How to use UAV-traffic-tool for Intersections

Konstantinos Pourgourides

Introduction

UAV-traffic-tool is a Python package that allows users to conduct in-depth analysis and visualization of UAV-based urban traffic data, with a focus on signalized intersections. In this document, a detailed walkthrough is provided on how the tool can be used properly *(with commands in the context of Python)* in order to extract valuable information and make insightful visualizations regarding signalized intersections, using the dataset *20181024_d2_0900_0930.csv*[1] from the pNEUMA experiment. This document follows closely the code provided in this usage example in the GitHub repository of the tool.

Contents

[1] This dataset contains an intersection of a certain architecture and topology. The document follows the example of the intersection of this particular dataset. In case of different datasets and intersection characteristics, the user should make the appropriate modifications.

1 Acquiring the data in the correct format

In order to use the tool properly in later stages, the first and most important task of all is to acquire the UAV-based traffic data we want to analyze in the correct format. Different datasets use different formats, and oftentimes, those are not very understandable or intuitive, for the sake of space saving. For this reason, the tool was designed to take the input data in only one, universal and well structured format, which will be explained in detail below. Thus, the user might have to do some data pre-processing before being able to successfully deploy the tool for their research.

Usually, UAV-based traffic datasets include information for the different vehicles in the experiment/recording, such as the id, the type τ (e.g.: car, motorcycle, bus), the position[2] x and y as a function of time, the instantaneous speed u and acceleration a as a function of time, etc. The correct format in which to give these data as an input to the tool, is to create a directory that contains different keys, where each key corresponds to a vector. Each vector, stores the information mentioned above, either in scalar values (such as vehicle ids and types, which remain constant throughout the dataset), or vectors (such as position, speed, acceleration, which evolve through time - denoted with bold letters in the document).

An example of the correct data format is[3]

```
data = { "id":id, "type":τ , "x":x, "y":y, "time":t, "speed":u }
```

The keys (i.e. vectors) in the data directory correspond to

```
data.get("id") =    [0, 1, ..., N]
data.get("type") =  [τ^0, τ^1, ..., τ^N]
data.get("x") = [x^0, x^1, ..., x^N]
data.get("y") = [y^0, y^1, ..., y^N]
data.get("time") = [t^0, t^1, ..., t^N]
data.get("speed") =  [u^0, u^1, ..., u^N]
```

where N+1 is the total number of vehicles in the dataset. For a vehicle with id i, the corresponding vector elements are

$$\texttt{data.get("time")[data.get("id").index(i)]} = \mathbf{t}^i = [t_j, t_{j+1}, \ldots, t_k]^i$$
$$\texttt{data.get("x")[data.get("id").index(i)]} = \mathbf{x}^i = [x(t_j), x(t_{j+1}), \ldots, x(t_k)]^i$$
$$\texttt{data.get("y")[data.get("id").index(i)]} = \mathbf{y}^i = [y(t_j), y(t_{j+1}), \ldots, y(t_k)]^i$$
$$\texttt{data.get("speed")[data.get("id").index(i)]} = \mathbf{u}^i = [u(t_j), u(t_{j+1}), \ldots, u(t_k)]^i$$

The time vector includes the time stamps when the vehicle was in the experiment recording, with respect to the beginning of the recording, which is usually set to 0 seconds. The rest of the vectors, include the values of the corresponding quantities at each time stamp. For each vehicle, the vectors should all have the same length, i.e. number of elements. This number is determined by the amount of time spent by the vehicle in the experiment recording and the refresh rate of the video camera of the UAV

$$|\mathbf{q}^i| = \texttt{camera refresh rate} \left[\frac{1}{sec}\right] \times (t_k^i - t_j^i)\,[sec]\,, \quad \forall \mathbf{q} \in \{\mathbf{x}, \mathbf{y}, \mathbf{t}, \mathbf{u}\}$$

When a UAV-based traffic dataset is converted to the format described in this section, the user is ready to use the tool to conduct their analysis and visualization tasks for an intersection (or link) of their choice. An example on how to do the above data transformations on the pNEUMA dataset exists at this location in the GitHub repository of the tool.

