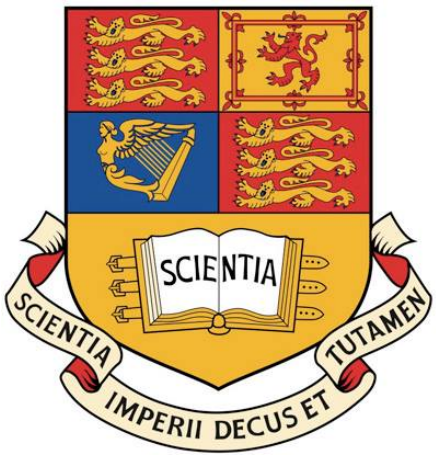
Imperial College London

Department of Electrical and Electronic Engineering

MEng Final Year Project 2019



Project Title:	**Learning for smart transportation: preventing loop solutions in reduced dimension origin-destination flow estimation**
Student:	**Jacopo Bacchelli**
CID:	**01112853**
Course:	**EEE4T**
Project Supervisor:	**Dr. Wei Dai**
Second Marker:	**Prof. Kin K. Leung**

Acknowledgments

I would first like to express my gratitude to my supervisor, Dr. Dai, for his advice, guidance and patience throughout the year.

My sincere thanks also go to Jingyuan Xia, whose help and feedback were crucial during the final steps of this project.

Finally, a special mention goes to my family and friends for their continued and unconditional support.

Abstract

A novel framework in origin-destination flow estimation was recently developed to reduce the complexity of inquired flows from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$. This project is focused on improving and extending the framework by incorporating elements of real-world transportation networks. A potential shortcoming is identified in the solver, that may converge to solutions containing loops despite the fact that only loop-free path flows are assumed to be generated. A necessary and sufficient condition on the variables of the optimisation problem is found to guarantee the estimated flows are generated via shortest path assignment, thus preventing the solver from converging to solutions containing loops. The modified framework is then extended to real link costs, which allow for a wider range of applications to real world transportation networks. It is found that by considering shortest-path assignment, the relative error of estimated OD flows is considerably reduced in both the single-step and multi-step traffic models, for rigid and real link costs.

Contents

List of Symbols **iii**

1 Introduction **1**

2 Background **3**
2.1 Traffic models . . . 3
2.1.1 OD flow based models . . . 3
2.1.2 Path flow based models . . . 5
2.1.3 O flow based models . . . 5
2.1.4 Non-linear models . . . 6
2.2 Comparison of traffic models . . . 6
2.3 Traffic assignment . . . 10
2.4 The inverse problem: OD estimation from link flow counts . . . 10
2.4.1 Zoning . . . 11
2.4.2 Traffic count locations . . . 12
2.4.3 Solving the inverse problem . . . 13

3 Analysis and design **19**
3.1 Original framework's description and issues . . . 19
3.2 Definition and characterisation of loop flows . . . 20
3.3 Extending the flow constraint to prevent U-turn loops . . . 21
3.4 Introducing destination flows as a possible improvement . . . 23
3.5 A sufficient condition for the absence of loop flows . . . 26
3.6 Shortest path assignment: matching solver and generator . . . 28
3.7 Introduction of non-unitary, unequal link costs . . . 30
3.7.1 Relation to real networks . . . 30
3.7.2 Modified mapping and constraints . . . 30
3.8 Applicability of O flow based frameworks . . . 34

4 Implementation **37**
4.1 Generator . . . 37
4.2 Solver . . . 39
4.2.1 Optimisation algorithm . . . 40
4.2.2 Constraints representation . . . 40
4.3 Testing . . . 41

5 Results **43**
5.1 Measuring performance . . . 43
5.2 Simulations . . . 44

6 Evaluation **50**

7 Conclusion **52**

References **53**

Appendices **56**
A Integer link costs: mapping and constraints 56
B Python code 57

List of Symbols

$a_{ij,od}$	Fraction of s_{od} flowing through link (i, j)
A	OD flow traffic assignment matrix
c_{ij}	Cost to travel on link (i, j)
$\boldsymbol{c}$	Link cost vector $= [\ldots, c_{ij}, \ldots]$
$\mathcal{C}_S$	Set of conditions, or constraints, on the solver
$\mathcal{C}_G$	Set of conditions, or assumptions, on the generator
$\delta_{ij,\boldsymbol{p}}$	Fraction of $s_{\boldsymbol{p}}^{\text{path}}$ that flows through link (i, j)
Δ	Path incidence matrix
$f_{i,o}$	Cost of shortest path from o to i
F	Shortest path matrix
$g_k^{i,j}$	Set of links that form the k^{th} path between OD pair (i, j)
$g^{i,j}$	Set of links in paths between i and j, given by the set union of all $g_k^{i,j}$
$\mathcal{G}$	Network
$\gamma_{ij,o}$	Number of times flow originated in o is counted in link (i, j)
Γ	Link count matrix
$\mathcal{L}$	Set of links in a network
M_t	Level set of the map $\mu(t)$: $M_t = \{t' : \mu(t') = \mu(t)\}$
$\mathcal{N}$	Set of nodes in a network
n_l	Number of links in a network
n_n	Number of nodes in a network
n_T	Number of samples in the observation time horizon $= \frac{\text{observation time horizon}}{\text{sampling time interval}}$
n_{OD}	Number of valid OD pairs in a network
$p_{ij,o}$	Fraction of x_o flowing through link (i, j)
P	O flow assignment matrix
$\boldsymbol{p}$	Path defined as a series of nodes
$\mathcal{P}_{od}$	Set of paths between origin o and destination d
$s_{\boldsymbol{p}}^{\text{path}}$	Path flow over path $\boldsymbol{p}$
$\boldsymbol{s}^{\text{path}}$	Path flow vector $= [\ldots, \boldsymbol{s}_{\boldsymbol{p}}^{\text{path}}, \ldots]$
$q_{ij,d}$	Fraction of z_d flowing through link (i, j)
Q	D flow assignment matrix
s_{od}	OD flow between o and d

$\boldsymbol{s}$	OD flow vector $= [\ldots, s_{od}, \ldots]$
t_s	Duration of one sampling time interval
T^-	Node arrival matrix $= \lceil F \rceil$
T^+	Node departure matrix $= \lfloor F + 1 \rfloor$
$\tau_{\max}$	Maximum path length
u	Average number of links per node
x_o	O flow originating in o
$\boldsymbol{x}_o$	O flow vector $= [\ldots, x_o, \ldots]$
y_{ij}	Flow between link i and link j
$\boldsymbol{y}$	Link flow vector $= [\ldots, y_{ij}, \ldots]$
z_d	D flow ending in d
$\boldsymbol{z}$	D flow vector $= [\ldots, z_d, \ldots]$

1. Introduction

Travelling entities in any given network (i.e. vehicles for transportation networks, bits for communication networks etc.) start their itinerary in origin nodes and may choose different paths to reach the destination nodes they are headed to. OD estimation consists in deriving the OD flows, that is, the traffic volume corresponding to each possible origin-destination pair, by observing flows through links in the network. This results in better understanding of the entities' movements, and is the inverse of the traffic modelling and assignment problem, where given a set of assumptions and characteristics of the network topology and of the entities' behaviour, we attempt to analytically determine the traffic volume in each link at a certain time.

OD estimation has been an active research area for several decades. [1, 2, 3, 4, 5, 6, 7, 8]. It is widely accepted that travel demands can be safely inferred from the estimated OD flows: as a consequence, these are particularly useful in contexts such as urban planning, for both long and short-term projects. New roads need to be constructed in order to improve the overall connectivity of nodes in the network, while works and modifications need to be planned such that viable alternative routes can be easily devised. Disruption caused by unexpected changes in the network or traffic conditions can also be mitigated or minimised by knowing the traffic demand.

The following paragraph provides a mathematical formulation of the problem. Consider a network $\mathcal{G} = \langle \mathcal{N}, \mathcal{L} \rangle$, where $\mathcal{N}$ refers to the set of nodes (vertices) in the network, and $\mathcal{L}$ refers to the set of links (edges). Let $n_n = |\mathcal{N}|$ and $n_l = |\mathcal{L}|$ be the number of nodes and links in the network, respectively. Let $\boldsymbol{y} \in \bar{R}^{n_l}$ denote the observed link flows, where $\bar{R} := R^+ \bigcup \{0\}$. Given the topology of network $\mathcal{G}$ and the observed link flows $\boldsymbol{y}$, our purpose is to find the origin-destination matrix $\boldsymbol{S} \in \bar{R}^{n_n \times n_n}$, where each element $\boldsymbol{S}_{o,d}$ denotes the volume of flow originating at node o and terminating at node d. Unless prior constraints are introduced in the model or the optimisation problem formulation, the OD flows produced can be assumed to be generated by any route assignment algorithms or models [6], which may include linear or non-linear models.

Several inherent challenges are present when dealing with OD estimation.

- The problem can be thought of as a generally ill-posed linear system, especially when complex real road networks are involved: as such, there is rarely a unique solution matrix $\boldsymbol{S}$ that explains the link flows $\boldsymbol{y}$. The choice between possible solutions depends on the availability and relevance of historical data, and on assumptions made about the nature of the flows.

- When a solution, that we denote $\hat{\boldsymbol{S}}$, is picked, one needs to evaluate its accuracy. This is fairly easily done if synthetic data is used (i.e. the flows are fully simulated), but in

the case where real-world data is employed, the absence of a "true" OD matrix requires other evaluation methods.

- The mathematical model derived to represent the relationship between observed flows and OD flows will have considerable impacts on the types of practical scenarios that can be studied and on the complexity of the optimisation problem. A trade-off is required between the number of inquired flows and the information obtained, and between the assumptions made and the applicability of the model to real world networks.

The principal aim of this project is to improve the framework developed in the paper "Dimension reduction for Origin-Destination Flow Estimation: Blind Estimation Made Possible" [8]. Regarding applicability to real-world situations, the current constraints imposed on the problem result in ideal scenarios whose settings are relatively simple and do not translate some fundamental aspects of road networks. From a more simulation oriented perspective, some discrepancies between the flow generator and the solver were identified: in particular, given the original constraints, the optimisation algorithm might converge to solutions containing loop flows that could not have been generated. Addressing such issues would be highly beneficial for the framework. It was thus decided to study loop flows and see if, and how, generator and solver could possibly be matched. Once this analysis is completed, elements of more realistic networks could be incorporated in the model and new constraints formulated.

This report is structured as follows: a literature review is included in section 2, covering background material on traffic assignment models and OD estimation techniques. Section 3 will present the work that was conducted in this project to match solver and generator in an O flow based framework, the modification that was carried out on the assignment in order to achieve this, and the extension of the model to non-unitary link costs. Decisions were made regarding research directions to be pursued: the trade-offs considered and the thinking process behind these choices are also detailed here. The section ends with a qualitative discussion of the applicability of O flow based models to simple networks, where user behaviour is taken into account by assigning physical meaning to every node. The software implementation of the framework and the technical challenges encountered in the process are described in section 4. Simulation results are presented, commented and interpreted in section 5, while section 6 contains a critical evaluation of the work accomplished. Lastly, section 7 contains conclusive remarks and outlines potential for further work.

