, using a higher-order Lagrange interpolation to create single function for an object trace can lead to large oscillation at the end points, also known as Runge's phenomenon (Vuik et al., 2016). As the end points are exactly the point of interest in this study, the moment where queues exist before an object leaves the detection area through the door, this is highly undesirable. More advanced techniques could be used that prevent this, as well as other piece-wise functions with smooth transitions like splices. This would however greatly increase the computational load while linear interpolation is already sufficiently smooth for the extent of this report.

3 Data provided by the smart-sensor

All the data used in this report is gathered by a smart-sensor capable of tracking individual objects in its detection area. A permanent smart-sensor attached to the building above the entrance was installed on December 13, 2021 and has provided data since. The detection area of the smart-sensor can be seen in Figure 3.1. First, the output files of the smart-sensor are discussed in section 3.1. The limitations of the smart-sensor are discussed in section 3.2. The timestamp used in the data is discussed in section 3.3. The way the data was processed into a usable data-set is briefly described in section 3.4. Visualisation of the data-set is described in section 3.5 and lastly, the removal of false reflection data is worked out in section 3.6.

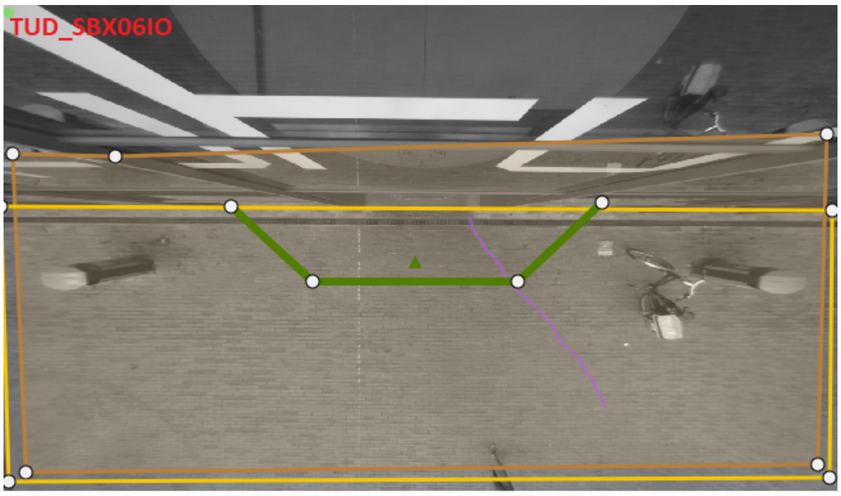


Figure 3.1: Detection area (orange) of the smart-sensor in front of the Industrial Design faculty (Daamen, 2023)

3.1. Output files with object trace data

The smart-sensor tracks moving objects across its detection area. It is capable of distinguishing individual objects, and stores their x and y coordinates in millimetres, as well as their unique object-id number and the Epoch Unix Timestamp (section 3.3) accurate in milliseconds at the moment of detection. The origin of the coordinate system is just in front of the entrance, with the x-axis parallel to the front of the building and going to the right and the y-axis perpendicular to the front of the building pointing away from the building. The limits of the detection area are from approximately -7000 mm to 6000 mm in x-direction and -3000 mm to 5000 mm in y-direction. The detection occurs a few times per second on a continuous basis (24 hours per day). The combined set of points in time for a single object gives the trace of the object through the detection area.

About every 27 seconds, an .JSON format output file is created containing the collected information for that time period. An example of such an output file can be seen in Appendix B. Object id-numbers

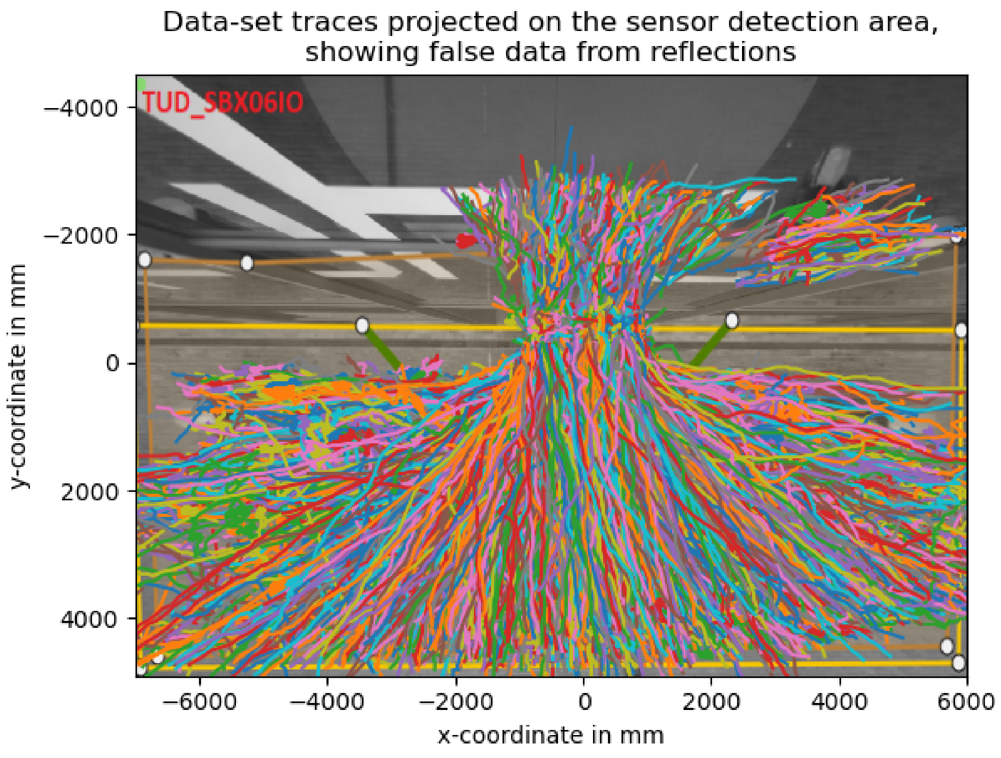


Figure 3.2: Object traces projected on the detection area showing reflections

are not limited to one output file, the same id-number is retained through other output files. Care has to be taken when parsing the .json files in python, as the complete trace of an object can be spread out across multiple output files.

3.2. Limitations of the smart-sensor

Although the smart-sensor can distinguish between different objects, it is incapable of identifying the objects themselves. A bird flying through the detection area, as well as a sliding piece of cardboard, are registered the same as pedestrians. These false sets of data are not accounted for in this report (chapter 6).

The size of objects is not registered, only their centre. Therefore, the data can not be used to distinguish between different types of objects, for example pedestrians with or without a backpack.

A part of the detection area of the smart-sensor is covered by the glass on the front of the building. The smart-sensor is incapable of distinguishing between objects in the walking area and reflections, causing some additional false data. This can be seen in Figure 3.2.

3.3. Epoch Unix Timestamp

The time information in the smart-sensor data is given as an Epoch Unix Timestamp accurate in milliseconds. This timestamp gives the amount of seconds passed since 00:00:00 UTC on the 1st of January 1970 (UnixTime.org, 2023). It is therefore time-zone independent. Each second is a full integer with the three decimal places used for milliseconds. Conversion to a date-time format can be done in various ways. The following is an example of a Unix Timestamp and its corresponding date-time (milliseconds are not given for the date-time):

Epoch Unix Timestamp:	1693567815.329
Local date-time:	Fri Sep 01 2023 13:30:15 GMT+0200

index	object_id	timestamp	first_timestamp	last_timestamp	x	y
0	201	$[t_0, t_1, ..., t_i]$	t_0	t_i	$[x_0, x_1, ..., x_i]$	$[y_0, y_1, ..., y_i]$
1	202	$[t_0, t_1, ..., t_i]$	t_0	t_i	$[x_0, x_1, ..., x_i]$	$[y_0, y_1, ..., y_i]$
⋮	⋮	⋮	⋮	⋮	⋮	⋮
n	200 + n	$[t_0, t_1, ..., t_i]$	t_0	t_i	$[x_0, x_1, ..., x_i]$	$[y_0, y_1, ..., y_i]$

Table 3.1: pandas DataFrame shape, with i depending on the amount of data for that object

3.4. Processing the smart-sensor data-set

The information of the output files of the smart-sensor was parsed to a pandas DataFrame (pandas, 2023). Special care was taken while parsing the information as .json files are seen as unordered (Python Software Foundation, 2023a) (Python Software Foundation, 2023b). This could cause the timestamp, x and y values to become scrambled.

The most convenient way to use the data was to store the data of each individual object on a single row in the DataFrame with the columns and cell values shown in Table 1. Objects spread over multiple output files had to be merged into a single row while ensuring the timestamps and corresponding x and y values were still in chronological order. The columns 'first_timestamp' and 'last_timestamp' were added for convenience to quickly show the extent of each object.

3.5. Visualisation

The smart-sensor information is visualised using matplotlib in Python. An example of a trajectory plot can be seen in Figure 3.3. The plot gives insight in the location of the door in the detection area and the general directions of the object traces.

Animations are used to visualise the trajectories and speed of objects. If a queue is identified, an animation of the moment in time can be used to verify the correctness of the identification. It can also be used to see if queue formation was missed by determining busy periods and visually inspecting the queue formations.

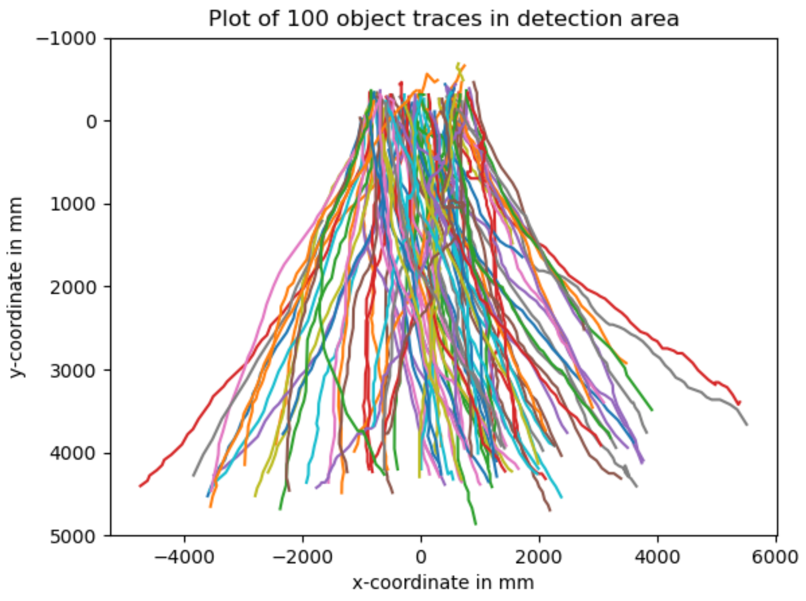


Figure 3.3: Trajectory plot

3.6. Removing false reflection data

In the provided data-set containing the information from the smart-sensor for the first two weeks of the academic year 2023/2024, 77705 individual objects were detected. As can be seen in Figure 3.4 (left), some objects are actually reflections picked up in the glass front of the building. After removing objects with a total travel distance of less than 1000 mm (reflections usually showed very short traces) or a y-coordinate below -1000 mm (a safe cut-off point without risking correct traces to be deleted), a total of 69013 objects remained. A sample from the remaining data-set can be seen in Figure 3.4.