[2] Given with respect to a reference system, often the WGS84 system, which corresponds to longitude and latitude coordinates.

[3] Acceleration is not needed as an input to the tool, because it is not included in some datasets, and can be extracted later through time differentiation of speed, which is almost always provided.

2 Setting up the analysis

After the user has transformed the original data (also referred to as raw data) in the correct format by following the previous section, they can proceed further with analysis and visualization tasks.

2.1 Defining the intersection under study

Initially, the user must locate and familiarize with the intersection they want to work with by using satellite imagery or mapping software, for example GoogleMaps, and spot key information, such as the possible movement and turning directions, and the number of lanes. In this document, the signalized intersection between *Panepistimiou Ave. & Omirou Str.* in Athens, Greece is examined (Figure 1).

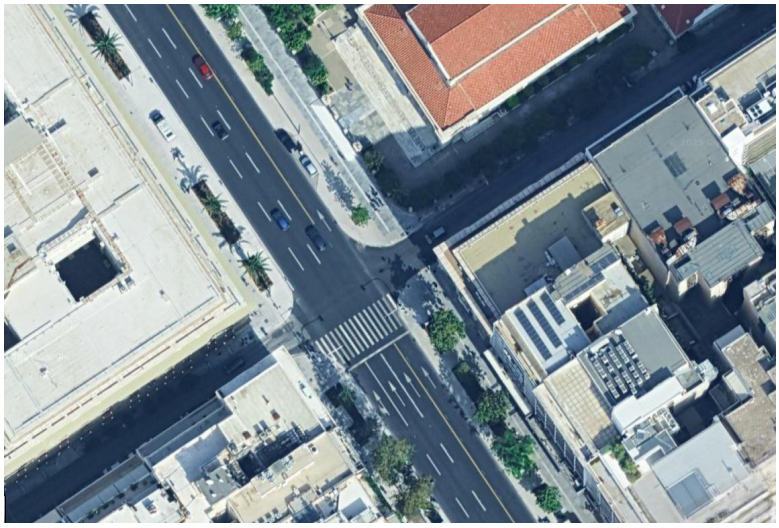

Figure 1: Intersection between Panepistimiou Ave. & Omirou Str. in Athens, Greece as seen through GoogleMaps

In this case, for Panepistimiou Ave., there are 4 lanes in total (3 normal + 1 bus lane), and drivers can either drive straight or turn to the left, and for Omirou Str., there is only one lane, and drivers can either drive straight or turn right. At the time when the dataset was collected (October 2018), Panepistimiou Ave. had 1 extra normal lane, making them 4 normal + 1 bus lane.

2.2 Acquiring some preliminary information about the intersection

Before proceeding further, the user must acquire some preliminary spatio-temporal information about the intersection, which will be a stepping stone for later.

The first task is to define the status of a boolean constant called **wgs** (word geodesic system), which will let the tool know whether it is about to handle geo-spatial (i.e. latitude and longitude coordinates) information, in case it is **True**, or position coordinates in another reference system (e.g.: coordinates for camera pixels), if it is **False**.

The next piece of information is the bounding box (**bbox**), which consists of four sets of spatial coordinates that define a box which encloses the intersection of interest. It is important to note that the box edges should intersect the roads perpendicularly, like in Figure 2. The bbox will help with reducing the dataset to include information only about the vehicles which belong to the area of interest, i.e. keep data for the vehicles that have traversed the intersection at least once at any point during the experiment recording. Data for the rest of the network will be discarded. The bbox is a Python list, and has the following form

```
bbox = [(ll_y,ll_x),(lr_y,lr_x),(ur_y,ur_x),(ul_y,ul_x)]
```

Where `ll,lr,ur`, and `ul` correspond to lower left, lower right, upper right, upper left respectively. If **wgs** is **True**, these coordinates can be picked up from GoogleMaps; if not, the user must provide appropriate coordinates according to the reference system for positions (e.g.: camera pixels). Also, the user must provide a Python tuple

with the center coordinates of the intersection, (y_c, x_c).