2. Background

This background review will focus on OD estimation. As was previously mentioned, this problem is the inverse of a traffic assignment task: indeed, before directly finding the OD flows that best explain the traffic counts, it is important to determine which traffic model is employed. Literature in this area has also been studied, but only for models relevant to the current aims of the project or to foreseeable extensions.

2.1 Traffic models

Traffic models determine the volume of vehicles flowing through a link, given the demand and supply models of the network. Complete reviews and explanations of traditional approaches can be found in [9, 10]. The assignment can be deterministic or stochastic, done for uncongested networks or following a user-equilibrium model, and may also assume static or dynamic flows and static or dynamic assignment.

2.1.1 OD flow based models

The most commonly used model in OD estimation literature consists of basic OD flows and a static assignment matrix A to explain observed link flows. [11, 2, 5, 12, 6] In these works, demand flows and assignment are assumed to be static, and the following *assignment relationship* is derived:

$$\boldsymbol{y} = A\boldsymbol{s} \tag{2.1}$$

where $\boldsymbol{y} \in \mathbb{R}^{n_l}$, $A \in \mathbb{R}^{n_l \times n_{OD}}$ and $\boldsymbol{s} = \text{vec}(\boldsymbol{S}) \in \mathbb{R}^{n_{OD}}$ is the vectorised OD matrix (i.e. the vector formed by stacking the columns of the matrix).

Remark 1. *In qualitative explanations and while presenting methods, we will refer interchangeably to the OD matrix ($\boldsymbol{S}$) or to the OD flows ($\boldsymbol{s}$) to denote the set of all origin-destination flows. When a distinction is required, it will be made clear from the context or the notation used.*

It can be noted that an upper bound on n_{OD} is given by the number of possible origin-destination pairs n_n^2 (including trips originating and terminating at the same node). Each element $a_{ij,od}$ of matrix A denotes the fraction of traffic originating in node o and directed to node d that flows through link (i, j), or equivalently, between nodes i and j. These types of models are referred to as *single-step*, or *static*, and are generally utilised for long-term planning projects where we are interested in an average of the demand and where the system may be assumed to be in steady-state[13].

On the other hand, for short-term strategies like route guidance or traffic analysis after unexpected perturbations in the network, another type of dynamic models needs to be employed. At this point, it is useful to introduce the following terms:

- The *sampling time interval* denotes a single time period over which traffic is observed (or over which data is collected)
- The *sampling time horizon* denotes the total observation period, the sum of all consecutive time intervals

In the single-step model, the link flows $\boldsymbol{y}$ correspond to the total trips travelling through links of the network over a sampling time interval that is sufficiently long (a work-day or a whole week, for example). *Multi-step* models, used for example in [9, 14, 15] for dynamic estimation, provide finer resolution and lead to the following assignment relationship:

$$\boldsymbol{y}^t = \sum_{k=1}^{t} A^{t,k} \boldsymbol{s}^k \tag{2.2}$$

where $a_{ij,od}^{k,t}$ denotes the fraction of the OD flow between o and d originated during time interval k and passing through link ij during time interval t.

Remark 2. *In equation (2.2) above, OD flows generated before the first interval are assumed to be negligible. A positive bias will be introduced in the OD estimates in the initial interval, since these will be used to explain all traffic present in the network at the start of the observation period. The effect of this assumption can be relaxed with an estimate of the traffic demand before the study period.*

Another formulation of the multi-step model can be found in [8, 9] and is given by:

$$\boldsymbol{y}^t = \sum_{\tau=1}^{\tau_{\max}} A^{\tau} \boldsymbol{s}^{t-\tau+1} = A^t * \boldsymbol{s}^t \tag{2.3}$$

where $a_{ij,od}^{\tau}$ denotes the fraction of the OD flow between o and d passing through link ij, τ intervals after the OD flow departed from o. The definition of A^t implies that its elements are non-zero only for $1 \leq t \leq \tau_{\max}$, equivalently between the interval where the OD flows are originated and the interval corresponding to the maximum trip time. Equation (2.3) could thus be rewritten as:

$$\boldsymbol{y}^t = \sum_{\tau=-\infty}^{\infty} A^{\tau} \boldsymbol{s}^{t-\tau+1} = A^t * \boldsymbol{s}^t \tag{2.4}$$

where $*$ denotes a convolution.

The model given by (2.3) involves a loss of generality over the one in (2.2), notably: the assignment matrix is chosen to be static, or time-independent, and determined only by the difference $\tau = t - k$ between the time interval of interest and the time interval when the OD flows were originated. This assumption on assignment, which can equivalently be formulated as $A^{k,t} = A^{t-k} \quad \forall k \forall t$, will be used throughout most of this report and the "simplified" form of

the multi-step model will be employed. Equivalence between the general and simplified models holds if the network is uncongested and link travel times are constant throughout the whole sampling time horizon (i.e. all vehicles travel at the same constant speed from start to end of their trip).

2.1.2 Path flow based models

Observed flows can also be related to the paths taken by vehicles travelling in the network [5, 7]. A path $\boldsymbol{p}$ is defined as a sequence of links that are traversed to go trough a sequence of nodes $o \to i \to j \to \ldots \to d$. Let us denote the flow along a particular path $\boldsymbol{p}$ by $s_{\boldsymbol{p}}^{\text{path}}$, and the vector of all path flows as $\boldsymbol{s}^{\text{path}} \in \mathbb{R}^{n_{\text{path}}}$. The resulting model is given by:

$$\boldsymbol{y} = \Delta \boldsymbol{s}^{\text{path}} \tag{2.5}$$

where Δ is the path incidence matrix, whose elements $\delta_{ij,\boldsymbol{p}}$ correspond to the proportion of flow traversing path $\boldsymbol{p}$ that flows through link (i, j) and:

$$\delta_{ij,\boldsymbol{p}} = \begin{cases} 1, & \text{if } \boldsymbol{p} \text{ contains link } (i,j) \\ 0, & \text{otherwise} \end{cases}$$

The multi-step equivalent of (2.5) is:

$$\boldsymbol{y}^t = \Delta^t * \boldsymbol{s}^{\text{path},t} \tag{2.6}$$

Elements of Δ^t can be easily computed following the same rules as for model (2.5) if the time sampling interval is greater than or equal to the maximum trip time. Otherwise, it can also be computed assuming constant speed and an uncongested network.
As we will see later, this is the formulation that gives the most information on the actual movement of vehicles within the network, but is also the one with the highest complexity in terms of flows considered to explain the system's behaviour.

2.1.3 O flow based models

An O flow based model, which considerably reduces the complexity of inquired flows, was recently used in [8] to solve the inverse problem. The total traffic generated from each viable origin node o is employed to explain the observed link flows and to uniquely derive the corresponding OD flows. In the single-step case, the following model is obtained:

$$\boldsymbol{y} = P\boldsymbol{x} \tag{2.7}$$

where $x_o = \sum_{d \neq o} x_{od}$ denotes the total traffic originated in node o, $\boldsymbol{x}$ denotes the O flow vector and P denotes (with a slight abuse of terminology) the assignment matrix whose elements $p_{ij,o}$ correspond to the proportion of traffic originated in node o that flows through link ij. Equation (2.7) is relevant whenever the sampling time interval is greater than or equal to the maximum trip time. Otherwise, the multi-step model must be employed:

$$\boldsymbol{y}^t = \sum_{\tau=1}^{\tau_{\max}} P^\tau \boldsymbol{x}^{t-\tau+1} = P^t * \boldsymbol{x}^t \tag{2.8}$$

OD flows can be uniquely derived from O flows by noting that the traffic volume between node o and node d is equal to the difference between the inflow in d that was originated from o and the outflow from d that was originated from o. Expressed mathematically, this gives:

$$s_{od} = x_o \left(\sum_i p_{id,o} - \sum_j p_{dj,o} \right) \tag{2.9}$$

Applying the same argument to (2.8) yields:

$$s_{od}^t = x_o^t \sum_{\tau=1}^{\tau_{\max}} \left(\sum_i p_{id,o} - \sum_j p_{dj,o} \right) \tag{2.10}$$

Path flows, on the other hand, cannot be derived from O flows or OD flows, as will be shown later. This is not problematic however, as the object of interest is the demand in the network, which can be clearly estimated from OD flows.

2.1.4 Non-linear models

So far, observed link flows have been assumed to be separable, that is: every traffic count y_{ij}^t can be expressed as a linear combination of demand flows. Some demand flows will have a certain proportion of them flowing through link ij at time t. Separability implies that if this proportion is equal to zero for all time intervals, then variations in the corresponding demand flow does not affect link flow y_{ij}^t. Non-separability would instead imply that observed link flows vary non-linearly as a function of demand flows that do not pass directly through them. This is intuitively explained by saying that, for example, if a demand flow is particularly high, there might be congestion spillback from a certain link (k, l) to another link (i, j), even though the actual demand flow does not usually travel via (i, j). Non-linear models have been developed [14, 16, 17] to account for congestion-related effects, but are considerably more complex and will not be investigated further.

The three first classes of models we have introduced relate observed link flows to OD flows, path flows and O flows, and for each of these relationships a single-step and a multi-step model can de defined and used, based on the nature of the problem to be solved. In the next section, the connections and differences between the models will be analysed.

2.2 Comparison of traffic models

The purpose of this section is to identify the connections between the assignment and incidence matrices that map inquired flows to link flows in the different models presented above, but also to study the complexity of the strategies and the information that can be obtained from each of them.

Let $\mathcal{P}_{od}$ be the set of paths that start in o and terminate in d. The OD flow model can be derived from the path flow model by expressing OD flows in terms of path flows

$$s_{od} = \sum_{\boldsymbol{p} \in \mathcal{P}_{od}} s_{\boldsymbol{p}}^{\text{path}} \tag{2.11}$$

and by expressing the assignment matrix A in terms of the path incidence matrix Δ and path flows as follows:

$$a_{ij,od} = \frac{\sum_{\boldsymbol{p} \in \mathcal{P}_{od}} \delta_{ij,\boldsymbol{p}} \cdot s_{\boldsymbol{p}}^{\text{path}}}{\sum_{\boldsymbol{p} \in \mathcal{P}_{od}} s_{\boldsymbol{p}}^{\text{path}}} \tag{2.12}$$

We refer to an example given in [8] to present the type of information that is lost when dealing with OD flows instead of path flows.