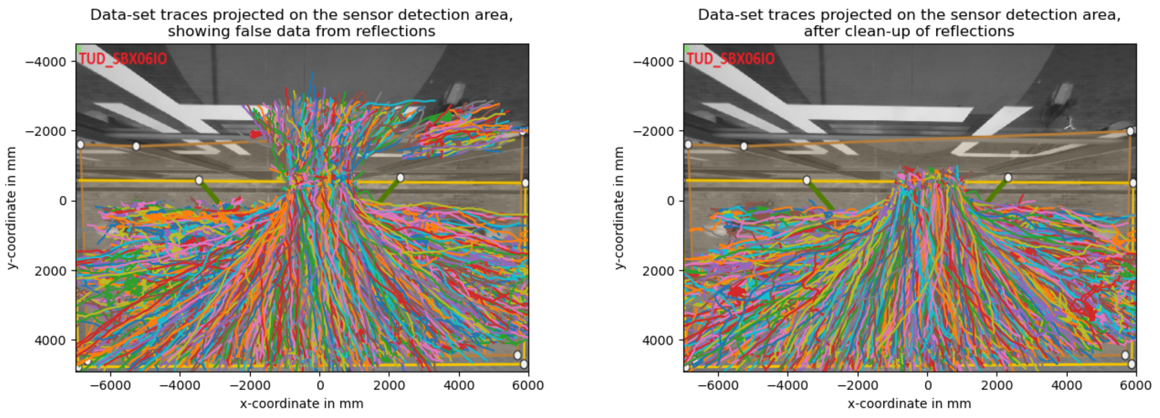


Figure 3.4: Detection area of the smart-sensor showing reflections (outside the yellow square) before (left) and after (right) correction of the data-set

4 Methodology

This chapter describes the steps that were taken to answer the following three sub-questions:

- *section 4.1*: How can a queue be defined based on the speed profile of objects?
- *section 4.2*: What are the required steps to identify queues in the smart-sensor data using this definition?
- *section 4.3*: How can the identified queues be verified?

The actions and computations were performed using the Python programming language. A detailed description of the code can be seen in appendix A.

4.1. Definition of a queue based on a speed profile

From the three queue components found in section 2.1, the spatial and time component can be found in the data-set. From an animation of queue formation in the data-set, a distinct 'stop and go' movement pattern was observed that was used to describe a definition of a queue in terms of the speed profile of an individual object just before entering the door. This definition revolves (pun intended) around capacity rather than the actual 'waiting in line'.

Revolving door cycle

Pedestrians arriving at a revolving door often have to wait for the door to be in the correct position to enter. This is from now on referred to as a door cycle, where each cycle is the moment pedestrians can enter or leave the door (not a full rotation). Even though waiting for the next door cycle would look like a queue formation, the 'stop and go' motion is not due to a lack of capacity but mere timing. Even with multiple pedestrians waiting for the door opening to be accessible, as long as every pedestrian can enter the door on the first opportunity (the first cycle after they arrive), capacity has not been reached.

Queue condition

From the moment a pedestrian is not able to enter the door at the first door cycle after they arrive, the capacity of the door is reached and a queue is formed. Even a single pedestrian would then count as a queue. Any pedestrian affected was given the formal definition of being in a queue before entering the door.

Queue condition described by a speed profile

The movement of pedestrians using a revolving door will have a typical 'stop and go' pattern when they have to wait before they can enter the door. This can be seen in the speed profile of an object: the speed variations of an object over time. As described earlier, one such 'stop and go' moment does not yet indicate a queue. When capacity is reached and a pedestrian is not able to enter the door at the first cycle, two such 'stop and go' patterns were observed. An arriving pedestrian first slows down to wait with other pedestrians. As the next door opening comes, the pedestrian moves with other pedestrians towards the door as they enter. If because of a capacity limit there was not enough time or room to enter the door, the pedestrian has to wait for the next cycle. From the moment a pedestrian exhibits two or more 'stop and go' motions, a queue formation based on capacity could be argued.

These 'stop and go' patterns can be described in terms of speed. With a certain speed threshold v_{lim}, a 'stop' movement has a lower speed than v_{lim} and a 'go' movement a higher speed than v_{lim}. The timing of these relative low and high speeds is synchronised with the timing of the door cycles. This was mathematically expressed in the following way:

$$v(t):\begin{cases}\text{'stop':} & v < v_{lim}, \quad t \neq t_0 - k\Delta t_d \pm \varepsilon \\ \text{'go':} & v > v_{lim}, \quad t = t_0 - k\Delta t_d \pm \varepsilon\end{cases} \tag{4.1}$$

where: $v(t)$	[m/s]	=	the speed of a pedestrian
v_{lim}	[m/s]	=	the speed threshold separating 'stop' and 'go' movements
t	[s]	=	the moment in time
t_0	[s]	=	the moment in time the door was entered
Δt_d	[s]	=	the time between two door cycles
ε	[s]	=	the error to compensate for Δt_d fluctuations and movement wave propagation delay
k	[-]	=	$0, 1, 2, \ldots$

A pedestrian was defined as being in a queue when two 'stop and go' motions are observed. In a simple descriptive way: there is a 'stop' movement, followed by a 'go' movement and a second 'stop' movement before the door is entered (the last 'go' movement where the door is entered was omitted to simplify the definition as it is not needed and often incomplete in the trajectory data). This was explicitly described using Equation 4.1 in the following way:

A pedestrian is defined as being in a queue when all three following conditions are met:

1. There is a 'stop' movement halfway between $t_0 - 2\Delta t_d$ and $t_0 - \Delta t_d$: the first 'stop'
2. There is a 'go' movement at $t = t_0 - \Delta t_d$: moving towards the door when the door can be entered
3. There is a 'stop' movement halfway between t_0 and $t_0 - \Delta t_d$: the second 'stop' movement when a pedestrian can not enter the door due to a capacity limit

4.2. Steps needed to identify queues

From this section onward, the word 'object' is used instead of 'pedestrian' as the trajectory data is being used and no confirmation can be given that a detected object was indeed a pedestrian.

4.2.1. Speed computation

To find the momentary speed of an object (in the direction of movement), first the speed of an object was computed in x and y direction using a backward difference formula. A backwards difference was chosen in order to get a result for the speed computation up to the last moment before the door was entered. To get the interpolated value of x and y for any time, the linear interpolation described in section 2.5 was used. The speed in the direction of movement was computed using the sum of the x and y components:

$$v(x(t), y(t)) = \sqrt{\left(\frac{x(t) - x(t-\Delta t)}{\Delta t}\right)^2 + \left(\frac{y(t) - y(t-\Delta t)}{\Delta t}\right)^2} \tag{4.2}$$

where: $v(x(t), y(t))$	[m/s]	=	the speed of an object at time t in the direction of movement
t	[s]	=	the moment in time
$x(t)$	[m]	=	the x-coordinate of an object at time t
$y(t)$	[m]	=	the y-coordinate of an object at time t
Δt	[s]	=	the time difference for the backward difference formula

4.2.2. Speed visualisation

A strong visual aid to assess the 'stop and go' movement is a plot of the speed of an object over the time before it enters the door. For these plots, the speed was computed and plotted for 0.1 second intervals. Even though animations of the data-set showed fluent movement of objects, large variations of the momentary speed of objects were observed when the data was plotted for speeds directly computed between successive data-points. By using the speed computation shown in Equation 4.2.1 and increasing the time difference Δt, a value of $\Delta t = 1$ s was found to give a sufficiently fluent result without flattening the curve too much. The different results are shown in Figure 4.1. At the smallest scale of $\Delta t = 0.001$ s, the speed computation is completely dependent on the linear interpolation and corresponds with speeds computed directly between successive data-points.

It must also be noted that a surprising effect was shown when the speed was plotted between successive data points (visible in Figure 4.1 for $\Delta t = 0.001$ s): recurring speeds over time, seemingly forming lines with slightly decreasing values over time. These were considered as artifacts from the smart-sensor detection methods and were not further investigated for the extent of this report as their effect was no longer visible for larger values of Δt.

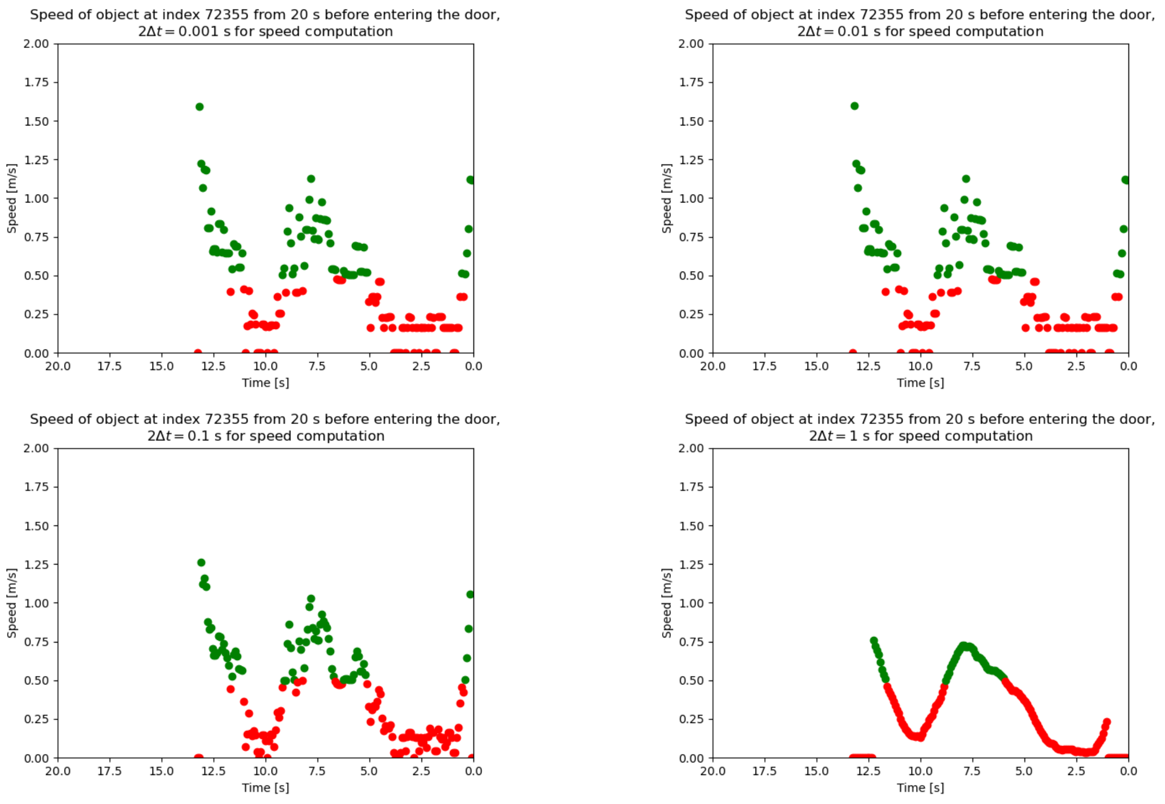


Figure 4.1: Comparison between $2\Delta t = [0.001, 0.01, 0.1, 1]$ [s] for speed computation.