The last piece of preliminary information is a global time axis for the experiment. For example, if the recording lasted for 3 minutes, the global time axis should be a list that starts at 0 seconds and ends at 180 seconds. The way to obtain the first and last values of the time axis (i.e. the duration of the experiment) is the following

```
start = min(min(vector) for vector in t)
end = max(max(vector) for vector in t)
```

The length of the time axis (i.e. the update rate of measurements in it) can be set by the user to correspond the refresh rate of the measurements of the experiment, or any other integer multiple of that. Also, the decimal points in the values of the time axis should correspond to that of the original measurements from the experiment. Finally, the time axis can be defined through the following commands in Python

```
time_axis = np.arange(start,end+update_period,update_period)
time_axis = list(np.round(time_axis,decimals=correct_decimals))
```

Once all the above information is gathered, it should be stored in a dictionary called `spatio_temporal_info` in the following fashion

```
spatio_temporal_info =
    {
    "wgs" : wgs,
    "bbox" : bbox,
    "intersection center" : intersection_center
    "time axis" : time_axis
    }
```

The above directoy, along with the data directory defined in Section 1 consist the 2 arguments passed to the tool classes in later stages of the analysis.

3 Analysis & Visualization

3.1 Loading the tool

To load the tool, the user should run the following commands

```
import mytool.uav_traffic_tool as uav
tool = uav.Wiz()}
```

In order to load classes and methods further down the analysis, simply run the following commands

```
class =  tool.class_name(*args)
method = class.method_name(*args)
```

3.2 Filtering the data

Before proceeding with the rest of the analysis, the user can optionally apply some filters on the data, which essentially filter out the vehicles which spent more than 95% of their time in the experiment immovable. These vehicles are considered to be parked on the pavement or the side of the road and are hence removed. This is achieved by running the following command

```
data = tool.dataloader(raw_data,spatio_temporal_info).get_filtered_data()
```

Optionally, the user can also pass an argument called `cursed_ids`, where they can list any vehicle ids that they desire to explicitly remove from the dataset, even if they are not filtered above.

3.3 Categorizing the trajectories

The first task is to group trajectories based on their origin (o) and destination (d), i.e. entry and exit points within the intersection. This will be important for later steps of the analysis when the user wants to separate between different od pairs to extract, for example, the different traffic light phases.

To achieve this, the intersection is split into 4 triangles using the center point and the bbox edges as provided by the user in the `spatio_temporal_info` directory. Each triangle is labeled with numbers 1 through 4, starting from the lower part of the intersection and proceeding clockwise. For each vehicle, an od pair is assigned; for example, if a vehicle enters the intersection within triangle 1 and exits it within triangle 3 (i.e. driving straight on Panepistimiou ave.), then the assigned od pair will be **(1,3)**. This is achieved by running the following commands

```
analysis = tool.analysis(data,spatio_temporal_info)
od_pairs = analysis.get_od_pairs()
```

In the intersection under examination, the correct od pairs are **(1,3)** (Driving straight on Panepistimiou Ave.), **(1,2)** (Initially driving on Panepistimiou Ave. and then turning left towards Omirou Str.), **(4,3)** (Initially driving on Omirou Str. and then turning right towards Panepistimiou Ave.), and finally **(4,2)** (Driving straight on Omirou Str.).

The user can visualize the trajectories of the different od groups (Figure 2) with different colors for clarity by running the following commands

```
visualization = tool.visualization(data,spatio_temporal_info)
visualization.draw_trajectories_od(valid_od_pairs)
```

The argument `valid_od_pairs` is a list that must contain the correct od pairs, i.e. [**(1,3),(1,2),(4,3),(4,2)**] in this case. This is done to avoid visualizing any vehicles that have unexpected od pairs. This can happen, for example, if a motorcycle is driving in the opposite direction of normal traffic on a pavement.