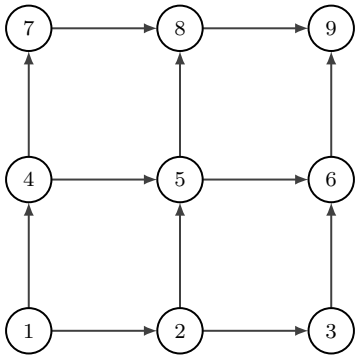


Figure 2.1: Diagram representation of a 3 by 3 unidirectional network

Let us define the following paths: $\boldsymbol{p}_1 = 1 \to 2 \to 5 \to 6 \to 9$, $\boldsymbol{p}_2 = 1 \to 2 \to 5 \to 8 \to 9$, $\boldsymbol{p}_3 = 1 \to 4 \to 5 \to 6 \to 9$, $\boldsymbol{p}_4 = 1 \to 4 \to 5 \to 8 \to 9$. All correspond to the same OD-pair $(1, 9)$. Using a single-step model for simplicity, let us set $x_{19} = 2c_l$ and $a_{ij,19} = 0.5$ if link ij is part of any of the four paths defined above, 0 otherwise. Equation (2.1) gives us $y_{ij} = c_l$ if link ij is part of any of the four paths, 0 otherwise. The OD flow based model gives us information on the distribution of s_{19} among the links in the network, but it is not possible to derive path flows uniquely from it. For instance, $s_{\boldsymbol{p}_1}^{\text{path}} = s_{\boldsymbol{p}_4}^{\text{path}} = c_l$ and $s_{\boldsymbol{p}_2}^{\text{path}} = s_{\boldsymbol{p}_3}^{\text{path}} = 0$, or $s_{\boldsymbol{p}_1}^{\text{path}} = s_{\boldsymbol{p}_4}^{\text{path}} = 0$ and $s_{\boldsymbol{p}_2}^{\text{path}} = s_{\boldsymbol{p}_3}^{\text{path}} = c_l$, or any convex combination of these two cases are all valid sets of path flows that correspond to the traffic demand and assignment matrix.
Qualitatively, one may say that by using an OD flow based model, we retain only which individual links or roads are taken by vehicles travelling from a point to another, but we have no information on the full ordered sequences of links (i.e. paths) taken by the individual vehicles to reach their destination.

Similarly, O flows and the assignment matrix P can be derived from OD flows and A as shown by equations (2.13) and (2.14), but information is lost in the process.

$$x_o = \sum_{d \neq o} s_{od} \tag{2.13}$$

$$p_{ij,o} = \frac{\sum_{d \neq o} a_{ij,od} \cdot s_{od}}{\sum_{d \neq o} s_{od}} \tag{2.14}$$

Using the network given in figure 2.1 again, let us consider another example present in [8]. Define four new paths: $\boldsymbol{p}_5 = 1 \to 2 \to 5 \to 6$, $\boldsymbol{p}_6 = 1 \to 4 \to 5 \to 6$, $\boldsymbol{p}_7 = 1 \to 2 \to 5 \to 8$, $\boldsymbol{p} = 1 \to 4 \to 5 \to 8$. Let us set $x_1 = 2c_l$ and $p_{ij,1} = 0.5$ if link ij is part of any of the four paths defined above, 0 otherwise. Equation (2.7) gives us $y_{ij} = c_l$ if link ij is part of any of the four paths, 0 otherwise. It is possible at this point to derive the OD flows involved, using equation (2.9), which gives us$s_{16} = s_{18} = c_l$. However, we do not know how these are distributed among individual links anymore.

Let us introduce the quantity $u \in \mathbb{Z}^+$, corresponding to the average number of links per node in network $\mathcal{G}$, in order to study the complexity of each model, focusing in particular on the number of inquired flows and non-trivial assignment (or incidence) matrix elements that are needed to describe traffic. These aspects, included in table 2.1, show the trade-off between model complexity and information about the vehicles' movement in a given network.

Remark 3. *By non-trivial matrix elements, we refer to components that cannot be directly inferred from the topology of the network. For instance, the path incidence matrix for the single-step model and for the simplified multi-step model (i.e. assuming constant speed and an uncongested network) can be constructed by observing the network, and no information about the flows is needed.*

Model	**Complexity of inquired flows**	**Non-trivial matrix elements**	**Information obtained**
O flow based	$n_O \in \mathcal{O}(n_n)$	$n_l n_O \in \mathcal{O}(n_l n_n)$	O flow volumes, map from O flows to link flows, OD flow volumes
OD flow based	$n_{OD} \in \mathcal{O}(n_n^2)$	$n_l n_{OD} \in \mathcal{O}(n_l n_n^2)$	All of the above, map from OD flows to link flows
Path flow based	$n_{\text{path}} \in \mathcal{O}(n_n^2 \cdot u^{n_n})$	0	All of the above, path flow volumes, map from path flows to link flows

Table 2.1: Comparison of single-step models' complexity and relevance

One last point worth discussing is the assumptions on the temporal behaviour of flows and assignment, which can either be *static* or *dynamic*. The exact definition of dynamic OD flows is more nuanced than simple time-invariance: if one assumes static assignment, it actually depends on the model used (i.e. the type of assignment/incidence matrix employed), as we are going to show.

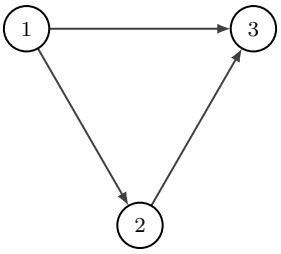


Figure 2.2: Diagram representation of a simple unidirectional network

Considering the network presented in figure 2.2, let us define the following paths: $\boldsymbol{p}_9 = 1 \to 2$, $\boldsymbol{p}_{10} = 1 \to 3$, $\boldsymbol{p}_{11} = 1 \to 2 \to 3$. Assume that pairs $1, 2$ and $1, 3$ are valid origin-destination pairs in the network, that flows are dynamic and that assignment is static.
If we neglect congestion and assume that all vehicles travel at the same constant speed, we can derive the path incidence matrices for $1 \leq t \leq \tau_{\max} = 2$ directly from the network:

$$\Delta^1 = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}; \Delta^2 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

At this point, for the traffic assignment matrix A^t to be static, one needs to impose a constraint on OD flows, notably: although each OD flow may fluctuate in volume, the proportions of path choice for each OD pair must remain constant throughout the sampling time horizon. This is not the case for path incidence matrices, for instance, and the difference can be explained mathematically by equation (2.12): in the case of time-varying path flows, the ratio $s_{\boldsymbol{p}}^{\text{path},t} / \left(\sum_{\boldsymbol{p} \in \mathcal{P}_{od}} s_{\boldsymbol{p}}^{\text{path},t} \right)$ should remain constant for all t, for all $\boldsymbol{p} \in \mathcal{P}_{od}$ and for all sets of paths $\mathcal{P}_{od}$ in order for the elements of the assignment matrix to remain constant. In the case of figure 2.2's network, assignment is given by:

$$A^1 = \begin{bmatrix} 1 & a_{12,13}^1 \\ 0 & a_{13,13}^1 \\ 0 & 0 \end{bmatrix}; A^2 = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & a_{23,13}^2 \end{bmatrix}$$

Assuming s_{13}^t vehicles leave node 1 towards 3 in $1 \leq t \leq n_T$: for all t, $a_{13,13}^1 s_{13}^t$ vehicles will choose $\boldsymbol{p}_{10}$ and the other $a_{12,13}^1 s_{13}^1 = (1 - a_{13,13}^1) s_{13}^1$ will choose $\boldsymbol{p}_{11}$.

Finally, a static P^t involves a further constraint on the dynamic flows: the proportions of destination choice for each origin must remain constant throughout the sampling time horizon. Mathematically, this is explained by equation (2.14), following a similar argument as in the previous paragraph, and again, A^t and Δ^t may be assumed static even though destination choice varies for each sampling time interval. Using the same example as before:

$$P^1 = \begin{bmatrix} p_{12,1}^1 \\ p_{13,1}^1 \\ 0 \end{bmatrix}; P^2 = \begin{bmatrix} 0 \\ 0 \\ p_{23,1}^2 \end{bmatrix}$$

Not only vehicles directed to node 3 will follow the same distribution among paths 10 or 11,

but also vehicles leaving node 1 will choose destinations 2 or 3 with fixed proportions for all the observation period.

By reducing the dimensionality of inquired flows, constraints need to be imposed on the dynamic flows for the mapping matrices to remain static. It is stated in [7], for example, that considering path flows allows for destination choice and path choice to be included in the model, but that the high dimensionality makes the approach viable only in situations where full path enumeration is feasible (e.g. in highway networks and urban settings where the path choice is limited).

2.3 Traffic assignment

Traffic over a network can be assigned following different models: this is directly relevant to OD estimation as the exact set of constraints imposed on the assignment matrix and flow vector will depend on which traffic assignment is assumed. Researchers in the transportation community decide to follow different strategies: some decide to employ path flows in order to accommodate for any form of assignment [7]. In this particular case, the authors reduced the dimensionality of the inquired flows by exploiting temporal patterns and by removing paths that are 2 kilometres longer than the shortest equivalent path. The extreme case of this second dimensionality reduction leads to a shortest path assignment, which has been extensively studied in the past decades [18, 19], and has been used for OD estimation with modifications to make it more realistic (e.g. the shortest path is always selected, but the notion and perception of shortest path varies among users [20]). One last traffic assignment that was found in the literature is the so called user equilibrium [10], still employed in order to solve the inverse problem[21]. Via this type of assignment, link flows are found based on known and complete OD information by solving the following minimisation problem:

$$\min z(\boldsymbol{y}) = \sum_{a} \int_{0}^{y_a} t_a(\omega) d\omega \tag{2.15}$$

subject to some constraints on path flows and link flows, where $t_a(\omega)$ denotes the travel time on link a and y_a denotes the flow on link a. User equilibrium takes link congestion into account, but in the basic formulation described above it is assumed that the link travel time depends on the respective link flow only, and not of other link flows. Relaxing this assumption leads to non-linear models described in subsection 2.1.4.

2.4 The inverse problem: OD estimation from link flow counts

OD estimation from flow counts, sometimes referred to in the literature as Origin Destination Count Based Estimation (ODCBE), [22, 9] involves a series of practical and theoretical considerations that depend on the nature of the network and traffic that is being studied. We will mainly focus on theoretical aspects that have to do with the methods used to cope with the

ill-posedness of the problem, the formulation of the function to be optimised and the constraints introduced.

Methods for OD estimation can be divided into three categories, as is explained in [2].
The first class is the so called 'transport demand model' approach and consists in applying sub-models that predicts the number of trips made by a certain mode (e.g. car, train, walking etc.) for a certain purpose (shopping, work, school etc.) during a certain period of time. This is done via large-scale home interviews and data processing that are time-consuming and expensive, which makes this method relevant to long term transport planning projects such as developing new towns.
The second category estimates OD flows by directly conducting road-side interviews or license plate surveys. Nevertheless, the former method inevitably disrupts road users while the latter requires considerable resources in terms of manpower and data processing.
The third method has increasingly attracted researchers' attention in the transportation community for the past four decades, as it results in cost-efficient, automated estimations that are useful even for contexts such as real-time traffic management schemes: it consists in using traffic counts directly and is the category studied in this project, which we will now present in more detail.
Most of the practical aspects of OD estimation go beyond the scope of this project, but the concept of zoning will be quickly introduced as it constitutes a non-negligible part of the problem in real-world settings.