A time window of 20 seconds before entering the door was chosen for the speed profile plots. Looking back further in time for each object has no added value, as pedestrians rarely dwell in the detection area for longer than 20 seconds. As the detection area is quite small, objects are often already out of bounds before this time window when waiting in a longer queue. The rest of the queue can not be detected. This can be seen in the animation snapshot in Figure 4.2. For the detection of queues however, this is not a problem, for if a pedestrian is queuing out of bound, another pedestrian will be queuing inside the detection area.

4.2.3. Speed threshold and required queue conditions

As a first estimation for the speed threshold v_{lim}, the value of 0.5 m/s found in section 2.4 was used. When the speed profiles of multiple objects were compared, large variations in timing were observed

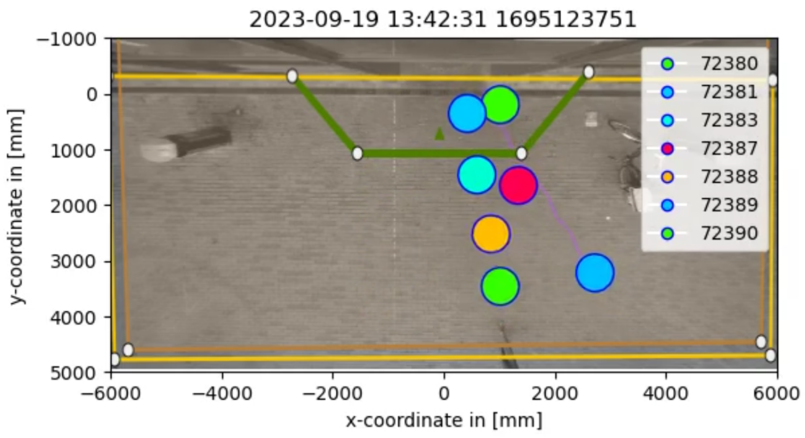


Figure 4.2: Animation snapshot showing a queue with real life size approximation of pedestrians

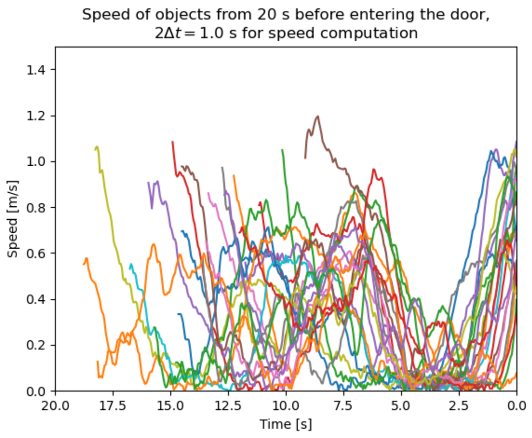


Figure 4.3: Plot of speed profiles of multiple objects showing a large spread in ’stop and go’ timing

for the ’stop and go’ movements. The large spread in timing can be seen in Figure 4.3. Because of this spread, as well as large fluctuations in speed (add figure), a simple and clear implementation of the queue conditions described in section 4.1 was not possible without adjustments.

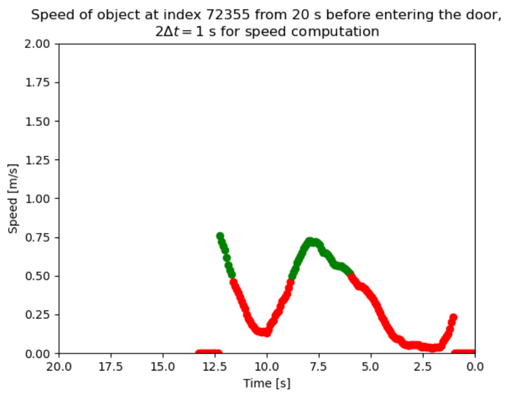
Door cycle time and error interval

The interval for the error ε in Equation 4.1 to compensate for the large spread in timing was first estimated by visual observation of speed profiles from objects showing ’stop and go’ movements. The interval for which each condition was valid ($v < v_{lim}$ for conditions 1 and 3 and $v > v_{lim}$ for condition 2) was noted. For all three conditions, the interval was found to be around 7 seconds ($\varepsilon = 3.5$ s) centered around 11.5, 8.5 and 4.5 seconds before entering the door for condition 1, 2 and 3 respectively.

Animations were used to determine the door cycle length Δt_d. On average, the waiting time between two moments of entry (between the last object entering the door and the first one of the next cycle) was around 7 seconds. The average time between two full cycles (between the first object to enter the door and the first object of the next cycle) was around 10 seconds. As only a maximum of two ’stop and go’ movements are needed and can be seen, a $\Delta t_d = 7$ s between to moments of entry was chosen as more appropriate as movement starts immediately after this interval, compared to the entire cycle. This was also in perfect unison with the estimated intervals based on observations. Adding a 1 second shift to Equation 4.1 to compensate for the delay before an object starts moving also lined up the timing with the observations.

add example 3712 (fast) and 4504 (slow) Two examples are shown in Figure 4.4.

Due to speed fluctuations and large overlaps for condition intervals due to spread in timing, a single



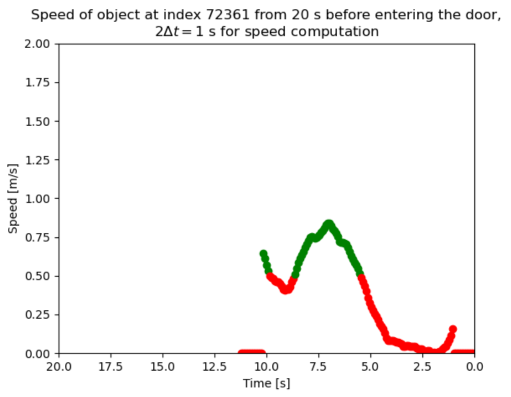


Figure 4.4: Example of the speed graph of two objects for the last 20 seconds before they go through the entrance. The speed threshold arbitrarily set at 0.5 m/s based on section 2.4

point in the speed profile where the speed was above or below v_{lim} for each condition was deemed insufficient. A minimal time limit for these conditions ($\Delta t_1, \Delta t_2$ and Δt_3) was added to make sure 'stop' and 'go' movements were not incidental spikes in speed measurement. Additionally, to make sure such an incidental spike does not dismiss a valid queue identification, a maximum of 10% mismatch was allowed. For the length of Δt_2 and Δt_3, 2 consecutive seconds was estimated based on observations, as both conditions have to be fully present for all conditions to be valid. The first condition however could be very short because an object could arrive just before the start of the first 'go' wave. As observations indicated that queues could be missed otherwise, the time limit for condition 1, Δt_1 was set to 0.5 seconds. Just like the plots of the speed profiles, the speed in the intervals was checked for every 0.1 seconds.

The resulting conditions based on the description in section 4.1 are now as follows:

1. $v < v_{lim}$ for > 90% of Δt_1 during $t = t_0 - (1.5\Delta t_d + 1) \pm 0.5\Delta t_d$
2. $v > v_{lim}$ for > 90% of Δt_2 during $t = t_0 - (\Delta t_d + 1) \pm 0.5\Delta t_d$
3. $v < v_{lim}$ for > 90% of Δt_3 during $t = t_0 - (0.5\Delta t_d + 1) \pm 0.5\Delta t_d$

where: v	[m/s]	=	the speed of an object at time t, checked every 0.1 seconds
v_{lim}	[m/s]	=	the speed threshold for 'stop' or 'go' movement
Δt_1	[s]	=	the minimal time limit for condition 1: 0.5 seconds
Δt_2	[s]	=	the minimal time limit for condition 2: 2 seconds
Δt_3	[s]	=	the minimal time limit for condition 3: 2 seconds
t_0	[s]	=	the moment in time the door was entered
Δt_d	[s]	=	the time between two door cycles: 7 seconds

Speed threshold

The threshold v_{lim} was estimated at 0.35 m/s in later iterations which indeed gave better results. Even though the literature mentions 0.5 m/s as threshold, it also mentions it usually never being exceeded in a queue. Observations do however show that 0.5 m/s is occasionally exceeded in stop and go waves, emphasizing the difference between classical queues and the situation at a revolving door. The application of the threshold with the three intervals in a speed profile plot can be seen in Figure 4.5.

During inspection of missed queues using the 0.35 m/s threshold, it was observed that a single fixed threshold was not able to detect queues precisely enough due to varying circumstances. In order to make sure that all variations were captured, the final method uses an accumulation of all queues found for various speed limits: v_{lim} = 0.2, 0.3, 0.4, 0.5

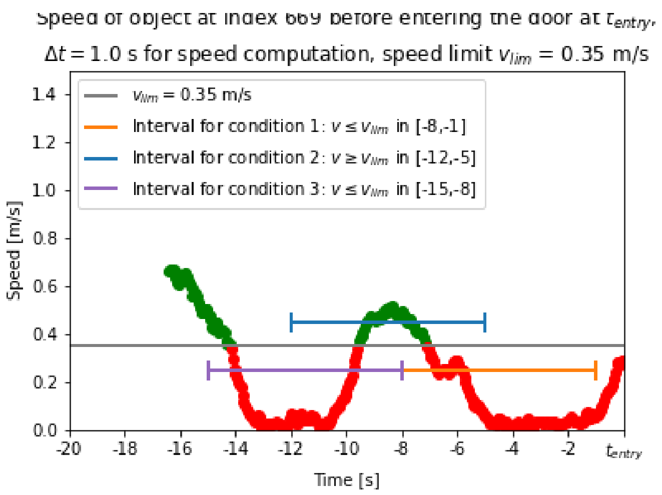


Figure 4.5: Three intervals for queue detection: 1) red: $\vec{v} < \vec{v}_{lim}$, 2) green: $\vec{v} > \vec{v}_{lim}$ and 3) blue: $\vec{v} < \vec{v}_{lim}$

4.2.4. Selecting viable objects

To make sure only viable objects were checked for the conditions of section 4.1, objects with the wrong direction of movement and a shorter total detection time than needed were filtered out of the data-set.

The direction of movement was used to identify objects that did not leave the detection area through the door. Using the trajectory plot in Figure 3.3, it can be seen that all trajectories converge at the door at the top of the figure. It was estimated that objects going in the negative y-direction during the last 5 seconds (to compensate for small movements while standing still) of their trajectory path were leaving the detection area through the door. This was expressed as $y'(t) < 0$.

The minimal total detection time needed to meet all three conditions in section 4.1 was determined by the limits given in subsection 4.2.3. With an estimated door cycle length of $\Delta t_d = 7$ seconds, the 1 second compensation shift before movement starts, a consecutive length of 0.5 seconds for condition 1 and 1 second needed for the time difference in the speed computation, the minimal required detection time added up to 9.5 seconds.