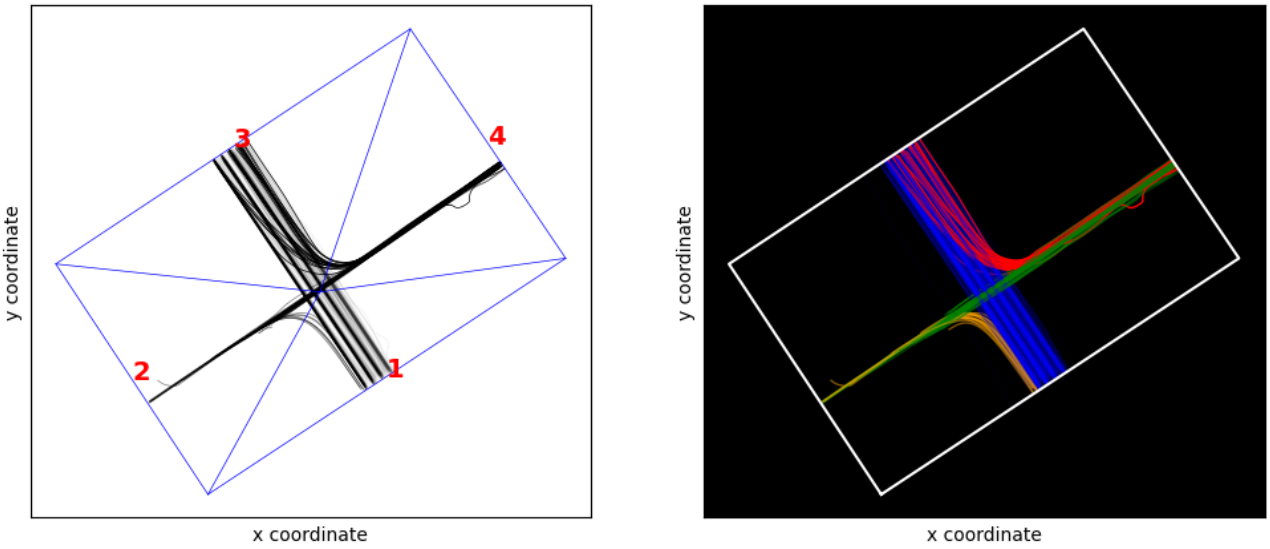

Figure 2: Trajectory categorization and visualization. Color correspondence: **(1,3)**-blue, **(1,2)**-gold, **(4,3)**-green, **(4,2)**-red

3.4 Separating the data based on od pairs

The next step is to separate the data into smaller sub-directories based on the different trajectory origins. Essentially, the objective is to isolate data which belong to the same traffic light phase, i.e. have the same origin within the intersection. Their destination need not be the same. In this example, the user must make two data sub-directories for od pairs [**(1,3),(1,2)**] and [**(4,3),(4,2)**] respectively. To do this, run these commands

```
data_13_12 = analysis.get_od_data(desirable_pairs=[(1,3),(1,2)])
data_43_42 = analysis.get_od_data(desirable_pairs=[(4,3),(4,2)])
```

In the following steps of the analysis and visualization process, the methods are applied separately on the two sub-directories. To do this, the user should activate the analysis and visualization classes with the following commands

```
analysis_13_12 = tool.analysis(data_13_12,spatio_temporal_info)
analysis_43_42 = tool.analysis(data_43_42,spatio_temporal_info)
visual_13_12 = tool.visualization(data_13_12,spatio_temporal_info)
visual_43_42 = tool.visualization(data_43_42,spatio_temporal_info)
```

3.5 Extracting lane-wise information

The next task is to extract lane-wise information for the components of the intersection, i.e. Panepistimiou Ave. and Omirou Str.. As many datasets do not include any lane-wise information, the following part of the subsection is focused on extracting it through data-driven methods.