2.4.1 Zoning

Zones are needed to determine the set of virtual nodes $\mathcal{N}_v$ and virtual links $\mathcal{L}_v$ among, respectively, nodes $\mathcal{N}$ and links $\mathcal{L}$ in network $\mathcal{G}$, and simply correspond to subsections of the total study area. The virtual nodes used in the models are generally (but not necessarily) the centroids of the different zones and thus do not always correspond to physical, accessible points in the transportation network. Theoretically, trips may start and end anywhere in the road network (with some limitations, of course) and the purpose of zones is to subdivide the study area into a discrete set of points with finite cardinality. The action of dividing the network under consideration into zones is called *zoning*, and has been traditionally carried out by a *domain expert* [10] and therefore at least partly manually defined. Today, effort is being made to automate this task: for example, a zoning algorithm [6] has been recently developed to determine the optimal zones in a road network. The two objective criteria to optimise were chosen as follows: the subset of nodes chosen should be as small as possible (the zoning was "fine-grained" as the centre of each zone was an individual node in the network, without any grouping of nodes involved), and the travel between pairs of nodes in the subset should account for most of the flow observed on the links. Mathematically, this is equivalent to finding the Pareto optimal point for the bi-criteria objective:

$$\min_{\boldsymbol{b}\in\{0,1\}^{|\mathcal{N}|}} \left\{ ||\boldsymbol{b}||_0, ||\boldsymbol{y} - A^{(\boldsymbol{b})}\boldsymbol{s}^{(\boldsymbol{b})}||_2^2 \right\}$$

where $\boldsymbol{b}$ indicates which nodes in $\mathcal{N}$ are chosen, and $A^{(\boldsymbol{b})}, \boldsymbol{s}^{(\boldsymbol{b})}$ correspond respectively to the truncation of $A, \boldsymbol{x}$ based on the new subset of $\mathcal{N}$ (defined by the non-zero elements of $\boldsymbol{b}$). This new technique was then tested and resulted in superior performance (for prediction of OD flows) when compared to manual zoning carried out by a domain expert.

Remark 4. *In this project, as we have done in the previous section, we will not make an explicit distinction between the studied network $\mathcal{G} = \langle \mathcal{N}, \mathcal{L} \rangle$ and its virtual counterpart $\mathcal{G}_v = \langle \mathcal{N}_v, \mathcal{L}_v \rangle$, as we are not directly concerned with zoning. Also, there will be no explicit distinction between the set of links $\mathcal{L}$ and the set of links for which flow counts are available $\mathcal{L}_c \subseteq \mathcal{L}$.*

2.4.2 Traffic count locations

An essential part of OD estimation from flow counts lies in the choice of traffic counting locations, where a trade-off is made between cost (due to the number of locations and sensors used) and collected data: the more traffic counts are available, the more precise the estimate will be. Four basic rules were introduced by Yang and Zhou [23] to determine the optimal number of points and their location:

1. OD covering rule: counting points are located so that a portion of any OD flow is observed
2. Maximal flow fraction rule: for a particular OD pair, counting points should be located at links so that the proportion of link flow due to the OD flow is maximised
3. Maximal flow-intercepting rule: links chosen for traffic count should intercept as many flows as possible
4. Link independence rule: counting points should be located so that traffic counts are linearly independent

At this point, after traffic count locations have been set and data is available for collection, let us study the dimensionality of the problem more thoroughly. This is usually carried out in the literature for the particular model that is chosen for traffic modelling and taking into account specific constraints imposed on the network (see [7, 8]), but we will present it for the general case of OD flow based models (2.1) and (2.3). Assuming the assignment matrix is fixed and known a priori, the number of unknowns for the single-step model corresponds to n_{OD} OD flows, while for the simplified multi-step model we have $n_T n_{OD}$ unknowns, where n_T is the number of samples in the observation time horizon. Without any prior constraints, the number of linearly independent equations available is n_l for the single-step model and $n_T n_l$ for the multi-step model. In both cases, we have $n_l < n_{OD}$ and $n_T n_l < n_T n_{OD}$: the problem is *ill-posed* as it does not admit a unique solution. If A needs to be estimated as well, this introduces $n_l n_{OD}$ unknowns in the single-step model and $\tau_{\max} n_l n_{OD}$ in the multi-step one. Additionally, the number of links for which flow counts are available is usually smaller than the total number of links, in practice, which makes the resolution even more complex. A reformulation is needed to pick one of the possible estimates of the OD matrix: this is generally done via the introduction of additional assumptions which increases the number of

independent equations, via a dimensionality reduction of inquired flows which reduces the number of unknowns, or via inclusion of historic data.

2.4.3 Solving the inverse problem

After having devised appropriate zoning of the study area and obtained data from the traffic count locations, let us now look at different methods for OD matrix estimation from flow counts, and start by giving a general formulation of the optimisation problem. The objective is to find the OD matrix such that the observed link flows are as close as possible to the link flows generated via the assignment of the estimated OD flows. This is given by:

$$\min_{\boldsymbol{s}\geq 0} z_1\big(\boldsymbol{y}(\boldsymbol{s}), \boldsymbol{y}\big) = \min_{\boldsymbol{s}\geq 0} z_1(A\boldsymbol{s}, \boldsymbol{y}) \tag{2.16}$$

where z_1 is a distance function and $\boldsymbol{y}(\boldsymbol{s})$ are the link flows predicted by an assignment of the demand to the network.

In the seventies, researchers initially employed the transport demand models described above [24, 25] by combining equation (2.1) for trip purpose m with the standard gravity model (see, for example, [26], p. 5-7 for an overview), which gives the following equation in its most general form:

$$y_{ij}^m = \sum_m \sum_o \sum_d O_o^m \cdot D_d^m \cdot A_o^m \cdot B_d^m \cdot f_{od}^m \cdot a_{ij,od}^m \tag{2.17}$$

where the superscript m corresponds to a certain trip purpose, O_o^m and D_d^m denote the trips originated in o and attracted in d, respectively, for trip purpose m (O_o is the equivalent of x_o in the single-step model, used here for clarity as destination flows are also present), A_o^m and B_d^m are balancing factors and f_{od}^m is the deterrence function. However, these techniques actually require some additional data about zones in order to calibrate the parameters, which makes them less reliable for external-external trips [1] (i.e. trips originating and destined outside the study area).

Because there might be several demand flows that explain the observed link flows with the same level of accuracy, it is very common to include prior information from e.g. surveys to further reduce the ill-posedness of the problem. Equation (2.16) can be rewritten as follows:

$$\min_{\boldsymbol{s}\geq 0} w_1 z_1(A\boldsymbol{s}, \boldsymbol{y}) + w_2 z_2(\boldsymbol{s}, \boldsymbol{s}^H) \tag{2.18}$$

where w_1, w_2 are weighting factors relative to the uncertainty of counts and historic data respectively, z_2 is a distance function, $\boldsymbol{s}^H$ denotes the historic OD flow vector and $\boldsymbol{s} \geq 0$ means that $\boldsymbol{s}$ is in the non-negative orthant (i.e. all elements of $\boldsymbol{s}$ are non-negative). Historic data is generally introduced when using statistical approaches (via parametric estimation techniques) like Maximum Likelihood (ML), Generalized Least Squares (GLS) and Bayesian Inference (BI). A clear overview of these methods can be found in [13, 9]. Note that statistical inference does not require historic survey data: all the methods below could be employed without any prior

knowledge $\boldsymbol{s}^H$, thus only estimating $\boldsymbol{s}$ with respect to $\boldsymbol{y}$.
A *Maximum Likelihood* estimator is obtained by maximising the likelihood of obtaining the observed link flow counts and the historic survey data given a set of demand flows. It can be expressed as:

$$\begin{aligned}
\boldsymbol{s}^{ML} &= \arg\max_{\boldsymbol{s}\in S} L(\boldsymbol{s}^H, \boldsymbol{y}|\boldsymbol{s}) \\
&= \arg\max_{\boldsymbol{s}\in S} L(\boldsymbol{s}^H|\boldsymbol{s}) \cdot L(\boldsymbol{y}|\boldsymbol{s}) \quad \text{(data in the two sets is statistically independent)} \\
&= \arg\max_{\boldsymbol{s}\in S} \ln\big(L(\boldsymbol{s}^H|\boldsymbol{x}) \cdot L(\boldsymbol{y}|\boldsymbol{s})\big) \\
&= \arg\max_{\boldsymbol{s}\in S} \big[\ln L(\boldsymbol{s}^H|\boldsymbol{s}) + \ln L(\boldsymbol{y}|\boldsymbol{s})\big]
\end{aligned} \tag{2.19}$$

where $\ln L(\boldsymbol{s}^H|\boldsymbol{s})$ is the log-likelihood function of the sampling survey data (i.e. logarithm of the probability of observing sampling vector $\boldsymbol{s}^H$ if $\boldsymbol{s}$ is the actual OD flow vector) and $\ln L(\boldsymbol{y}|\boldsymbol{s})$ is the log-likelihood function of the traffic counts (i.e. logarithm of the probability of observing link flow vector $\boldsymbol{y}$ if $\boldsymbol{s}$ is the actual OD flow vector) and S is the set of feasible OD flow vectors, which usually coincides with the non-negative orthant. Depending on the assumption made regarding the underlying distribution, one reaches specific likelihood functions to optimise. The most widely used probability laws are Poisson and multivariate normal: the former is non-negative and therefore models traffic more accurately, but the latter is computationally more efficient.
Another possible estimator is the *Generalised Least Squares*, which starts from a system of linear stochastic equations, notably:

$$\begin{aligned}
\boldsymbol{y} &= A\boldsymbol{s} + \boldsymbol{\varepsilon} \qquad & E[\boldsymbol{\varepsilon}] = \mathbf{0}; Var[\boldsymbol{\varepsilon}] = W \\
\boldsymbol{s}^H &= \boldsymbol{s} + \boldsymbol{\eta} \qquad & E[\boldsymbol{\eta}] = \mathbf{0}; Var[\boldsymbol{\eta}] = V
\end{aligned} \tag{2.20}$$

where $\boldsymbol{\varepsilon}$ is the vector of deviations between the observed link flows $\boldsymbol{y}$ and the flows obtained by assigning the true unknown demand vector $\boldsymbol{s}$ to the links in the network through mapping A, $\boldsymbol{\eta}$ is the vector of deviations between the elements of the sampled demand vector $\boldsymbol{s}^H$ and its true unknown counterpart $\boldsymbol{x}$, and V, W are covariance matrices.

Remark 5. *$\boldsymbol{\varepsilon}$ is actually the algebraic sum of the assignment error vector $\boldsymbol{\varepsilon}^{SIM}$ and the measurement error vector $\boldsymbol{\varepsilon}^{OBS}$. Let $\boldsymbol{y}^{true}$ be the true link flow vector and A^{true} be the true assignment matrix, then:*

$$\begin{aligned}
\boldsymbol{y}^{true} &= A^{true}\boldsymbol{s} = A\boldsymbol{s} + \boldsymbol{\varepsilon}^{SIM} \\
\boldsymbol{y} &= \boldsymbol{y}^{true} + \boldsymbol{\varepsilon}^{OBS}
\end{aligned}$$

The two equations above lead to:

$$\boldsymbol{y} = A\boldsymbol{s} + \boldsymbol{\varepsilon}^{SIM} + \boldsymbol{\varepsilon}^{OBS} = A\boldsymbol{s} + \boldsymbol{\varepsilon}$$

Assuming $\boldsymbol{\eta}$ has zero mean (i.e. the estimator used to obtain $\boldsymbol{s}^H$ is unbiased), then the GLS estimator can be expressed as:

$$\boldsymbol{s}^{GLS} = \underset{\boldsymbol{s}\in S}{\arg\min} \Big[(\boldsymbol{x}^H - \boldsymbol{s})^T V^{-1} (\boldsymbol{s}^H - \boldsymbol{s}) + (\boldsymbol{y} - A\boldsymbol{s})^T W^{-1} (\boldsymbol{y} - A\boldsymbol{s}) \Big] \tag{2.21}$$

Note that this objective corresponds to equation (2.18) with z_1 and z_2 being *Mahalanobis distance functions*. By employing this particular metric, $\boldsymbol{s}^{GLS}$ minimises the sum of squared euclidean distances between OD flow estimates and historic OD flow samples, and between observed link flows and assigned flows, with each distance having weights inversely proportional to the variance of its respective error. In other words, measurements and sampled OD flows for which estimates are unsure (i.e. have high variance $Var[\varepsilon_{ij}]$ or $Var[\eta_{OD}]$) will have a smaller impact on the estimation of $\boldsymbol{s}^{GLS}$. It can also be noted that the GLS and ML estimator are equivalent if probability laws for sampled OD flows and traffic counts are assumed to be multivariate normal distributions.