After these objects were filtered out, 3846 viable objects were left from the total 69013 objects in the data-set.

4.3. Verifying results

Verifying the results was done by visual inspection of the speed profiles of random objects in the data-set. Discrepancies are easily spotted in the speed profiles. Where speed profiles were inconclusive, animations were created for a last visual inspection. This allowed for a thorough, though labour intensive verification of the results.

5 Results

With the method and conditions for queue detection described in chapter 4, a list was generated with objects that were identified as being in a queue. First, the observations of the identified queues are given in section 5.1. The results are verified in section 5.2.

5.1. Queue identification

The Python code in Appendix A was used to check each of the 69013 objects in the database for the queue requirements defined in subsection 4.2.3. After filtering out viable objects for direction and total detection time, 3846 were left. Of these objects, a total of 1109 were found to satisfy all requirements, or in other words: were identified as being in a queue before entering the door. The results were verified in the next section to see if the found objects are indeed correctly identified as being in a queue.

5.2. Verification of the results

To verify the results, the identified objects and random samples from all objects were manually checked for correct identification. The manual check consisted of a visual check of the speed profile for each object, as inconsistencies are easily spotted. For the situations where the speed profile of an object was inconclusive, an animation was used as a final confirmation.

Identified queues

Of all the 1109 identified objects being in a queue, 4 could be found as incorrectly identified (false positives). All of these were caused by objects moving seemingly at random in front of the door before entering. As there is no video confirmation available, the exact nature of the situations can only be guessed. As an example, a simple explanation could be a pedestrian waiting for someone else before entering the door. All these occurrences were unavoidable with using the speed profile method in this report.

Around 20 random objects were also checked using an animation which indeed showed queuing behaviour. As queuing is often with multiple pedestrians at the same time, other objects in the animations that also showed queuing behaviour were checked for identification as well. All these objects were indeed correctly identified.

Random samples

To check for false negatives (where a queue was not identified when it should have been), two different random manual verifications were performed. A 1000 random objects out of the original data-set of 69013 and 100 random objects out of the viable 3846 objects were manually checked for discrepancies.

Of those 1000 objects:

- 4 were previously identified as queues by the program
- 34 showed a speed profile with possible queuing or other discrepancies and were animated for visual confirmation

- 1 of those 34 was found to be a false negative, though an edge case with room for discussion, all others were correctly dismissed as queues

Out of the 100 random objects from the set of 3846 viable objects (in this case the identified queues were not included):

- all were checked using animations
- 4 were found as false negatives
- 3 of those coincidentally were from the same situation, where an unusually long period was observed before the door could be entered again, causing the first ’stop and go’ wave to fall outside of the selected intervals.

6 Discussion

6.1. Queues and capacity

The emphasis on the definition of a queue in this report is the moment capacity is reached. With the more classical approach however, using the same three conceptual ideas (section 2.1: a line or group waiting for their turn), it could also be argued that for the situation of a revolving door, there is already queue formation before capacity is reached. With the more common density computations used for queues (section 2.3), these formations would also count as queues.

The method to identify queues discussed in this report is especially useful when studying the capacity of a revolving door, as it can distinguish between queues formed due to capacity limits and queues formed during normal use of the door. Classical computations like using density are typically not able to make this distinction.

6.2. Smoothing or smart-sensor hardware to compensate for speed fluctuations

Like discussed in Equation 2.5, smoothing of the trajectory data using higher-order functions or other methods was not used for this report. Even though animations showed that linear interpolation was sufficiently smooth, a relatively large Δt of 1 second was needed for the speed computation as the momentary speed still showed large fluctuations (subsection 4.2.1). New studies are needed to see if smoothing or other compensation could improve the results.

This also depends largely on the type of smart-sensor for the trajectory data. Fine tuning will be needed depending on the situation for which the method is used.

6.3. Detection area size

While viewing animations during queue formation, it was noticed that the size of the detection area was barely larger than the portion of the queue that is able to enter during one cycle of the door. This means that the path of a pedestrian entering the detection area and joining the queue is often very short and that longer queue formations can not be studied.

6.4. Influence of weather conditions

Although the relatively small span of two workweeks gave plenty of data for creating and testing the method in this report, it is not enough to investigate the influences of weather conditions on the results. The two weeks had high temperatures with little fluctuations. The described method to identify queues could very well give different results due to behaviour changes in (heavy) rain or cold temperatures. This still needs to be tested in future studies.

6.5. Unexpected pedestrian behaviour

The method of finding queues from trajectory data derived in this report can not account for unexpected behaviour of pedestrians. Situations like stopping and looking at one's phone just before entering the door could lead to false queue identifications. As there is no video footage to compare to the trajectory data, some situations can not be explained through the trajectory data. Though exact numbers were not found, results indicate that the contribution of these situations to false data is very small.

6.6. Smart-sensor errors

Some errors due to limitations of the smart-sensor were noticed that could influence the results of this report. For example: an object was seen to merge with another, possibly due to likeliness. Two objects were also seen with an extremely close and consistent proximity to one other, possibly a pedestrian carrying a large item. Even though this prevents the possibility for exact measurements, the encounters of such errors were so rare that they are considered negligible for the purpose of this report.

6.7. Minimal time limits

The minimal consecutive time limits for the interval conditions were chosen as 2 seconds for condition 2 and 3 and 0.5 seconds for condition 1. As discussed in section 4.2, the 0.5 seconds was already quite limited due to the detection area size. For condition 2 and 3 however, the 2 second limit was estimated by observation. When using a longer limit of 3 seconds, only two-thirds of the queues were found, resulting in a large number of false negatives. With a shorter limit of 1 second however, almost double the amount of queues were identified with an enormous amount of false positives, indicating the importance of this interval approach. More fine-tuning could be studied for better results.

7 Conclusion

The goal of this report was to answer the research question: *"How can queues at the revolving door of the Industrial Design faculty entrance be identified using speed and direction information from smart-sensor trajectory data?"*. The sub-questions and the sections in which they are answered are as follows:

- *section 7.1:* What is currently known in literature about queue identification?
- *section 7.2:* What information does the smart-sensor data provide?
- *section 7.3:* How can a queue be defined for the specific situation?
- *section 7.4:* What are the required steps to identify queues in the smart-sensor data using this definition?
- *section 7.5:* How can the identified queues be verified?
- *section 7.6:* Results

7.1. Literature study

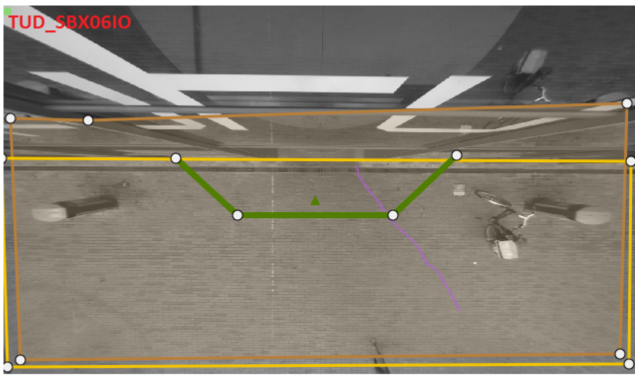
Literature showed that the specific queue type in front of a revolving door has not yet been explicitly studied (Kneidl, 2016). Density was used most often for queue description and definition but there is no complete answer. Each application is dependent on specific solutions and assumptions (Duives et al., 2015). Although speed is occasionally used for description and comparisons to other variables, it has not yet been used explicitly to identify queues.

7.2. Smart-sensor information

The information provided by the smart-sensor consists of an x and y coordinate in mm together with an Epoch Unix Timestamp (UnixTime.org, 2023) accurate in milliseconds for each detection of an individual object in the detection area. The origin of the coordinate system is just in front of the entrance. There are multiple detection moments per second. The combination of all coordinates and timestamps of an individual object gives the trace (or trajectory) of an object in the detection area. The smart-sensor creates a .JSON format output file (an example can be seen in appendix B) for the accumulated information every 27 seconds.

The smart-sensor has the following limitations:

- The smart-sensor can distinguish between individual objects but is incapable of identifying objects: a piece of sliding cardboard is registered the same as a pedestrian
- The size of objects is not registered, only their centre
- A part of the detection area consists of glass of the front of the building, reflections in the glass can be registered as objects (these objects were removed where possible to prevent false readings)



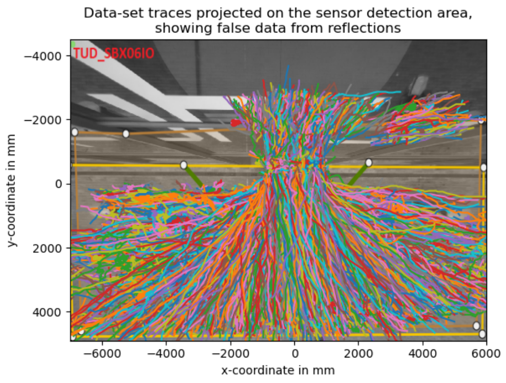


Figure 7.1: The detection area (left) and a sample of object trajectories including false data from reflections plotted on the detection area (right)

The timestamps used in the smart-sensor information are given in the Epoch nix Timestamp format: the amount of seconds since the 'Epoch' on the 1st of January 1970. The timestamps are accurate in milliseconds.

7.3. Queue definition

The definition of a queue in this report is based on a distinct movement pattern of pedestrians when the door capacity is reached. This movement pattern was observed using an animation of queuing pedestrians and consists of a 'stop and go' pattern. Pedestrians arriving at a revolving door often have to wait for the door to be in the correct position to enter. Even though waiting for the next door cycle would look like a queue formation, the 'stop and go' motion is not due to a lack of capacity but mere timing.

From the moment a pedestrian is not able to enter the door at the first door cycle after they arrive, the capacity of the door is reached and a queue is formed. Even a single pedestrian would then count as a queue. Any pedestrian affected was given the formal definition of being in a queue before entering the door.