The first step is to identify the number of lanes in each street. In order to do this, the user must first identify the flow direction in each one. If the flow direction is upwards or downwards, then a vector **b** is defined between the lower and upper left edges of the bbox, and if the flow direction is rightwards or leftwards, the vector is defined between the lower right and lower left edges of the bbox

$$\mathbf{b} = \begin{cases} (ul_x - ll_x)\hat{\lambda} + (ul_y - ll_y)\hat{\varphi}, & \texttt{if flow direction is up or down} \\ (lr_x - ll_x)\hat{\lambda} + (lr_y - ll_y)\hat{\varphi}, & \texttt{if flow direction is left or right} \end{cases}$$

where $\hat{\lambda}, \hat{\varphi}$ are the unit vectors in the longitudinal and latitudinal directions respectively. Then, for a vehicle with id i in the street, and for each of its time steps, $[t_j, \ldots, t_k]^i$, vectors $[\mathbf{a}(t_j), \ldots, \mathbf{a}(t_k)]^i$ are defined from the lower left edge of the bbox to its center. In detail, for a time step t_j^i

$$\mathbf{a}^i(t_j) = \left(x^i(t_j) - ll_x\right)\hat{\lambda} + \left(y^i(t_j) - ll_y\right)\hat{\varphi}$$

This is done for every vehicle in the street. Using this information for a vehicle with id i in the street, the perpendicular distance from its center to the appropriate bbox edge, $[\delta(t_j), \ldots, \delta(t_k)]^i$, is calculated for each of its time steps. In detail, for a time step t_j^i

$$\delta^i(t_j) = \frac{1}{|\mathbf{b}|}\left(\mathbf{a}^i(t_j) \times \mathbf{b}\right)$$

This distribution of δ is expected to be centered around a common point for all vehicles within the same lane. Thus, if the user plots this distribution, for all the vehicles in the same street, and for all their individual time steps, it is expected for distinct peaks to appear at the center of each lane. By doing this, the user can identify the number of lanes as well as their spatial extend, and finally, the lane in which every vehicle in the street belongs to, for each of their time steps. To execute the previous steps with the tool, the user must initially run the following commands

```
lane_info_13_12 = analysis_13_12.get_lane_info(flow_direction="up")
lane_info_43_42 = analysis_43_42.get_lane_info(flow_direction="left")
```

By doing this, the user will see the distribution of δ, as in Figure 3, in the case of Panepistimiou Ave.:

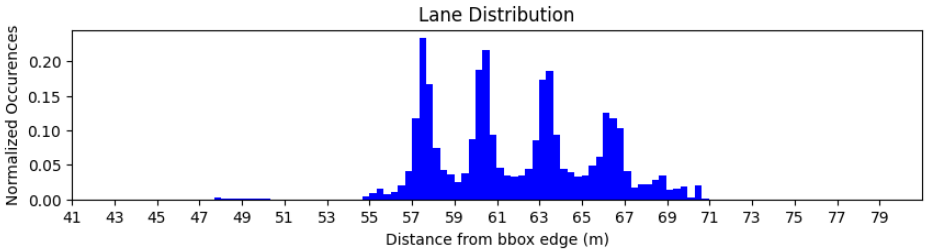

Figure 3: Initial distribution of δ for Panepistimiou Ave.

The user will be asked to input the number of lanes (in this case 5), and the lower and higher limits of the distribution (in this case 55 and 70 respectively). Subsequently, the tool will cluster the bins, as in Figure 4

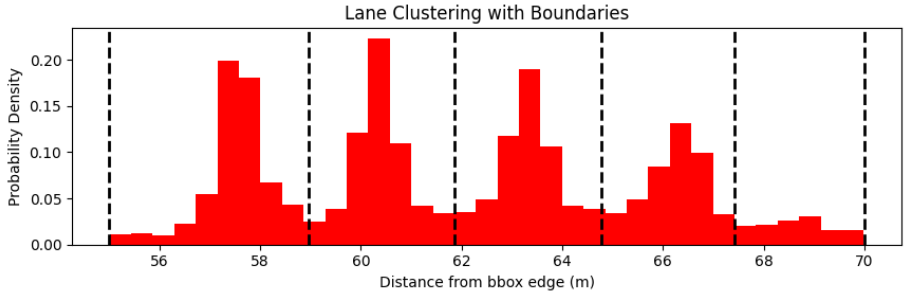

Figure 4: Distribution of δ for Panepistimiou Ave. after user inputs and clustering

This process allows for the calculation of the spatial boundaries of the different lanes. Following, for every vehicle, the corresponding lane will be stored in a vector for each of its time steps. For a vehicle with id i, the vector will be $\boldsymbol{\ell}^i = [\ell(t_j), \ldots, \ell(t_k)]^i$.