The last estimator uses *Bayesian inference* methods, where experimental information is combined with non-experimental information. A Bayesian estimator is obtained by maximising the a posteriori probability of the OD flow vector $\boldsymbol{s}$ given a priori non-experimental information $\boldsymbol{s}^{NE}$ and experimental information $\boldsymbol{y}$. According to Bayes theorem, this quantity can be expressed as the product of the a priori probability $g(\boldsymbol{s}|\boldsymbol{s}^{NE})$ and the likelihood of observing the traffic counts $\boldsymbol{y}$ given the OD flow vector $\boldsymbol{s}$, $L(\boldsymbol{y}|\boldsymbol{s})$. By taking the natural logarithm of these two terms, the Bayesian estimator can be expressed as:

$$\boldsymbol{s}^B = \underset{\boldsymbol{s}\in S}{\arg\max} \Big[\ln g(\boldsymbol{s}|\boldsymbol{s}^{NE}) + \ln L(\boldsymbol{y}|\boldsymbol{s}) \Big] \tag{2.22}$$

Once again, if both distributions are multivariate normal, this is equivalent to the GLS objective. One major difference, however, is that $\boldsymbol{s}^{NE}$ is not necessarily obtained experimentally and may relate to the predictions of an expert that is applying his domain knowledge, with the corresponding covariance matrix V^B translating the analyst's confidence in his estimates. For an in-depth analysis and comparison of the three statistical methods presented here, one can refer to [27].

In practical situations, historic information about OD flows may not be available, or may simply be irrelevant. As was previously mentioned, $\boldsymbol{s}^H$ can be omitted from the formulation of the various statistical estimators. Researchers have recently looked into this particular case and investigated ways of mitigating the ill-posedness of the problem through assumptions. To this purpose, [6] proposed a generic sparsity regulariser, and formulated a *sparse GLS objective*:

$$\boldsymbol{s}^{SGLS} = \underset{\boldsymbol{s}\geq 0}{\arg\min} \Big[(\boldsymbol{y} - A\boldsymbol{s})^T W^{-1} (\boldsymbol{y} - A\boldsymbol{s}) + \lambda ||\boldsymbol{s}||_1 \Big] \tag{2.23}$$

where $||\cdot||_1$ denotes the $l1$-norm of a vector and λ is an arbitrary parameter. The underlying assumption is that the OD vector should be sparse, that is, the demand for most origin-destination pairs should equal zero and most of the traffic observed is the result of travel between a small subset of $\mathcal{N} \times \mathcal{N}$. The second term introduces sparsity through a convex

relaxation of the $l0$-norm $||\cdot||_0$, which counts the number of non-zero elements of a vector. This approach generated promising results for OD vectors that are effectively sparse. However, the "spatial" sparsity assumption may not hold in all situations.
Up to this point, the assignment matrix A was implicitly assumed to be known a priori. It is generally computed via an *all-or-nothing* assignment (i.e. each OD flow chooses exclusively the path that has the least cost) or through more complex algorithms (e.g. *equilibrium based* assignments [10]). We will now present two methods that jointly estimate the assignment matrix and demand flows.

In [7], temporal patterns have been investigated to reduce the dimensionality of the assignment elements (O flow path choice proportions in this case), thus mitigating the further complexity introduced. The authors assumed constant path choice proportions for O flows over sets of sampling time intervals, and the exact model was expressed as:

$$y_l^t = \sum_{k=t-\tau_{\max}}^{t} \sum_{o} \sum_{\boldsymbol{p}\in\mathcal{P}_o} \delta_{\boldsymbol{p},l}^{k,t} \cdot x_o^k \cdot r_{\boldsymbol{p}|o}^k \tag{2.24}$$

where $r_{\boldsymbol{p}|o}^k$ denotes the portion of x_o that flows through $\boldsymbol{p}$ to reach its destination, and $\mathcal{P}_o$ denotes the set of paths that start from node o. Matrix R^k is effectively a mapping from O flows $\boldsymbol{x}_o$ to path flows $\boldsymbol{s}^{\text{path}}$. It is assumed that the sampling time interval is greater than $\tau_{\max}$ so that $\delta_{\boldsymbol{p},l}^{k,t} = 1$ if path $\boldsymbol{p}$ contains link l and $k = t$, 0 otherwise.
The dimensionality reduction is carried out by partitioning the set of all time intervals into level sets $M_t = \{t' : \mu(t') = \mu(t)\}$ of the map $\mu(t)$ such that $r_{\boldsymbol{p}|o}^t = r_{\boldsymbol{p}|o}^{\mu(t)}$: this means that path choice proportions of origin flows are assumed to be constant for the level sets M_t. A possible partition could be over the same time-of-day on days of the same category: if school periods and school holidays are distinguished and if the sampling time interval is 1 hour, this would lead to $24 \times 14 = 336$ level sets. A least squares objective is defined to estimate the best possible O flows and path choice proportions, effectively making this a path flow based method.
This approach effectively reduces the complexity of the path flow model for estimating the OD matrix, under the assumption of *semi-dynamic flows*, but full path enumeration is still required, which considerably restricts practical use cases.

The method above assigns O flows to observed link flows via a path flow based mapping and the OD matrix is still estimated through a path flow model. In [8], the inverse problem is built directly on O flows, so that the number of inquired flows increases linearly with the number of nodes in the network. Moreover, temporal patterns are also considered to arrive at a unique solution: notably, sparsity in a transform domain is assumed. Because the project aims at extending this work, let us study it in more detail.
After proving that OD flows can be uniquely reconstructed from O flows without any loss of information, the authors propose a joint estimation of O flows and the assignment matrix P, which is formulated as follows:

$$\min_{P,\{\boldsymbol{x}^t\}} \sum_{t=1}^{n_T} ||\boldsymbol{y}^t - P\boldsymbol{x}^t||_2^2 \tag{2.25}$$

for the single-step model, and:

$$\min_{\{P^t\},\{\boldsymbol{x}^t\}} \sum_{t=1}^{n_T} \left|\left| \boldsymbol{y}^t - \sum_{\tau=1}^{\tau_{\max}} P^\tau \boldsymbol{x}^{t-\tau+1} \right|\right|_2^2 \tag{2.26}$$

for the multi-step model.
The $l2$-norm was used as the distance function in both cases. If prior information is available, the objective function can easily be extended to accommodate it, leading to:

$$\begin{aligned} \min_{\{P^t\},\{\boldsymbol{x}^t\}} & \sum_{t=1}^{n_T} \left\| y^t - \sum_{\tau=1}^{\tau_{\max}} P^\tau x_o^{t-\tau+1} \right\|_2^2 \\ & + \sum_{t=1}^{\tau_{\max}} \operatorname{vect}\left(P^t - P^{H,t}\right)^T W_P^{-1} \operatorname{vect}\left(P^t - P^{H,t}\right) + \sum_{t=2-\tau_{\max}}^{n_T} \left(\boldsymbol{x}^t - \boldsymbol{x}^{H,t}\right)^T W_X^{-1} \left(\boldsymbol{x}^t - \boldsymbol{x}^{H,t}\right) \end{aligned} \tag{2.27}$$

Assignment is assumed to be static and flows are assumed to be dynamic (although destination and path choice proportions are implicitly fixed for each destination, by static assignment). Five constraints are imposed on the network's topology and vehicles' behaviour:

C1: Non-negativity of flows: $\boldsymbol{x}^t \geq \boldsymbol{0}, \forall t$

C2: Probability constraint on assignment matrix: $0 \leq p_{ij,o}^t \leq 1, \forall t$

C3: Observability constraint: all traffic originated from a node is observed: $\sum_{i \neq o} p_{io,o} = 1,\ \forall o$ for the single-step model and $\sum_{\tau=1}^{\tau_{\max}} \sum_{i \neq o} p_{io,o}^\tau = 1,\ \forall o$ for the multi-step one

C4: Speed constraint. A "soft" version of this constraint is that all vehicles travel at the same constant speed: this allows us to have the assignment matrix P^t corresponding to the t-th step of the flows and to set $p_{ij,o}^t = 0$ if link ij is not involved in the t-th step of x_o. A "harder" version is to assume the traffic model to be *rigid* (as opposed to *elastic*), where all vehicles move through exactly one link in one sampling time interval. This further condition leads to two new sets of equations: $p_{oi,o}^t = 0$ for $t > 1$, $p_{ij,o}^t = 0$ for $t = 1$, for all o, for all $i \neq o$.

C5 : Flow constraint: for all nodes $i \neq o$, the inflow from o must be greater than or equal to the outflow from o: $\sum_j p_{ji,o} - \sum_k p_{ik,o} \geq 0,\ \forall o$ and $\forall i \neq o$ for the single-step model and $\sum_j p_{ji,o}^{t-1} - \sum_k p_{ik,o}^t \geq 0,\ \forall t,\ \forall o$ and $\forall i \neq o$ (under a rigid traffic model assumption)

Note that constraint C4, and constraint C3 to a lesser extent, limit the applicability of the framework to real road networks. On the other hand, constraints C1, C2 and C5 are concerned with the interpretability of flows and assignment: they hold in any practical context and actually make sure that the results obtained are physically sound.
The authors then proceed to proving that the sparsity assumption promotes uniqueness of the solution to the inverse problem, and assume that O flows are sparse in a domain transform

(i.e. that temporal patterns exist in demand flows), which makes the framework significantly different from that in [6] and more applicable when analysing busy communities, for example. Sparsity in the transform domain is enforced as follows: define an orthonormal transformation D such that $\boldsymbol{c}_o = D\boldsymbol{x}$ and $\|\boldsymbol{c}_o\|_0 \ll n_T$. (2.26) is reformulated as:

$$\min_{\{P^t\},\{\boldsymbol{x}_o^t\}} \sum_{t=1}^{n_T} \frac{1}{2} \left\| \boldsymbol{y}^t - \sum_{\tau=1}^{\tau_{\max}} P^\tau \boldsymbol{x}^{t-\tau+1} \right\|_2^2 + \sum_o \lambda_o \|D\boldsymbol{x}\|_1 \tag{2.28}$$

where λ_o is a regularisation parameter and the $l1$-norm $\|\cdot\|_1$ is chosen as a convex relaxation of the $l0$-norm $\|\cdot\|_0$.
Finally, noise is considered to arrive at a constrained optimisation problem. Observation error is assumed to be known a priori, with the relative level of noise being upper bounded by $\varepsilon > 0$:

$$\frac{\sum_t \|\boldsymbol{y}^t - P^t * \boldsymbol{x}^t\|_2^2}{\sum_t \|\boldsymbol{y}^t\|_2^2} \leq \varepsilon$$

leading to the following formulation:

$$\min_{\{P^t,\boldsymbol{x}^t\}} \sum_o \|D\boldsymbol{x}_o\|_1 \tag{2.29}$$

$$\text{subject to } \sum_{t=1}^{n_T} \left\|\boldsymbol{y}^t - P^t * \boldsymbol{x}^t\right\|_2^2 \leq \varepsilon \sum_{t=1}^{n_T} \left\|\boldsymbol{y}^t\right\|_2^2 \tag{2.30}$$

The Gauss-Seidel method is proposed in order to iteratively solve the problem. Experimental results show that transform domain sparsity significantly improves estimation accuracy, but also that a faster algorithm might considerably improve the performance of the framework, especially for unidirectional network.