These 'stop and go' patterns can be described in terms of speed. With a certain speed threshold v_{lim}, a 'stop' movement has a lower speed than v_{lim} and a 'go' movement a higher speed than v_{lim}. The timing of these relative low and high speeds is synchronised with the timing of the door cycles. This was mathematically expressed in the following way:

$$v(t):\begin{cases}\text{'stop'}: & v < v_{lim}, \quad t \neq t_0 - k\Delta t_d \pm \varepsilon \\ \text{'go'}: & v > v_{lim}, \quad t = t_0 - k\Delta t_d \pm \varepsilon\end{cases} \tag{7.1}$$

where: $v(t)$	[m/s]	=	the speed of a pedestrian
v_{lim}	[m/s]	=	the speed threshold separating 'stop' and 'go' movements
t	[s]	=	the moment in time
t_0	[s]	=	the moment in time the door was entered
Δt_d	[s]	=	the time between two door cycles
ε	[s]	=	the error to compensate for Δt_d fluctuations and movement wave propagation delay
k	[-]	=	$0, 1, 2, ...$

A pedestrian was defined as being in a queue when two 'stop and go' motions are observed. There is a 'stop' movement, followed by a 'go' movement and a second 'stop' movement before the door is entered. This was explicitly described in the following way:

A pedestrian is defined as being in a queue when all three following conditions are met, looking backwards from the moment the door was entered:

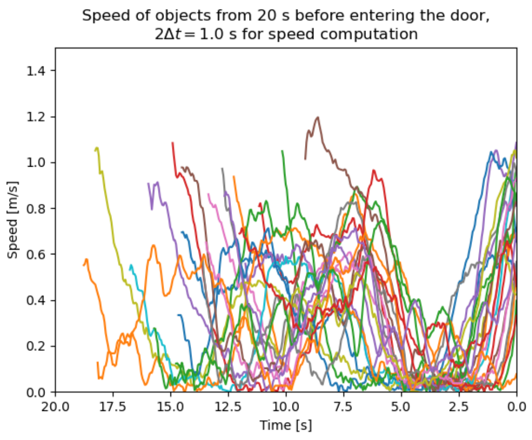


Figure 7.2: Plot of speed profiles of multiple objects showing a large spread in 'stop and go' timing

1. There is a 'stop' movement halfway between $t_0 - 2\Delta t_d$ and $t_0 - \Delta t_d$: the first 'stop'
2. There is a 'go' movement at $t = t_0 - \Delta t_d$: moving towards the door when the door can be entered
3. There is a 'stop' movement halfway between t_0 and $t_0 - \Delta t_d$: the second 'stop' movement when a pedestrian can not enter the door due to a capacity limit

7.4. Steps to identify queues

Speed computation
The momentary speed was computed using a backward difference method based on linear interpolation for the speeds in x and y direction and computing their sum:

$$v(x(t), y(t)) = \sqrt{\left(\frac{x(t) - x(t - \Delta t)}{\Delta t}\right)^2 + \left(\frac{y(t) - y(t - \Delta t)}{\Delta t}\right)^2} \tag{7.2}$$

where:	$v(x(t), y(t))$	[m/s]	=	the speed of an object at time t in the direction of movement
	t	[s]	=	the moment in time
	$x(t)$	[m]	=	the x-coordinate of an object at time t
	$y(t)$	[m]	=	the y-coordinate of an object at time t
	Δt	[s]	=	the time difference for the backward difference formula

Speed visualisation
The speed of objects was visualised plotting it over the time before an object entered the door. Even though animations of the data-set showed fluent movement of objects, large variations of the momentary speed of objects were observed. A value of $\Delta t = 1$ s was found to give a sufficiently fluent result without flattening the curve too much.

Speed threshold and required queue conditions
As a first estimation for the speed threshold v_{lim}, the value of 0.5 m/s found in section 2.4 was used. Large variations in timing were observed for the 'stop and go' movements. The large spread in timing can be seen in Figure 7.2.

Door cycle time and error interval
Animations were used to determine the door cycle length Δt_d. On average, the waiting time between two moments of entry (between the last object entering the door and the first one of the next cycle) was around 7 seconds. As only a maximum of two 'stop and go' movements are needed and can be seen,

a $\Delta t_d = 7$ s between to moments of entry was chosen. This was also in perfect unison with estimated intervals based on observations. Adding a 1 second shift to compensate for the delay before an object starts moving also lined up the timing with the observations.

Due to speed fluctuations and large overlaps for condition intervals due to spread in timing, a minimal time limit was added for each condition ($\Delta t_1, \Delta t_2$ and Δt_3). Additionally, to make sure such an incidental spike in speed does not dismiss a valid queue identification, a maximum of 10% mismatch was allowed. For the length of Δt_2 and Δt_3, 2 consecutive seconds was estimated based on observations. The first condition however could be very short because an object could arrive just before the start of the first 'go' wave. As observations indicated that queues could be missed otherwise, the time limit for condition 1, Δt_1 was set to 0.5 seconds.

The resulting conditions based on the description in section 4.1 are now as follows:

1. $v < v_{lim}$ for > 90% of Δt_1 during $t = t_0 - (1.5\Delta t_d + 1) \pm 0.5\Delta t_d$
2. $v > v_{lim}$ for > 90% of Δt_2 during $t = t_0 - (\Delta t_d + 1) \pm 0.5\Delta t_d$
3. $v < v_{lim}$ for > 90% of Δt_3 during $t = t_0 - (0.5\Delta t_d + 1) \pm 0.5\Delta t_d$

where: v	[m/s]	=	the speed of an object at time t, checked every 0.1 seconds
v_{lim}	[m/s]	=	the speed threshold for 'stop' or 'go' movement
Δt_1	[s]	=	the minimal time limit for condition 1: 0.5 seconds
Δt_2	[s]	=	the minimal time limit for condition 2: 2 seconds
Δt_3	[s]	=	the minimal time limit for condition 3: 2 seconds
t_0	[s]	=	the moment in time the door was entered
Δt_d	[s]	=	the time between two door cycles: 7 seconds

Speed threshold
The threshold v_{lim} was estimated at 0.35 m/s in iterations. Even though the literature mentions 0.5 m/s as threshold, it also mentions it usually never being exceeded in a queue. Observations do however show that 0.5 m/s is occasionally exceeded in stop and go waves, emphasizing the difference between classical queues and the situation at a revolving door.

During inspection of missed queues using the 0.35 m/s threshold, it was observed that a single fixed threshold was not able to detect queues precisely enough due to varying circumstances. In order to make sure that all variations were captured, the final method uses an accumulation of all queues found for various speed limits: v_{lim} = 0.2, 0.3, 0.4, 0.5

Selecting viable objects
To make sure only viable objects were checked for being in a queue, objects with the wrong direction of movement and a shorter total detection time than needed were filtered out of the data-set. The direction of movement was found using $y'(t) < 0$ for the last 5 seconds of an object. The minimal total detection time needed to meet all three conditions in section 4.1 was determined to be to 9.5 seconds. After these objects were filtered out, 3846 viable objects were left from the total 69013 objects in the data-set.

7.5. Verification

Verifying the results was done by visual inspection of the speed profiles of random objects in the data-set. Discrepancies are easily spotted in the speed profiles. Where speed profiles were inconclusive, animations were created for a last visual inspection. This allowed for a thorough, though labour intensive verification of the results.

7.6. Results

The Python code found in Appendix A was used to check each of the 69013 objects in the database for the defined queue requirements. After filtering out viable objects for direction and total detection time, 3846 were left. Of these objects, a total of 1109 were found to satisfy all requirements, or in other words: were identified as being in a queue before entering the door.

To verify the results, the identified objects and random samples from all objects were manually checked for correct identification. The manual check consisted of a visual check of the speed profile for each object, as inconsistencies are easily spotted. For the situations where the speed profile of an object was inconclusive, an animation was used as a final confirmation.

Identified queues

Of all the 1109 identified objects being in a queue, 4 could be found as incorrectly identified (false positives). All of these were caused by objects moving seemingly at random in front of the door before entering. All these occurrences were unavoidable with using the speed profile method in this report. Around 20 random objects were also checked using an animation which indeed showed queuing behaviour. As queuing is often with multiple pedestrians at the same time, other objects in the animations that also showed queuing behaviour were checked for identification as well. All these objects were indeed correctly identified.

Random samples

To check for false negatives (where a queue was not identified when it should have been), two different random manual verifications were performed. A 1000 random objects out of the original data-set of 69013 and 100 random objects out of the viable 3846 objects were manually checked for discrepancies.

Of those 1000 objects:

- 4 were previously identified as queues by the program
- 34 showed a speed profile with possible queuing or other discrepancies and were animated for visual confirmation
- 1 of those 34 was found to be a false negative, though an edge case with room for discussion, all others were correctly dismissed as queues

Out of the 100 random objects from the set of 3846 viable objects (in this case the identified queues were not included):

- all were checked using animations
- 4 were found as false negatives
- 3 of those coincidentally were from the same situation, where an unusually long period was observed before the door could be entered again, causing the first 'stop and go' wave to fall outside of the selected intervals.

7.7. Final conclusion

The results show that the speed of objects can indeed be used to identify queues from trajectory data. The computations are quick (200 seconds for 70000 objects on an average computer) compared to known density methods, while also being able to focus on the capacity of the door and can distinguish between queues and groups of pedestrians standing in close proximity to the door.

The results also show that with rough estimations for the used parameters, a very solid result is produced with very limited false positive and false negative results. With further fine-tuning of the method, it could prove to be a very useful tool in identifying queues.

8 Recommendations

8.1. Smart-sensor placement and detection area adjustment

Due to a part of the detection area of the smart-sensor overlapping with the glass front of the building, false data is accumulated due to reflections in the glass. A repositioning of the smart-sensor could help not only to improve the quality of the data, but also to create a larger view of the actual pavement.

8.2. Fine-tuning of the method

As there was not enough time to improve the method of queue identification in this report, it is recommended that the used parameters are studied for improvement of the results.

8.3. Testing the method on different circumstances

As only a small portion of the available data of the smart-sensor was used, other weather conditions like rain and cold temperatures could not be accounted for. Research is needed to verify the results of the method for different circumstances.

8.4. Applying the method to other revolving doors

As only one data-set of one smart-sensor was available, the method could not be tested for different revolving doors and surroundings. Before the method can be used in general, this has to be studied first.

Appendix A
Python code

Python code to parse JSON file data into Pandas DataFrame: In order to access and work with the data provided by the smart-sensor (section 3.1), the output files were parsed to a pandas DataFrame (pandas, 2023) using the Python programming language. Due to the nested json object format of the output files (Appendix B), pandas could not be used to directly create a DataFrame from the files themselves. A workaround was used where the json files were opened and then parsed into a Python dictionary using the `json.load()` method from the `json` module:

```python
import json
with open('file_name') as user_file:
    file_cont_dict = json.load(user_file)
```

Special care has to be taken due to the nature of the json and python dictionary format. Both json and python dictionary store data as key/value pairs without a corresponding index, meaning they are unordered by nature and data can only be accessed by calling the key as opposed to an index number (Python Software Foundation, 2023a)(Python Software Foundation, 2023b). Since the json parser `json.load()` processes the file data in the order of the file contents and Python dictionaries retain their order since Python version 3.7 (Python Software Foundation, 2023a)(Python Software Foundation, 2023b), nested **`for`** loops could be used to extract the 'x', 'y' and 'timestamp' data from within the nested dictionaries without risking corruption of data:

```python
for key1, value1 in file_cont_dict.items():
    for key2, value2 in value1.items():
        traces = value2.get('traces')
        for trace_key, trace_value in traces.items():
            data = {}
            data['object_id'] = trace_key
            data['timestamp'] = trace_value.get('timestamp')
            data['first_timestamp'] = min(trace_value.get('timestamp'))
            data['last_timestamp'] = max(trace_value.get('timestamp'))
            data['x'] = trace_value.get('x')
            data['y'] = trace_value.get('y')
```