The results of `lane_info_13_12, lane_info_43_42` are dictionaries with all the lane-wise information. For example,

```
lane_info_13_12  =
{
"number" = 5,
"boundaries" =  [55,...,70] ,
"distribution" =  [ℓ^0,...,ℓ^N]
}
```

3.6 Extracting traffic light phases & cycles

3.6.1 Traffic Light Phases

Another useful task when it comes to signalized intersections is studying the different phases for different traffic lights, i.e. when the light becomes green, for how long it stays green, when it turns red, and when it goes to green again. In order to achieve that for a certain traffic light, a number of virtual detectors (equal to the number of lanes) are placed in the appropriate spatial coordinates (if **wgs** is **True**, these can be extracted from GoogleMaps). The detectors measure flow (count hits) per moment of the time axis. Initially, the user should type these commands

```
flow_info_13_12 = analysis_13_12.get_flow_info(detector_positions_13_12)
flow_info_43_42 = analysis_43_42.get_flow_info(detector_positions_43_42)
```

Where the `detector_positions` arguments are lists with the coordinates (y, x) of the flow detectors. The result of the `get_flow_info` methods is a dictionary of dictionaries, where each nested dictionary has the keys `"time stamp"`, `"flow"` and `"id"`. The key `"time stamp"` denotes the moment of measurement with respect to the time axis, `"flow"` denotes how many counts were registered between the current and previous time stamp, and `"id"` is a list of the ids of the vehicles responsible for the registered count hits (the list is empty if 0 counts were registered between the current and previous time stamp). For example,

```
flow_info_13_12 =
{
 {"time stamp": 0.0, "flow": 0, "id": []},
  ⋮
 {"time stamp": 24.0, "flow": 4, "id": [71,128,142,145]},
  ⋮
 {"time stamp": 900.0, "flow": 0, "id": []}
}
```

The next step uses the above information to alter the flow detectors output into binary (1 if there were registered count hits, 0 if there were not), and group them together. The idea here is that the different groups correspond to the different phases. The user must run

```
flow_13_12,normalized_flow_13_12 = analysis_13_12.get_normalized_flow(threshold)
flow_43_42,normalized_flow_43_42 = analysis_43_42.get_normalized_flow(threshold)
```

The `threshold` arguments refer to the maximum accepted distance (in time) in order to consider count hits in the same group. The result of `flow` is a list with the unnormalized hits per time axis entry, and `normalized_flow`

is a list with the normalized (0 or 1) and grouped hits per time axis entry.

To understand the above, the result of plotting `flow_13_12, flow_43_42` with respect to the time axis is in Figure 5