So far, we have covered the foundations underlying the work carried out in this project, and have presented theoretical and practical aspects of OD estimations, as well as the most common and novel methods employed in literature. The following section describes the work undergone to improve and extend the O flow based framework.

3. Analysis and design

3.1 Original framework's description and issues

As previously mentioned, the O flow based framework developed in [8] achieved a dimension reduction in inquired flows for OD estimation. It relies on several theoretical assumptions, which we will also use throughout our discussion, unless specified otherwise. These assumptions are:

- The network is uncongested, and all vehicles travel at constant speed throughout their journeys
- The links have unit length and are traversed in exactly one observation time sample
- OD flows do not take paths that contain loops, neither do they take paths longer than a known maximum path length $\tau_{\max}$

The work carried out in this project starts with the observation that the O flow based framework for blind estimation of OD flows provides results that are less satisfactory in the single-step case than in its multi-step counterpart (see figures 5.1a and 5.2a for simulation results proving this statement). This is far from being unexpected, since the multi-step assignment matrix P^t at each step t will be sparser than the single step matrix $P = \sum_{\tau=1}^{\tau_{\max}} P^\tau$, and will therefore yield more accurate solutions for the O flow vector $\boldsymbol{x}_o$. However, although less precise, the single-step model is used in real-life studies [28] and improving the solver for this version of the model is still relevant. Apart from the reduced sparsity, a possible cause of the loss in performance is found in loop-flow assignment by the solver, despite the fact that a loop-free path generation of flows is assumed: equivalently, there are currently no constraints in the optimisation problem, as it is formulated in [8], to prevent loop flows from being present in the solution. We use the term *solver* as a more concise way of referring to the part of the framework that is concerned with the optimisation of the objective function and the formulation of the constraints, while the term *generator* refers to the assignment of flows in the network and the assumptions that define it.

The aim of removing loop flows in the solver can be formally expressed as follows: we aim to find a condition $\mathcal{C}_S$ on the problem's variables such that all and only path flows that are actually generated can be part of the final solution. The generated flows depend on the conditions, or assumptions, that are enforced on the assignment: they will be denoted by $\mathcal{C}_G$ in the discussion that follows. The condition $\mathcal{C}_S$ will later be translated as a constraint, or a set of constraints, in the optimisation problem formulation. When applied to the scenario that is being studied, this gives:

$$\mathcal{C}_S\left(\{P^t, \boldsymbol{x}^t\}\right) \Leftrightarrow \mathcal{C}_{G,LF} \tag{3.1}$$

where $\mathcal{C}_{G,LF}$ stands for the loop-free path flow assignment assumption on the generator. In less technical terms, equation (3.1) may be expressed as: *"the assignment can be entirely explained by path flows that do not contain any loops"*. The equivalence consists of two implications, the

first one ($\Leftarrow$) stating that all assignments that can be explained by loop-free path flows are considered by the solver and the second one ($\Rightarrow$) stating that only assignments that can be explained by loop-free path flows are considered by the solver. Note that the first implication is already satisfied by the constraints in the original framework, but not the second one. With this goal set, we now proceed to give a more precise definition of loop flows, identify the different types of loops that may be formed by flows in road networks, and show examples of loop situations that might deteriorate the performance of the solver.

3.2 Definition and characterisation of loop flows

Loop flows should be more correctly referred to as *path flows containing loops*, but we will use the former terminology for convenience. Let $s_{\boldsymbol{p}}^{\text{path}}$ be the path flow through path $\boldsymbol{p} = [n_1, n_2, \ldots, n_l]$ where n_i $(i = 1, \ldots, l)$ are nodes and l is the length of the path. $s_{\boldsymbol{p}}^{\text{path}}$ is defined as a loop flow if there exist two elements of $\boldsymbol{p}$, n_i and n_j such that $n_i = n_j$ and $i \neq j$, that is, if the flow passes at least twice through the same node when travelling from its origin to its destination.
In order to devise a suitable condition $\mathcal{C}_S$, let us have a more detailed look at loop flows, and try to divide them into distinct cases based on the link flows that take part in the loop formation. This will help in carrying out a more constructed analysis.
Two classes of loops were identified:

- Whenever a flow leaves a node i using link (i, j) and comes back to it (thus forming a loop) using link (j, i), we will refer to it as a *"U-turn loop"*, since it is the most likely physical analogy where this might happen (see sub-figure 3.1a for a graphical representation of an example)

- When a flow leaves a node i using link (i, j) (or a set of links) and comes back to it using link (k, i), $k \neq j$, (or a set of links from the one that was used to leave the node) we will refer to it as a *"circular loop"* as the vehicles flow through a series of links in a circular fashion before reaching a point they were previously in (see sub-figure 3.1b for a graphical representation of an example)

Of course, loop flows can contain several instances of both types of loops, and different loops may pass though the same node.
In order to understand why such flows are problematic in the initial framework, consider the networks depicted in figure 3.1, where link flows have been included next to their corresponding edge for clarity.

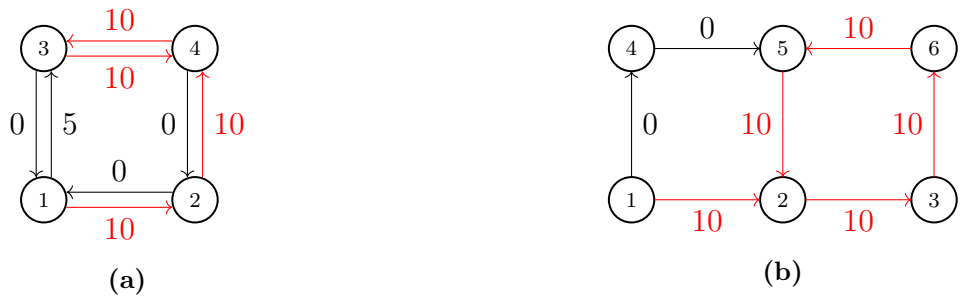


Figure 3.1: Example of situations where loop flow assignment is possible

Remark 6. *When creating examples, it is essential to consider situations where an assignment containing loop flows and another containing only loop-free flows are both possible, since we know that the flows generated do not contain loops. Thus, considering link flows that must be necessarily explained by loop flows is pointless.*

In sub-figure 3.1a, assume that nodes 1 and 3 originate traffic while nodes 2, 3 and 4 accept incoming traffic. Regarding the single-step model, this set of link flows could be generated since it is possible to assign OD flows such that there are no loop flows, for example $x_{13} = 10$, $x_{14} = 5$ and $x_{34} = 5$. However, it is also possible for the solver to explain all the O flows by origin 1, that is, to estimate $\hat{x}_1 = 15$ and $\hat{x}_3 = 0$, in which case the path flow starting at node 1 and ending in node 4 is necessarily a loop flow, as it contains a U-turn loop.
For the multi-step model, consider the circular loop in sub-figure 3.1b: the solver is susceptible to assign O flow proportions such that a loop flow interpretation is necessary. Assume again that nodes 1 and 5 are origins while node 2 is the only viable origin. If each link count belongs to a successive time step, starting with link $(1, 2)$ and ending with link $(5, 2)$, then explaining all link flows by origin flow x_1 leads to a necessary loop interpretation.
For both single-step and multi-step, the five constraints defined in section 2.4.3 do not prevent the loop flow assignments from happening in the final solution $\left(\hat{P}, \{\hat{\boldsymbol{x}}^t\}\right)$ or $\left(\{\hat{P}^t\}, \{\hat{\boldsymbol{x}}^t\}\right)$.

3.3 Extending the flow constraint to prevent U-turn loops

To prevent the solver from estimating assignments that cannot be explained by loop-free path flows and thus are known not to be possible solutions, a way to identify loops based on $\boldsymbol{x}^t$ and P^t (or P) needs to be devised.
By analysing the example in sub-figure 3.1a more carefully, it is possible to notice that the current formulation of the loop constraint does not reduce the feasible region as much as it could, given the information we possess: indeed, there is no reason for adding $p_{43,1}$ to the summation that calculates the inflow in node 3 due to origin flow 1, since all of $p_{43,1}$ must stop in 3, or else it would necessarily form a loop. Additionally, the new constraint must be expressed on every link, not just on every node. By taking these elements into consideration, we can now check whether it is possible to explain an estimated assignment by paths that do not contain U-turn loops. To conveniently formulate this condition, let $g_k^{i,j}$ denote the set of

links that form the k^{th} path between OD pair (i, j), and let $g^{i,j}$ denote the union of all such sets:

$$g^{i,j} = \bigcup_k g_k^{i,j}$$

Given that flows are allowed to take any loop-free path to reach their destination, the assignment estimated by the solver can be explained by paths that do not contain U-turn loops when the following condition is enforced:

$$\sum_{\substack{j \neq k \\ (j,i) \in \bigcup\limits_{l \neq i} g^{o,l}}} p_{ji,o} - p_{ik,o} \geq 0, \ \forall o, \ \forall (i,k) \in \mathcal{L}, \ i \neq o \tag{3.2}$$

The condition given by equation (3.2) expresses the fact that for each O flow x_o, the fraction of it that flows through any link (i, k) can be explained by inflows in node i that do not come from node k and that are allowed to continue their path through link (i, k) (that is, that do not have to stop at node i).

However, this condition does not solve the problem for two main reasons: first, it does not prevent necessary explanation of the assignment by flows containing circular loops. This can be easily proved with a counter-example, as in sub-figure 3.2b: all conditions, including the new one expressed in (3.2), are satisfied if all link flows are explained by x_3, in which case a circular loop path is necessary. Second, it is not possible to impose a maximum path length on the generated paths: in sub-figure 3.2a, setting $\tau_{\max} = 4$ makes the red path from 1 to 6 illegal. However, an assignment fully explained by origin node 1 satisfies all constraints. This is actually a further problem of the single-step model that exists in the original framework.