Python dictionaries have strictly unique key names, if a key/value pair is added to a dictionary that already has the same key, only the last added entry is retained (Python Software Foundation, 2023b). As shown in Appendix B, the json file is structured such that each key is unique, making it safe to parse without the risk of losing data. When combining the output files into one DataFrame however, objects spread over multiple files will have the same object-id number as key. To prevent data from being overwritten, each object-id number from each file with its corresponding data was stored as a separate dictionary in a list. Using the `pandas.DataFrame()` method, the list of dictionaries was turned into a pandas DataFrame with a row for each dictionary:

```python
import pandas as pd
data_list = [{dict_1}, {dict_2}, ..., {dict_n}]
df = pd.DataFrame(data_list, index=None)
```

The objects spread out over multiple rows were grouped together without changing the order of the data using the `pandas.DataFrame.groupby().agg()` method, creating a DataFrame with each row a unique and complete trace of a single object:

index	object_id	timestamp	first_timestamp	last_timestamp	x	y
0	201	$[t_0, t_1, ..., t_i]$	t_0	t_i	$[x_0, x_1, ..., x_i]$	$[y_0, y_1, ..., y_i]$
1	202	$[t_0, t_1, ..., t_i]$	t_0	t_i	$[x_0, x_1, ..., x_i]$	$[y_0, y_1, ..., y_i]$
⋮	⋮	⋮	⋮	⋮	⋮	⋮
n	200 + n	$[t_0, t_1, ..., t_i]$	t_0	t_i	$[x_0, x_1, ..., x_i]$	$[y_0, y_1, ..., y_i]$

Table 1: pandas DataFrame shape, with i depending on the amount of data for that object

```python
df = df.groupby('object_id',
                as_index=False
                ).agg({'timestamp': 'sum',
                       'first_timestamp': 'first',
                       'last_timestamp': 'last',
                       'x': 'sum',
                       'y': 'sum'})
```

The shape of the resulting pandas DataFrame can be seen in Table 1.
The complete code is as follows:

```python
# loading required packages
import json
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import os
import time
from datetime import datetime

# monitor code execution time
start = time.time()

# initialise list to store file contents as dictionary of a
data_list = []
for folder in os.listdir():
    if folder[:4] == '2023':
        path = f'{folder}/{folder}/traces_json'
        for file_name in os.listdir(path):
           # open the file and load in python as dictionary with json.load()
            with open(str(path + '/' + file_name)) as user_file:
                file_contents = json.load(user_file)
            # loop through the (key,value) combinations of the dictionary
             ↪ items
            for key1, value1 in file_contents.items():
                # loop through the (key,value) combinations of the
                # second nested dictionary
                for key2, value2 in value1.items():
                    # store the next nested dictionary as variable 'traces'
                    traces = value2.get('traces')
                    # Loop through the 'traces' dictionary, keys are the
                     ↪ object id's
                    # with values of 'timestamp', 'x', and 'y'.
                    # Each object id creates a new dictionary to add to the
                     ↪ data_list.
                    for trace_key, trace_value in traces.items():
                        # initialise new dictionary to store object id data
                        data = {}
                        # store the object id in the 'object_id' column
                        data['object_id'] = trace_key
                        # store the timestamp values as list in the
                         ↪  'timestamp' column
                        data['timestamp'] = trace_value.get('timestamp')
                        # store the first and last timestamps in separate
                        # columns for convenience
                        data['first_timestamp'] =
                         ↪  min(trace_value.get('timestamp'))
                        data['last_timestamp'] =
                         ↪  max(trace_value.get('timestamp'))
                        # store x and y values as lists in the 'x' and 'y'
                         ↪  columns
                        data['x'] = trace_value.get('x')
                        data['y'] = trace_value.get('y')
                      # add the new dictionary filled with the file contents
                        # to the data_list
                        data_list.append(data)


# convert the data_list of dictionaries to a pandas DataFrame
df_traces_json = pd.DataFrame(data_list, index=None)

# group measurements of objects that were saved in separate save files
# together in one row (every object has a single row with all datapoints)
df_traces_json = df_traces_json.groupby('object_id',
                                        as_index=False
                                        ).agg({'timestamp': 'sum',
                                               'first_timestamp': 'first',
                                               'last_timestamp': 'last',
                                               'x': 'sum',
                                               'y': 'sum'})
df_traces_json = df_traces_json.sort_values('first_timestamp')
# Reset index starting at 0
df_traces_json = df_traces_json.reset_index(drop=True)
df_traces_json.to_pickle('df_traces_json_raw')


df = df_traces_json
drop_list = []
for i in range(len(df)):
    dist = 0
    x = df.iloc[i]['x']
    y = df.iloc[i]['y']
    for j in range(len(x) - 1):
        dist += np.sqrt(((x[j] - x[j+1]) ** 2)
                        + ((y[j] - y[j+1]) ** 2))
    if (dist < 1000) or (np.min(y) < -1000):
        # print('idx ', i, ' dropped')
        drop_list.append(i)

df = df.drop(drop_list)
df = df.reset_index(drop=True)

# store the DataFrame as pickle (save the DataFrame as is instead of CSV)
df.to_pickle('df_traces_json')
# load the saved DataFrame in variable df_traces_json
```

```python
df = pd.read_pickle('df_traces_json')


# Print execution time of the code
end = time.time()
print('Time needed for execution: ', int(end - start), 'seconds.')
```

Python code for plots:

```python
#loading requred packages
import json
import pandas as pd
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
import os
import time

# monitor code execution time
start = time.time()

# df = pd.read_pickle('df_traces_json')
df = pd.read_pickle('df_traces_json_queue_finder')


def xt(index, timestamp, df=df):
    obj_id, t, t0, tn, x, y = df.iloc[index][:]
    if t0 <= timestamp < tn:
        for k in range(len(t)):
            if t[k] == timestamp:
                return x[k]
            if t[k] > timestamp:
                k = k - 1
                x_i = (x[k]
                       + (x[k + 1] - x[k])
                       * ((timestamp - t[k]) / (t[k + 1] - t[k])))
                return x_i
    else:
        return False


def yt(index, timestamp, df=df):
    obj_id, t, t0, tn, x, y = df.iloc[index][:]
    if t0 <= timestamp < tn:
        for k in range(len(t)):
            if t[k] == timestamp:
                return y[k]
            if t[k] > timestamp:
                k = k - 1
                y_i = (y[k]
                       + (y[k + 1] - y[k])
                       * ((timestamp - t[k]) / (t[k + 1] - t[k])))
                return y_i
    else:
        return False


```

```python
def lin_x_y_value(df, index, timestamp):
    '''Returns [x, y] coordinates for given index and timestamp as a list
    computed by linear interpolation if object exists during the timestamp.

    Parameters
    ----------
    df : pandas DataFrame
        A pandas DataFrame to perform the computations on.

    index : int
        Index of df to perform computation on.

    time : float
        Moment in time for [x, y] coordinate computation.

    Returns
    -------
    [x_i, y_i] : list
        List of length 2 with the interpolated x and y coordinate if
        the object at the given index exists at given timestamp

    False : boolean
        If object at given index doesn't exist at given timestamp.
    '''

    obj_id, t, t0, tn, x, y = df.iloc[index][:]
    if t0 <= timestamp < tn:
        for k in range(len(t)):
            if t[k] == timestamp:
                return [x[k], y[k]]
            if t[k] > timestamp:
                k = k - 1
                x_i = (x[k]
                       + (x[k + 1] - x[k])
                       * ((timestamp - t[k]) / (t[k + 1] - t[k])))
                y_i = (y[k]
                       + (y[k + 1] - y[k])
                       * ((timestamp - t[k]) / (t[k + 1] - t[k])))
                return [x_i, y_i]
    else:
        return False


def speed(index, timestamp, df=df, dt=0.5):
    '''returns the average speed in m/s of the object at the specified index
    and timestamp for the interval 2 * dt'''
    i = index
    t = timestamp
    if xt(i, t - dt) and xt(i, t + dt):
        vx = (0.001 * xt(i, t + dt) - 0.001 * xt(i, t - dt)) / (2 * dt)
        vy = (0.001 * yt(i, t + dt) - 0.001 * yt(i, t - dt)) / (2 * dt)
        v = np.sqrt((vx ** 2) + (vy ** 2))
        return v
    else:
        return False