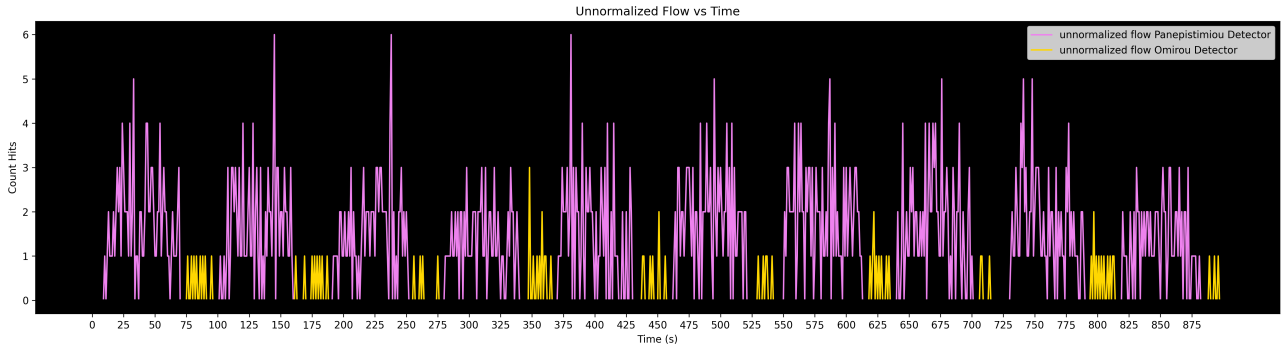

Figure 5: Plot of `flow_13_12, flow_43_42` with respect to the time axis

In Figure 5, the ungrouped and unnormalized count hits are shown. To see the grouped and normalized count hits, the user should plot `normalized_flow_13_12,normalized_flow_43_42` with respect to the time axis, as in Figure 6

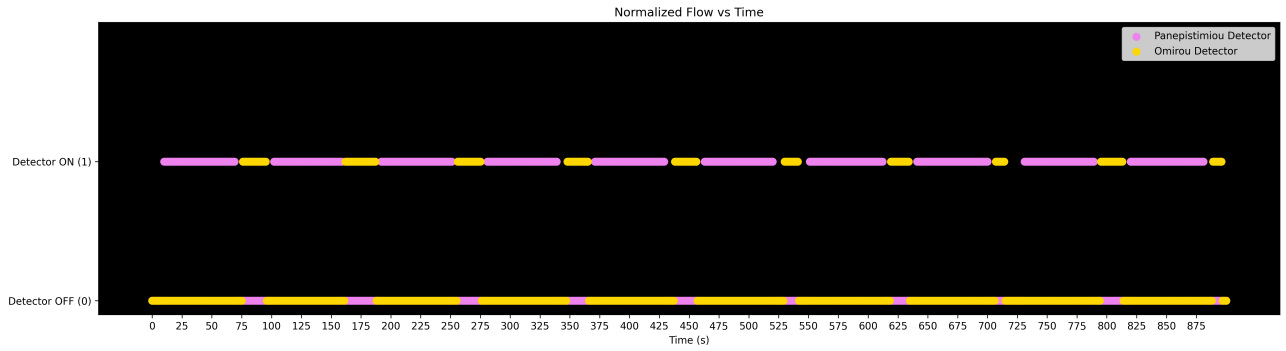

Figure 6: Plot of `normalized_flow_13_12, normalized_flow_43_42` with respect to the time axis

To translate Figure 6 in formal traffic light phase data, the user must run the following commands

```
tlp_13_12 = analysis_13_12.get_traffic_light_phases()
tlp_43_42 = analysis_43_42.get_traffic_light_phases()
```

The output of `tlp_13_12, tlp_43_42` is a list of dictionaries, where each dictionary has the following keys: `"Green","Duration ON","Red","Duration OFF","Phase Duration"`, which correspond to the moment the respective traffic light turned green, the duration of green, the moment it turned red, the duration of red, and the entire phase duration respectively. For example,

```
tlp_13_12[0] =
{
"Green": 10.0,
"Duration ON": 60.0,
"Red": 70.0,
"Duration OFF": 32.0,
"Phase Duration": 92.0
}
```

If the UAV recording had stopped before the completion of a phase, the appropriate keys will have the value `None`. For example, if the recording stopped while the traffic light was red, the key `"Duration OFF"` cannot be calculated, and the same is true for key `"Phase Duration"`.

3.6.2 Traffic Light Cycles

In order to combine the information of `tlp_13_12, tlp_43_42` to get the information on the full cycles, i.e. the subsequent completion of the 2 phases, the user must run the command

```
cycles = analysis.get_traffic_light_cycles(tlp_13_12,tlp_43_42)
```

The output of `cycles` is a list of dictionaries, where each dictionary has the following keys: `"Start","Break", "Continue","Stop","End"`, which correspond to the moment the cycle started (i.e. the first phase started), the moment the cycle stops temporarily (i.e. the first phase goes into red), the moment the cycle restarts (i.e. the second phase started), the moment the cycle stops (i.e. the second phase goes into red), and finally, the moment the cycle restarts (i.e. the first phase has re-started) respectively. For example,

```
cycles[0] =
{
"Start": 10.0,
"Break": 70.0,
"Continue": 76.0,
"Stop": 96.0,
"End": 102.0
}
```