The ability of setting $\tau_{\max}$ is a desirable property in the simulations. Indeed, if one could not set a maximum path length, this would pose a problem in terms of computation and performance, since the assignment matrices would be less sparse, but it would also cause a loss in behavioural meaning of the generated flows. Consider a 3 by 3 bidirectional network: if a loop-free path assignment is chosen to reflect the fact that loop paths would not be chosen by users to reach their destination, then it does not make sense to let the OD flow s_{19} take an "S-shaped" path that goes through all the nodes: such a detour is as (if not more) unlikely to be chosen as a path that contains a loop.

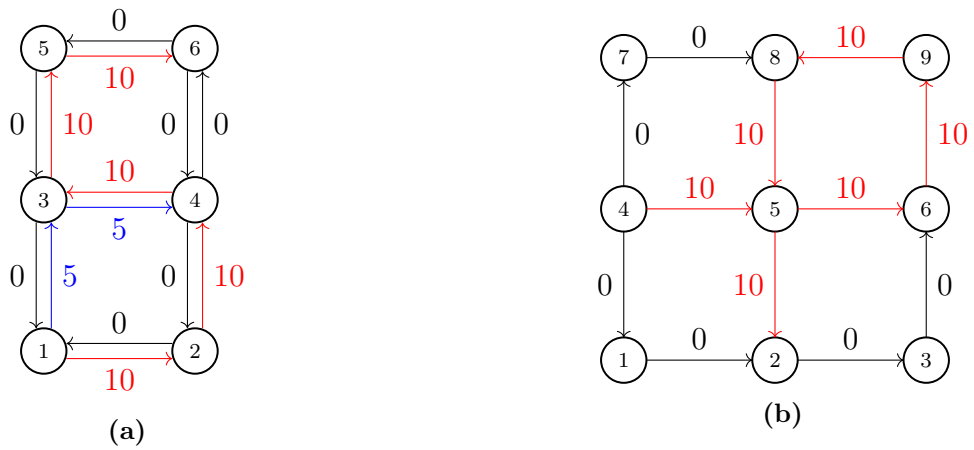


Figure 3.2: Example of undesirable situations where assignment satisfies old and newly introduced constraints

The findings above can be intuitively interpreted by noticing that the loop-free path assignment assumption with maximum path length is a condition on path flows. As was shown in 2.2, by reducing the complexity of inquired flows, information is lost both on path flow assignment and on OD flow assignment, which is why it is hard to formulate the condition $\mathcal{C}_S$ that selects all and only assignments that can be explained by loop-free paths shorter than $\tau_{\max}$. An algorithm to identify the presence of necessary loop flows based on assignment, and in general that checks the validity of an assignment given a set of legal paths, should be fairly straightforward to design, possibly using backtracking. However, it is outside the scope of this project and was thus not investigated further. In any case, it would be tedious if not impossible to translate a backtracking algorithm into constraints for a convex (or bi-convex, more precisely) optimisation problem.

3.4 Introducing destination flows as a possible improvement

Another research direction that was taken consisted in using information derived from destination flows (D flows) in order to prevent loops in the final solution. Before explaining how this might help, let us first introduce this new type of flows: let $\boldsymbol{z}$ be the D flow vector and let Q be the assignment matrix that maps D flows to link flows. One has:

$$\boldsymbol{y} = Q\boldsymbol{z} \tag{3.3}$$

for the single-step model, and:

$$\boldsymbol{y}^t = \sum_{\tau=-\tau_{\max}}^{-1} Q^\tau \boldsymbol{z}^{t-\tau-1} = Q^t * \boldsymbol{z}^t \tag{3.4}$$

for the multi-step model. Notice that in this last case, the assignment matrix Q^τ is non-negative for negative values of τ, that is, for previous steps of the D flows concerned. Intuitively, if one

wants to assign destination flows to current link flows at time t, the destination flows involved have not yet reached, or are just reaching their destination. Also note the difference with the O flow based multi-step model, in particular the fact that a 1 is subtracted from $\boldsymbol{z}$'s time index in the summation: this is due to the fact that indexing for τ does not start at zero. This slight adjustment is required for consistency of assignment maps P^τ and Q^τ.

Going back to the example in figure 3.4, one may notice that loops in the two links connecting nodes 3 and 4 may be prevented by setting $q_{34,3} = 0$ and $q_{43,4} = 0$, or more generally:

$$q^t_{di,d} = 0 \quad \forall i, \quad \forall d, \quad -\tau_{\max} \leq t \leq -1 \tag{3.5}$$

Equation (3.5) states that no vehicles can leave their destination after having reached it, a property of the estimated assignment that cannot be expressed using matrix P alone, as was discussed in the example. In the O flow model, a respective set of equations applied to P^t is included in the speed constraint and is expressed as:

$$p^t_{io,o} = 0 \quad \forall i, \quad \forall o, \quad 1 \leq t \leq \tau_{\max} \tag{3.6}$$

stating that flows do not pass by their origin node after having left it.

Now that the usefulness of D flows was shown, the following optimisation problem results from merging them in the O flow based model:

$$\min_{P,\{\boldsymbol{x}^t\},Q,\{\boldsymbol{z}^t\}} f_{\text{cost}}\left(P, \{\boldsymbol{x}^t\}\right) + f_{\text{cost}}\left(Q, \{\boldsymbol{z}^t\}\right) \tag{3.7}$$

for the single-step model, and a similar formulation considering P^t and Q^t for the multi-step model. This will be subject to 11 constraints: the 5 constraints defined near the end of section 2.4.3 for the O flow model, 5 more constraints, similar to the previous ones and relating to the D flow part of the new problem, and a final constraint that ensures the equality between the two mappings from O flows and from D flows to link flows, that is:

$$P\boldsymbol{x}^t = Q\boldsymbol{z}^t, \qquad 1 \leq t \leq n_T \tag{3.8}$$

for the single-step model, and:

$$\sum_{\tau=1}^{\tau_{\max}} P^\tau \boldsymbol{x}^{t-\tau+1} = \sum_{\tau=-\tau_{\max}}^{-1} Q^\tau \boldsymbol{z}^{t-\tau-1}, \qquad 1 \leq t \leq n_T \tag{3.9}$$

for the multi-step one.
This formulation brings more information than any of its constituent parts individually and does not alter the complexity of the inquired flows, since $\mathcal{O}(n_O + n_D) \in \mathcal{O}(n_n)$.
However, it was decided not to implement this framework and instead focus on other strategies for several reasons.
First, as we have seen in the model comparison in section 2.2, each model imposes different

"degrees of freedom" for dynamic OD flows. In this case, if P^t and Q^t are both assumed to be static, then a further constraint is imposed on destination flows, such that the origin proportions for each destination node must remain constant throughout the sampling time horizon. This consequence is particularly problematic because it reduces the number of additional independent equations that are obtained from link flows as the sampling time horizon n_T increases, which is undesirable in a setting where the problem we are trying to solve is already severely ill-posed. As an example, consider the following 5-node unidirectional network representing a crossing (note that this is only an example of a situation where D flows make the problem more ill-posed, and it is not a formal proof that this is the case in general).

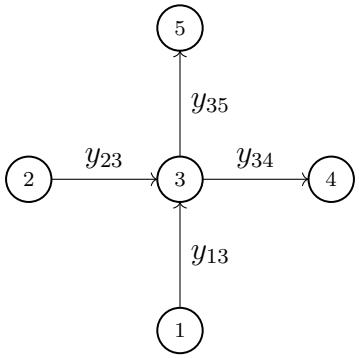


Figure 3.3: Situation where D flows make the problem more ill-posed

The edges were labelled by their corresponding link flows. Now let's assume that nodes 1 and 2 are origins in the network while nodes 4 and 5 are destinations. In this simple setting, origin and destination flows as well as some elements of their respective assignment matrices are straightforward to obtain from link flow counts and from the network topology:

$$\boldsymbol{x}^t = \begin{bmatrix} x_1^t \\ x_2^t \end{bmatrix} = \begin{bmatrix} y_{13}^t \\ y_{23}^t \end{bmatrix}; \boldsymbol{z}^t = \begin{bmatrix} z_4^t \\ z_5^t \end{bmatrix} = \begin{bmatrix} y_{34}^t \\ y_{35}^t \end{bmatrix}; p_{13,1} = p_{23,2} = q_{34,4} = q_{35,5} = 1$$

In the general case, it is safe to assume $p_{34,1} > 0, p_{34,2} > 0, p_{35,1} > 0, p_{35,2} > 0$, which means that there will be flows from each origin directed to each of the destinations. In an O flow based framework, a unique solution (in both single and multi-step cases) will normally be found for an observation time horizon $n_T \geq 2$, since this will result in a system of 8 equations in 8 unknowns. The only condition for this is that the O flows are in different proportions in at least two time steps. But when D flows are introduced, one obtains the following set of equations for the first time step:

$$\begin{cases} x_1^1 = & q_{13,4} z_4^1 + q_{13,5} z_5^1 \\ x_2^1 = & q_{23,4} z_4^1 + q_{23,5} z_5^1 \\ z_4^1 = & p_{34,1} x_1^1 + p_{34,2} x_2^1 \\ z_5^1 = & p_{35,1} x_1^1 + p_{35_2} x_2^1 \end{cases}$$

If at the next time step $x_1^2 = a x_1^1,\ a \geq 0$, this implies:

$$
\begin{aligned}
& x_1^2 = q_{13,5} z_5^2 + q_{13,4} z_4^2 \\
\implies & x_1^2 = (q_{13,4} p_{34,1} + q_{13,5} p_{35,1}) x_1^2 + (q_{23,4} p_{34,2} + q_{23,5} p_{35,2}) x_2^2 \\
\implies & a(1 - q_{13,4} p_{34,1} - q_{13,5} p_{35,1}) x_1^1 = (q_{23,4} p_{34,2} + q_{23,5} p_{35,2}) x_2^2
\end{aligned}
$$

But $(1 - q_{13,4} p_{34,1} - q_{13,5} p_{35,1}) x_1^1 = (q_{23,4} p_{34,2} + q_{23,5} p_{35,2}) x_2^1$, which leads to:

$$
x_1^2 = a x_1^1 \implies \begin{cases} x_2^2 = a x_2^1 \\ z_4^2 = a z_4^1 \\ z_5^2 = a z_5^1 \end{cases}
$$

This is equally valid if any single one of the other OD flows involved is scaled by a, and the example holds in the multi-step case as well. In a hybrid O and D flow based framework, $a = 0$ at any observation time sample would imply null flows over the whole network, bringing no information at all. On the other hand, $x_1^t = 0$ and $x_2^t \neq 0$ in an O flow based framework would result in the derivation of a unique solution.
Furthermore, the introduction of D flows only prevents loop flows for which the repeated node in the path is the path's destination, which is only a small subset of all possible loop flows. Consider the following 4-node network:

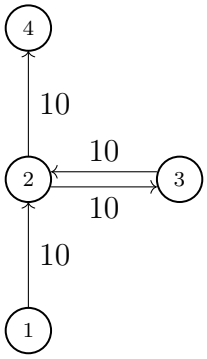


Figure 3.4: Loop flow that newly introduced D flow constraints cannot prevent

Assume the solver leads to an assignment where $x_1 = 10$ and $z_4 = 10$ are the only origin and destination flows, respectively. Then, the assignment is necessarily explained by a U-turn loop even though it satisifies the constraint defined in (3.6).
As such, only a subset of loop flows are prevented by the introduction of D flows, while at the same time imposing additional fixed proportions in the flows which may make the problem even more ill-posed in certain situations: hence, it was decided to abandon this strategy.