```

```python

def b_speed(index, timestamp, df=df, dt=0.5):
    '''returns the average speed in m/s of the object at the specified index
    and timestamp for the interval dt'''
    i = index
    t = timestamp
    if xt(i, t - dt) and xt(i, t):
        vx = (0.001 * xt(i, t) - 0.001 * xt(i, t - dt)) / (dt)
        vy = (0.001 * yt(i, t) - 0.001 * yt(i, t - dt)) / (dt)
        v = np.sqrt((vx ** 2) + (vy ** 2))
        return v
    else:
        return False


def col_rg(h, col_lim=0.5):
    '''returns color red for 'h' < col_lim and green for 'h' >= col_lim'''
    if h >= col_lim:
        return 'g'
    if h < col_lim:
        return 'r'


def col_inf(h, a=0, b=2):
    '''returns color from colormap 'inferno' on scale of [a, b] for value
    of 'h', where a <= h <= b'''
    color = mpl.colormaps['inferno']
    return color((1 / (b - a)) * (h - a))


def rng(seed): return np.random.default_rng(seed).random()


color_hsv = mpl.colormaps['hsv']
def col(u, colormap=color_hsv): return colormap(rng(u))



def speed_plot_rg(index, df=df, dt=0.5, col_lim=0.5, t_len=30):
    obj_id, t, t0, tn, x, y = df.iloc[index][:]
    t_start = t[-1] - t_len
    plt.figure()
    for w in range(len(t) - 1):
        if t[w] > t_start:
            plt.plot(t_len - (t[w] - t_start),
                     speed(index, t[w], dt=dt),
                     'o',
                     c=col_rg(speed(index, t[w], dt=dt), col_lim=col_lim))
    plt.xlabel('Time [s]')
    plt.ylabel('Speed [m/s]')
    plt.title(f'Speed of object at index {index} from {t_len} s before
     ↪  entering the door,'
              '\n'
              rf'$2 \Delta t = {dt}$ s for speed computation')
    plt.ylim(0, 1.5)
    plt.xlim(t_len, 0)
    return

def speed_plot_rg_cont(index, df=df, dt=1.0, col_lim=0.5, t_len=20,
                       intvl=0.05, cl=[1, 8, 5, 12, 8, 15],
                       test=False):
    obj_id, t, t0, tn, x, y = df.iloc[index][:]
    t_start = t[-1] - t_len
    plt.figure()
    for w in np.arange(t_start, t[-1] + intvl, intvl):
        spd = b_speed(index, w, dt=dt)
        if spd:
            plt.plot(t_len - (w - t_start),
                     spd,
                     'o',
                     c=col_rg(spd, col_lim=col_lim))
    plt.xlabel('Time [s]')
    plt.ylabel('Speed [m/s]')
    plt.title(f'Speed of object at index {index} before '
              'entering the door at $t_{entry}$,'
              '\n'
              rf'$ \Delta t = {dt}$ s for speed computation, '
              'speed limit $v_{lim}$ '
              f'= {col_lim} m/s')
    plt.ylim(0, 1.5)
    plt.xlim(t_len, 0)

    plt.hlines(col_lim, 0, 20, colors='tab:gray', label=('$v_{lim} = $' +
    ↪ f'{col_lim} m/s'), linewidth=2)

    plt.hlines(col_lim - 0.1, cl[0], cl[1], colors='tab:orange',
    ↪ linewidth=2)
    plt.vlines(cl[0:2], col_lim - 0.14, col_lim - 0.06,
    ↪ colors='tab:orange',
              label=(f'Interval for condition 1: ' + '$v \leq v_{lim}$ in '
                ↪ + f'[-{cl[1]},-{cl[0]}]'), linewidth=2)

    plt.hlines(col_lim + 0.1, cl[2], cl[3], colors='tab:blue', linewidth=2)
    plt.vlines(cl[2:4], col_lim + 0.06, col_lim + 0.14, colors='tab:blue',
               label=f'Interval for condition 2: ' + '$v \geq v_{lim}$ in '
                ↪ + f'[-{cl[3]},-{cl[2]}]', linewidth=2)

    plt.hlines(col_lim - 0.1, cl[4], cl[5], colors='tab:purple',
    ↪ linewidth=2)
    plt.vlines(cl[4:], col_lim - 0.14, col_lim - 0.06, colors='tab:purple',
               label=f'Interval for condition 3: ' + '$v \leq v_{lim}$ in '
                ↪ + f'[-{cl[5]},-{cl[4]}]', linewidth=2)

    plt.xticks(np.arange(0, 22, 2),
               ['$t_{entry}$', '-2', '-4', '-6', '-8', '-10', '-12', '-14',
                ↪ '-16', '-18', '-20'])
    plt.legend(loc='upper left')
    if not test:
        if not os.path.isfile(f'plots/rg_plot_{index}.png'):
            plt.savefig(f'plots/rg_plot_{index}')
    if test:
        test_path = test + f'rg_plot_{index}'
```

```python
        plt.savefig(test_path)

    # plt.hlines(0.5, 8, 20)
    return


def speed_plot_line(index, df=df, dt=1.0, col_lim=0.5, t_len=20,
  ↪ intvl=0.05):
    obj_id, t, t0, tn, x, y = df.iloc[index][:]
    t_start = t[-1] - t_len
    xtim = []
    yspd = []
    for w in np.arange(t_start, t[-1] + intvl, intvl):
        spd = b_speed(index, w, dt=dt)
        if spd:
            yspd.append(spd)
            xtim.append(t_len - (w - t_start))
    plt.plot(xtim, yspd, label=f'Obj. index {index}')
    plt.xlabel('Time [s]')
    plt.ylabel('Speed [m/s]')
    plt.title(f'Speed of object from {t_len} s before entering the door,'
              '\n'
              rf'$\Delta t = {dt}$ s for speed computation')
    plt.ylim(0, 1.5)
    plt.xlim(t_len, 0)
    return


def speed_plot_distance(index, df=df, dt=1.0, col_lim=0.5, t_len=20,
  ↪ intvl=0.05):
    obj_id, t, t0, tn, x, y = df.iloc[index][:]
    t_start = t[-1] - t_len
    xdist = []
    yspd = []
    for w in np.arange(t_start, t[-1] + intvl, intvl):
        spd = b_speed(index, w, dt=dt)
        if spd:
            lok = lin_x_y_value(df, index, w)
            dist = np.sqrt((lok[0])**2 + (lok[1] + 500)**2)
            yspd.append(spd)
            xdist.append(dist)
    plt.plot(xdist, yspd, label=f'Obj. index {index}')
    plt.xlabel('Distance to door (at 0, -500) [mm]')
    plt.ylabel('Speed [m/s]')
    plt.title(f'Speed of object for distance to the door,'
              '\n'
              rf'$ \Delta t = {dt}$ s for speed computation')
    plt.ylim(0, 1.5)
    plt.xlim(5000, 0)
    return



# Print execution time of the code
end = time.time()
print('Time needed for execution: ', int(end - start), 'seconds.')
```

Python code for animation using linear interpolation:

```python
# loading required packages
import json
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import os
import time
from datetime import datetime
import matplotlib.animation as animation
import matplotlib as mpl
from matplotlib.lines import Line2D

# monitor code execution time
start = time.time()

# Load saved DataFrame
df = pd.read_pickle('df_traces_json_queue_finder')

# Load list of found queues (index numbers of objects defined as a queue)
queue_list = np.load('queue_list.npy')
long_queue_list = np.load('long_queue_list.npy')



def unix_from_datetime(day, hour, minute, second, year=2023, month='09'):
    '''Returns unix timestamp from given date and time parameters for GMT+2.
    day (int) = day of the month;
    hour (int) = hour of the day (24-hour format);
    minute (int) = the minute of the specified time;
    second (int) = the second of the specified time;
    year (int) = the year of the specified time;
    month (str) = month of the specified time in a two number format
    ↪ string'''
    # Create isoformat string from parameters
    if len(str(day)) == 1:
        day = f'0{day}'
    if len(str(hour)) == 1:
        hour = f'0{hour}'
    if len(str(minute)) == 1:
        minute = f'0{minute}'
    if len(str(second)) == 1:
        second = f'0{second}'
    isoform_str = f'{year}-{month}-{day}T{hour}:{minute}:{second}+02:00'
    unix_time = int(datetime.fromisoformat(isoform_str).timestamp())
    return unix_time


def lin_x_y_value():
    '''Returns [x, y] coordinates for given index and timestamp as a list
    computed by linear interpolation if object exists during the timestamp.

    Parameters
    ----------
    df : pandas DataFrame
        A pandas DataFrame to perform the computations on.

    index : int
        Index of df to perform computation on.

    time : float
        Moment in time for [x, y] coordinate computation.

    Returns
    -------
    [x_i, y_i] : list
        List of length 2 with the interpolated x and y coordinate if
        the object at the given index exists at given timestamp

    False : boolean
        If object at given index doesn't exist at given timestamp.
    '''

    # obj_id, t, t0, tn, x, y = df.iloc[index][:]
    if t0 <= timest < tn:
        for k in range(len(t)):
            if t[k] == timest:
                return [x[k], y[k]]
            if t[k] > timest:
                k = k - 1
                x_i = (x[k]
                       + (x[k + 1] - x[k])
                       * ((timest - t[k]) / (t[k + 1] - t[k])))
                y_i = (y[k]
                       + (y[k + 1] - y[k])
                       * ((timest - t[k]) / (t[k + 1] - t[k])))
                return [x_i, y_i]
    else:
        return False


def xt(index, timestamp, df=df):
    obj_id, t, t0, tn, x, y = df.iloc[index][:]
    if t0 <= timestamp < tn:
        for k in range(len(t)):
            if t[k] == timestamp:
                return x[k]
            if t[k] > timestamp:
                k = k - 1
                x_i = (x[k]
                       + (x[k + 1] - x[k])
                       * ((timestamp - t[k]) / (t[k + 1] - t[k])))
                return x_i
    else:
        return False

```

```python

def yt(index, timestamp, df=df):
    obj_id, t, t0, tn, x, y = df.iloc[index][:]
    if t0 <= timestamp < tn:
        for k in range(len(t)):
            if t[k] == timestamp:
                return y[k]
            if t[k] > timestamp:
                k = k - 1
                y_i = (y[k]
                       + (y[k + 1] - y[k])
                       * ((timestamp - t[k]) / (t[k + 1] - t[k])))
                return y_i
    else:
        return False


def speed(index, timestamp, df=df, dt=0.5):
    '''returns the average speed in m/s of the object at the specified index
    and timestamp for the interval 2 * dt'''
    i = index
    t = timestamp
    if xt(i, t - dt) and xt(i, t + dt):
        vx = (0.001 * xt(i, t + dt) - 0.001 * xt(i, t - dt)) / (2 * dt)
        vy = (0.001 * yt(i, t + dt) - 0.001 * yt(i, t - dt)) / (2 * dt)
        v = np.sqrt((vx ** 2) + (vy ** 2))
        return v
    else:
        return False


def col_rg(h, a=0, b=2):
    '''returns color from colormap 'inferno' on scale of [a, b] for value
    of 'h', where a <= h <= b'''
    # color = mpl.colormaps['inferno']
    # return color((1 / (b - a)) * (h - a))
    if h >= 0.5:
        return 'g'
    if h < 0.5:
        return 'r'


def rng(seed): return np.random.default_rng(seed).random()


color_hsv = mpl.colormaps['hsv']
def col(u, colormap=color_hsv): return colormap(rng(u))





def update(frame):
    '''Update function for animation.FuncAnimation(), returns scatter plot
    per requested frame;
    frame = frame number given by animation.FuncAnimation() for which a new
    scatterplot must be returned
    '''
    global obj_id, t, t0, tn, x, y, timest
    # Compute timestamp for the current frame, start_time (unix timestamp)
    # and interval (in milliseconds) are declared globally.
    timest = start_time + frame * interval * 0.001
    # Plot datetime information in title
    ax.set(title=f'{datetime.fromtimestamp(int(timest))} {int(timest)}')
    # Create list of object x-y coordinates.
    xy_cor_list = []
    # Create lis of object specific colors.
    ob_col_list = []
    # Create list of legend elements.
    legend_el = []
    # Add x-y coordinates to the list for every object present in the
    # timeframe of the animation.
    for i in obj_in_anim_idx_lst:
        obj_id, t, t0, tn, x, y = df.iloc[i][:]
        xy_cor = lin_x_y_value()
        # If object is present in frame.
        if xy_cor:
            legend_el.append(Line2D([0],
                                    [0],
                                    color='w',
                                    markerfacecolor=col(i),
                                    markeredgecolor='k',
                                    marker='o',
                                    label=f'{i}'))
            xy_cor_list.append(xy_cor)
            ob_col_list.append(rng(i))
    if xy_cor_list:
        data = np.array(xy_cor_list)
        # print(data)
        # Set new scatter plot data for the frame
        scat.set_offsets(data)
        scat.set_array(ob_col_list)
        ax.legend(handles=legend_el, loc=1)
        return (scat)
    else:
        scat.set_offsets(np.array([-10000, -10000]))
        scat.set_array([0])
        return (scat)


def make_animation(l_start_time=False,
                   year=2023,
                   month='09',
                   day=19,
                   hour=13,
                   minute=41,
                   second=4,
                   end_time=False,
                   duration=3,
                   max_duration=10,
                   l_interval=30,
                   fps=60,
                     save=True,
                     save_name=False,
                     background=False):

    # Declare list of global variables used in the 'update' function
    global start_time, interval, ax, obj_in_anim_idx_lst, scat
    # Declare real time starting point of the animation, year=2023 and
     ↪  month='09'
    day = day
    hour = hour
    minute = minute
    second = minute
    if not l_start_time:
        start_time = unix_from_datetime(day, hour, minute, second,
                                        year=year, month=month)
    else:
        start_time = l_start_time

    # declare animation file fps
    fps = fps
    # declare interval between frames in milliseconds (60 fps real time ~
     ↪  16,7 ms)
    interval = l_interval
    # end_time for object index list
    if end_time:
        total_frames = min(int((end_time - start_time) / (interval *
         ↪  0.001)),
                           fps * 60 * max_duration)
        start_time = end_time - total_frames * interval * 0.001
    if not end_time:
        # declare total number of frames
        total_frames = fps * 60 * duration
        end_time = start_time + total_frames * interval * 0.001


    # Create list of objects present during the timeframe of the animation
    obj_in_anim_idx_lst = []
    for i in range(len(df)):
        if (
                ((df.iloc[i]['first_timestamp'] <= start_time) and
                 (df.iloc[i]['last_timestamp'] > start_time)) or
                ((start_time <= df.iloc[i]['first_timestamp']) and
                 (end_time > df.iloc[i]['first_timestamp']))):
            obj_in_anim_idx_lst.append(i)
    # print(obj_in_anim_idx_lst)


    # Create figure as basis for the animation
    fig, ax = plt.subplots()
    ax.set(xlim=[-6000, 6000],
           ylim=[5000, -1000],
           title='Object locations in detection area',
           xlabel='x-coordinate in [mm]',
           ylabel='y-coordinate in [mm]')
    scat = ax.scatter(0,
                      0,
                          c=0.5,
                          cmap='hsv',
                          edgecolor='b',
                          s=400,
                          vmin=0,
                          vmax=1)

    if background:
        background_plot = plt.imread('permanent_sensor_view.png')
        plt.imshow(background_plot, extent=[-6000, 6000, 5000, -4000])
        # plt.hlines(0, -5000, 5000)
        # Create animation object

    ani = animation.FuncAnimation(fig=fig,
                                  func=update,
                                  frames=total_frames,
                                  interval=30)

    if save:
        save_name_str = 'time_animation.mp4'
        if save_name:
            save_name_str = save_name
        # Set filetype and save location
        writervideo = animation.FFMpegWriter(fps=fps)
        ani.save(save_name_str, writer=writervideo)
    return


for b in range(150):
    if b not in long_queue_list:
        print(f'Making animation for obj {b}')
        make_animation(l_start_time=df.iloc[b]['first_timestamp'],
                       year=2023,
                       month='09',
                       day=19,
                       hour=13,
                       minute=41,
                       second=4,
                       end_time=(df.iloc[b]['last_timestamp'] + 1),
                       duration=3,
                       max_duration=1,
                       l_interval=90,
                       fps=30,
                       save=True,
                       save_name=f'test_plots/animation_object_{b}.mp4',
                       background=False)

# Print execution time of the code
end = time.time()
print('Time needed for execution: ', int(end - start), 'seconds.')
```