3.7 Extracting queue-wise information

The last task performed in this document is the extraction of queue-wise information. The queue characteristics are calculated for each lane, and for each traffic light phase of each street. Initially, the user must run

```
sorted_id_13_12 = analysis_13_12.get_sorted_id()
sorted_id_43_42 = analysis_43_42.get_sorted_id()

gaps_13_12 = analysis_13_12.get_gaps()
gaps_43_42 = analysis_43_42.get_gaps()
```

These commands help with extracting the sorted vehicle ids and the respective gaps between them (front bumper to rear bumper) for every moment of the time axis, which will later help with extracting the queue-wise information, such as the queue length. The output of both methods is a list of dictionaries, where each dictionary has the following keys: `"time stamp", "lane 0",..,"lane L"`, where L+1 the total number of lanes. The time stamp is the time axis entry, and each lane-specific key is a list of the sorted ids or gaps. For example,

```
sorted_id_13_12[time_axis.index(tlp_13_12[7].get("Green"))] =
{
"time stamp": 551.0,
"lane 0": None,
"lane 1": [1778],
"lane 2": [1922, 2021],
"lane 3": [2045, 1659],
"lane 4": [2062]
}
```

```
gaps_13_12[time_axis.index(tlp_13_12[7].get("Green"))] =
{
"time stamp": 551.0,
"lane 0": None,
"lane 1": [-1.0],
"lane 2": [9.4, -1.0],
"lane 3": [14.7, -1.0],
"lane 4": [-1.0]
}
```

The value None is for when there are no vehicles in the lane, and the value -1.0 corresponds to lane leaders that have no vehicle in front of them.

The queue characteristics use the above outputs, and are calculated when the queue has its maximum potential, i.e. when the corresponding traffic light turns green. In order for a vehicle in a specific lane, and in a specific street to be considered as part of the queue, it should satisfy the 3 following conditions:

- **The vehicle is physically behind the traffic light.** The traffic light position for each lane is taken according to the detector positions, which were given as arguments to the corresponding `get_flow_info()` method previously.
- **The vehicle has speed lower than a given threshold**, provided by the user.
- **The gap between the vehicle and its leader is smaller than a given threshold**, provided by the user.

The above are achieved when the user runs the following commands

```
queue_info_13_12 = analysis_13_12.get_queue_info(speed_threshold, gap_threshold)
queue_info_43_42 = analysis_43_42.get_queue_info(speed_threshold, gap_threshold)
```

The output of the above methods is a list of lists, where each nested list corresponds to a traffic light phase, and has L+1 dictionaries inside, where L+1 the total number of lanes. Each nested dictionary has the following keys: `"Lane","Queued Vehicles","Queue Length","Queued IDs","Dissipation Duration"`, which correspond to the lane in question, the number of queued vehicles at the time of green light, the queue length in meters, the ids of the queued vehicles, and the queue dissipation duration in seconds, respectively. The latter is the time it takes for all the queued vehicles to move past the green light pole. If only a part of the queue dissipates before the light turns red again, the queue dissipation duration is equal to the duration of the green light. For example,

```
queue_info_43_42[6] =
[{
"Lane": 0,
"Queued Vehicles": 6,
"Queue Length": 38.0,
"Queued IDs": [2192, 2166, 2139, 2067, 2029, 1784],
"Dissipation Duration": 11.0
}]
```

This concludes the intersection pipeline usage example of the tool.

4 General Comments

As stated in the introductory section, this usage example corresponds to a specific dataset, for a specific intersection, under specific conditions. When a user applies the tool to a different dataset, they should be aware to make the appropriate changes or modifications to the inputs. The form of the output will always be the same, but the specifics will vary depending on the situation. The tool is under continuous development, and if any user has any questions or suggestions for fixes and improvement, they should open an issue in the GitHub repository of the tool.