Python code for queue identification:

```
#loading requred packages
# loading required packages
import json
import pandas as pd
```

```python
import numpy as np
import matplotlib.pyplot as plt
import os
import time
from datetime import datetime
import matplotlib.animation as animation
import matplotlib as mpl
from matplotlib.lines import Line2D

# monitor code execution time
start = time.time()

# Load saved DataFrame
df = pd.read_pickle('df_traces_json')
obj_id, t, t0, tn, x, y = df.iloc[0][:]


def xt(timestamp):
    if t0 <= timestamp < tn:
        for k in range(len(t)):
            if t[k] == timestamp:
                return x[k]
            if t[k] > timestamp:
                k = k - 1
                x_i = (x[k]
                       + (x[k + 1] - x[k])
                       * ((timestamp - t[k]) / (t[k + 1] - t[k])))
                return x_i
    else:
        return False


def yt(timestamp):
    '''returns y-coordinate from linear interpolation for given timestamp
     ↪  for
    the row information given through *kwargs'''
    if t0 <= timestamp < tn:
        for k in range(len(t)):
            if t[k] == timestamp:
                return y[k]
            if t[k] > timestamp:
                k = k - 1
                y_i = (y[k]
                       + (y[k + 1] - y[k])
                       * ((timestamp - t[k]) / (t[k + 1] - t[k])))
                return y_i
    else:
        return False


def b_speed(timestamp, dt=1):
    '''returns the average speed in m/s of the object at the specified index
    and timestamp for the interval dt using backward difference'''
    ti = timestamp
    if xt(ti - dt) and xt(ti):
        vx = (0.001 * xt(ti) - 0.001 * xt(ti - dt)) / (dt)
        vy = (0.001 * yt(ti) - 0.001 * yt(ti - dt)) / (dt)
        v = np.sqrt((vx ** 2) + (vy ** 2))
        return v
    else:
        return False


def intvl_cond(arr):
    global lim_e
    a = arr[0]
    b = arr[1]
    t_len = arr[2]
    below = arr[3]
    con_sp = []
    time_arr = np.arange(tn - a,
                         tn - b - intvl_dt,
                         -1 * intvl_dt)
    for j in time_arr:
        if b_speed(j):
            if below:
                if b_speed(j) <= cut_off:
                    con_sp.append(1)
                else:
                    con_sp.append(0)
            if not below:
                if b_speed(j) >= cut_off:
                    con_sp.append(1)
                else:
                    con_sp.append(0)
    for m in range(len(con_sp) - int(t_len / intvl_dt) + 1):
        if np.sum(con_sp[m:m+int(t_len / intvl_dt)]) * (intvl_dt / t_len) >
         ↪  valid:
            if not below:
                lim_e = max(lim_e, tn - time_arr[m+int(t_len / intvl_dt) -
                 ↪  1])
            return True


def cond_1(lim_t=5):
    '''returns True if direction was in negative y-direction for last lim_t
    seconds'''
    return (y[-1] - yt(tn - lim_t)) < 0


def set_lim():
    global lim_a, lim_b, lim_c, lim_d, lim_e, lim_f
    lim_a = 1
    lim_b = 8
    lim_c = 4.5
    lim_d = 11.5
    lim_e = 8
    lim_f = 15


set_lim()
```

```python

# Interval properties: [left, right, condition consecutive seconds,
#                        below cut_off = True / above = False]
a_b = [lim_a, lim_b, 2, True]
c_d = [lim_c, lim_d, 2, False]
e_f = [lim_e, lim_f, 0.5, True]


# Exclude objects that move away from the door or have less than the minimal
# required time to be in a queue.
drop_list = []
for i in range(len(df)):
    y = df.iloc[i]['y']
    t = df.iloc[i]['timestamp']
    # First condition: direction away from the door in its last 5 seconds
    # Second condition: total recording time of an object less than 9.5
     ↪  seconds
    if ((y[-1] - yt(t[-1] - 5)) > 0) or (t[-1] - t[0] < 9.5):
        drop_list.append(i)

df = df.drop(drop_list)
df = df.reset_index(drop=True)

# store the DataFrame as pickle (save the DataFrame as is instead of CSV)
df.to_pickle('df_traces_json_queue_finder')


breakout_flag = False
queue_list = []
cut_off = 0.35
intvl_dt = 0.1
valid = 0.9
st = 5000
amount = 20000
# for i in range(72339, 72339 + 100):
# for i in range(st, st + amount):
for i in range(len(df)):
    obj_id, t, t0, tn, x, y = df.iloc[i][:]
    # Set interval limits back to normal for this iteration
    set_lim()
    # First interval condition check
    if intvl_cond(a_b):
        # Second interval condition check
        if intvl_cond(c_d):
            # Third interval condition check
            # Only check third interval from the time the second interval
            # condition was met (by resetting e_f: lim_e adjusted by the
            # intvl_cond() function in the second interval)
            e_f = [lim_e, lim_f, 0.5, True]
            if intvl_cond(e_f):
                queue_list.append(i)
                print(f'Object {obj_id} at index {i} is in a queue')


np.save('queue_list_35_2sec.npy', queue_list)
# np.save('queue_list.npy', queue_list)
```

```



condition_limits = []
set_lim()
for i in [lim_a, lim_b, lim_c, lim_d, lim_e, lim_f]:
    condition_limits.append(i)
# np.save('condition_limits.npy', condition_limits)
np.save('condition_limits_test.npy', condition_limits)

# Print execution time of the code
end = time.time()
print('Time needed for execution: ', int(end - start), 'seconds.')
```

Appendix B
Smart-sensor output file example

Example of a smart-sensor output file in .JSON format

```
{
    "Sensor_ID#": {
        "save_file_name": {
            "traces": {
                "894": {
                    "timestamp": [
                        1695333029.154,
                        1695333029.234
                    ],
                    "x": [
                        -1361.0301557528662,
                        -1362.8495528511462
                    ],
                    "y": [
                        4215.283178260297,
                        4277.6178779225975
                    ],
                    "id": 894
                },
                "895": {
                    "timestamp": [
                        1695333029.154,
                        1695333029.234,
                        1695333029.314,
                        1695333029.394
                    ],
                    "x": [
                        -196.04632465787003,
                        -167.68817848709256,
                        -167.9797652631643,
                        -139.6042351907825
                    ],
                    "y": [
                        4305.051594243129,
                        4336.201764572378,
                        4430.185860171737,
                        4492.959136860266
                    ],
                    "id": 895
                }
            }
        }
    }
}
```

Reference list

Buchmüller, S., & Weidmann, U. (2006). *Parameters of pedestrians, pedestrian traffic and walking facilities* (tech. rep.). Swiss Federal Institute of Technology Zurich.

Daamen, W. (2004). *Modelling Passenger Flows in Public Transport Facilities* (Doctoral dissertation). TU Delft.

Daamen, W. (2023). Figure of Smart-sensor detection area provided by Winnie Daamen, TU Delft.

Duives, D. C., Daamen, W., & Hoogendoorn, S. P. (2015). Quantification of the level of crowdedness for pedestrian movements. *Physica A*, *427*(June), 162–180.

Fruin, J. J. (1971). *Pedestrian Planning and Design*. Metropolitan Association of Urban Designers; Environmental Planners.

Kneidl, A. (2016). How Do People Queue? A Study of Different Queuing Models. *Traffic and Granular Flow '15*.

Mullick, P., Appert-Rolland, C., Warren, W., & Pettre, J. (2022). Methods of density estimation for pedestrians moving in small groups without a spatial boundary.

Okazaki, S., & Matsushita, S. (1993). A Study of Simulation Model for Pedestrian Movement with Evacuation and Queuing. *International Conference on Engineering for Crowd Safety*, 271–280.

pandas. (2023). pandas documentation. https://pandas.pydata.org/docs/index.html#

Python Software Foundation. (2023a). json - JSON encoder and decoder. https://docs.python.org/3/library/json.html

Python Software Foundation. (2023b). Mapping Types - dict. https://docs.python.org/3/library/stdtypes.html#typesmapping

Steffen, B., & Seyfried, A. (2009). Methods for measuring pedestrian density, flow, speed and direction with minimal scatter. *Physica A*, (389), 1902–1910.

UnixTime.org. (2023). Epoch & Unix Timestamp. https://unixtime.org

van den Heuvel, J. (2022). *Mind your passenger! The passenger capacity of platforms at railway stations in the Netherlands* (Doctoral dissertation). TRAIL Research School.

Vuik, C., Vermolen, F. J., van Gijzen, M. B., & Vuik, M. J. (2016). *Numerical Methods for Ordinary Differential equations*. Delft Academic Press / VSSD.