3.5 A sufficient condition for the absence of loop flows

At this point, it was decided to focus our research on finding a sufficient condition on the estimated assignment matrix for the absence of loop flows, keeping in mind that some legal

assignments might not satisfy the set of constraints any more.
It is useful to study network topologies where loop flows cannot happen, and all paths are necessarily loop-free paths: one such example is found in figure 2.1, where it can be noticed that all flows, when travelling from one node to another, necessarily end up strictly further away from their origin as compared to the node they were previously in. This is not necessarily the case in all network topologies, but the same behaviour can easily be enforced in general networks (by the condition that when traversing a link, vehicles must always end up strictly further away from the origin than where they were before).
Let us introduce two further elements:

- The *link cost vector* $\boldsymbol{c}$, where c_{ij} denotes the cost to take link (i, j) and $\boldsymbol{c} = [\ldots, c_{ij}, \ldots]$
- The *shortest path matrix* F, where each element $f_{j,i}$ denotes the cost of the shortest path starting from node i and ending in node j, that is: $f_{j,i} = \min_k \left(\sum_{l \in g_k^{i,j}} c_l \right)$. If a node i cannot be reached from origin o as $g^{i,j} = \bigcup_k g_k^{i,j} = \varnothing$, then $f_{i,o}$ is undefined. This might pose problems during implementation. As such, we propose the following remedy: $f_{i,o} = \infty$ if $g^{i,j} = \varnothing$, and $P^{\infty} = \mathbf{0}$, where $\mathbf{0}$ denotes a matrix of zeros.

Note that because the original framework is still being considered at this point and that the traffic model is assumed to be rigid, we have $\boldsymbol{c} = \mathbf{1}^{n_l}$ (a vector of n_l ones) and $F \in \mathbb{N}^{n_n \times n_n}$, with $f_{j,i} = \min_k |g_k^{i,j}|$, where here $|\cdot|$ denotes the cardinality of a set.

The condition and its implication on the generator can now be expressed more formally as follows:

Theorem 1. *Provided that* $\boldsymbol{c} > 0$ *element-wise:*

$$p_{ij,o}^t = 0, \forall i, j, o \text{ such that } f_{i,o} + c_{ij} > f_{j,o} \Rightarrow \mathcal{C}_{G,LF} \tag{3.10}$$

Proof. The proof is carried out by contradiction: consider any assignment P^t where the following condition holds: $p_{ij,o}^t = 0, \forall i, j, o$ *such that* $f_{i,o} + c_{ij} > f_{j,o}$. Now assume that the same assignment must necessarily be explained by path flows containing a loop, that is, assume that loop flows were generated. Let n_2 be a node that appears twice in any of the paths containing loops, and let (n_1, n_2) be the link that closes the loop in question at time t. Also let o be the node that generated the loop flow in question. There are two possibilities here:

- $f_{n_1,o} + c_{n_1 n_2} > f_{n_2,o}$: in a rigid setting, this means the flow moves closer to its origin o when traversing link (n_1, n_2), or ends at the same distance from o.
  But this implies $p_{n_1 n_2,o}^t = 0$, which is impossible since this link should carry the flow that closes the loop.

- $f_{n_1,o} + c_{n_1 n_2} = f_{n_2,o}$: the flow moves further from its origin o when traversing link (n_1, n_2). The flow must have reached n_1 from n_2, otherwise this wouldn't constitute the loop as we

have defined it. Denote by $\{m_k\}$ the sequence of nodes traversed by the flow to reach n_2 from n_2 passing by n_1. Since $f_{n_1,o} + c_{n_1 n_2} = f_{n_2,o}$, somewhere along this sequence of nodes the flow must have traversed a link (m_1, m_2) at time t' such that $f_{m_1,o} + c_{m_1 m_2} > f_{m_2,o}$, which in turn implies $p^{t'}_{m_1 m_2,o} = 0$, which is contradictory. ($f_{n_1,o} + c_{n_1 n_2} < f_{n_2,o}$ is not a possibility since $f_{n_2,o}$ is the minimum distance from n_2 to o by definition).

□

By enforcing this condition, the solver loses the property of considering all assignments generated by legal path flows:

$$p^t_{ij,o} = 0, \forall i, j, o \text{ such that } f_{i,o} + c_{ij} > f_{j,o} \nLeftarrow \mathcal{C}_{G,LF}$$

It is straightforward to construct an example of a legal assignment that does not satisfy the left hand side of (3.10). In the bi-directional network depicted in sub-figure 3.1a, path $1 \to 2 \to 4 \to 3$ is a perfectly legal path, not containing any loops, that does not satisfy the aforementioned condition. In the next section, an alternative assignment is proposed so that the assignments estimated by the solver will cover all and only path flows that could have been generated.

3.6 Shortest path assignment: matching solver and generator

Condition (3.10) is sufficient to check for the absence of loop flows, but it does not hold for some legal assignments, as shown in the previous section. One way to perfectly match solver and generator is by modifying the assumptions on the assignment of generated path flows: notice indeed that the left-hand side of (3.10) is a sufficient and necessary condition for shortest path generation. Denote by $\mathcal{C}_{G,SP}$ the condition on the flow assignment such that the generated OD flows must randomly select one of the shortest paths between origin and destination.

Theorem 2. *Provided that* $\boldsymbol{c} > 0$ *element-wise:*

$$p^t_{ij,o} = 0, \forall i, j, o, t \text{ such that } f_{i,o} + c_{ij} > f_{j,o} \Leftrightarrow \mathcal{C}_{G,SP} \tag{3.11}$$

The proof is self-evident, since the left-hand side of the equivalence is the definition of a shortest path assignment, but we provide it for completeness.

Proof. The proof is conducted in two steps.

($\Leftarrow$) $\mathcal{C}_{G,SP}$ holds: all O flows are known to take the shortest path to reach their destinations. Now assume that there exist $(i, j) \in \mathcal{L}, o \in \mathcal{N}, t \leq \tau_{\max}$ such that $p^t_{ij,o} > 0$ and $f_{i,o} + c_{ij} > f_{j,o}$. This means the flow has not taken the shortest path to reach node j, which is impossible.

($\Rightarrow$) $p^t_{ij,o} = 0$ holds for all i, j, o, t such that $f_{i,o} + c_{ij} > f_{j,o}$. It follows that any node j is reached by any flow exactly $f_{j,o}$ time steps after it has left its origin o, because it cannot reach it before (by definition of F), but it also cannot reach it after, as this would contradict the above condition on assignment. By definition, this implies shortest path assignment of flows.

□

Solver and generator are thus matched: all and only assignments that are generated via shortest path assignment are picked as potential solutions by the solver. Note this formulation also allows to specify a maximum path length $\tau_{\max}$ for the allowed paths. A loss of generality occurs, but the assumption is not unreasonable, especially since it is assumed that the networks we are working with are uncongested.

Regarding constraints under shortest path assignment, most remain similar to the ones defined for loop-free assignment, when considering rigid models. The modifications made involve the speed constraint, which will make P sparser. We will introduce a sixth constraint, referred to as shortest path constraint to emphasize that it was not present in the original framework, although it might be included in the speed constraint. A further simplification involves constraint 5 in order to consider less variables, although the implementation might remain strictly unchanged without affecting the results.

$\text{C5}_{\text{sp,rigid}}$:

$$\sum_j p^{f_{i,o}}_{ji,o} - \sum_k p^{f_{k,o}}_{ik,o} \geq 0, \ \forall o \text{ and } \forall i \neq o$$

for the multi-step model. In the single-step case, the flow constraint remains unchanged.

$\text{C6}_{\text{sp,rigid}}$: All flows only take the shortest paths to reach their destinations.

$$p_{ij,o} = 0, \forall i, j, o \text{ such that } f_{i,o} + c_{ij} > f_{j,o}$$

for the single-step model, and:

$$p^t_{ij,o} = 0, \forall i, j, o, t \text{ such that } f_{i,o} + c_{ij} > f_{j,o}$$

for the multi-step case, which can be reduced to:

$$p^t_{ij,o} = 0, \forall t \neq f_{j,o}$$

since flows reach nodes at specific time steps dictated by F.

Up to this point, the feasible region of the optimisation problem was reduced to prevent convergence to a solution whose assignment could not have been generated, and the set of possible paths was also restricted to perfectly match the feasible region defined by the constraints: the discussion until now has focussed on the theoretical aspects of the framework. On the practical side, one major shortcoming of the framework is the unitary link cost assumption, which makes

the model not applicable to most real-world situations, even for rough estimations or idealised scenarios.

3.7 Introduction of non-unitary, unequal link costs

3.7.1 Relation to real networks

In real-world transportation network, streets have different lengths and different speed limits applying to them. Both these characteristics can be modelled by increasing or decreasing the cost of traversing a certain link. Therefore, let us first relax the constant speed constraint from *"All vehicles travel at constant speed throughout their journey"* to *"All vehicles travelling through a given link in the network will spend the same amount of time to traverse that link"*. Notice this is a property of uncongested networks, and it allows to use link costs in order to model different characteristics of the network's topology and required user's behaviour. By making link costs a function of the time users spend in a link, we capture both the length of a link and the speed limit through that link, should it be imposed by law or by the condition of the road.

The main points discussed in the above paragraphs can be expressed mathematically as:

$$c_{ij} = h\left(\textit{length of link } (i,j),\ \textit{speed limit in link } (i,j)\right)$$
$$\boldsymbol{c} = [\ldots, c_{ij}, \ldots] \in \mathbb{R}^{n_l}$$

The purpose of function h is simply to show that the link cost is derived from length and speed limit on a link: we are not concerned with finding an analytic formulation of the relationship. What is important to understand is that $\boldsymbol{c}$ has units of time. In the single-step model the choice of the duration of one unit time for $\boldsymbol{c}$ is completely arbitrary, while in the multi-step model, we set one time unit as the duration of one sampling time interval, for convenience. For example, say that the sampling time interval of 20 minutes, and that vehicles take 5 minutes to traverse a link: the cost associated with that link will be $5/20 = 0.25$ sampling time intervals.

3.7.2 Modified mapping and constraints

Once the framework is extended to a less rigid traffic model where links have non-unitary costs, not all of the constraints defined earlier hold anymore, and the multi-step and single-step case need to be studied individually.

Remark 7. *Before working with real link costs, the simplified case where costs were positive integers was considered, and constraints were defined for it. However, because the real link costs constraints apply to this scenario as well, and to avoid repetition, it was chosen to omit them from the the current explanation, and to include them in appendix A instead.*

Multi-step model

The division in sampling time intervals makes the design of new constraints more intuitive for the multi-step case.

First, it is important to redefine the exact meaning of x_o^t in a more accurate way. Define by t_s the duration of one sampling time interval, then x_o^t denotes the flow (i.e. number of vehicles) departing from node o during time interval $[(t-1)t_s,\ t \cdot t_s)$. For example x_o^1 denotes the number of vehicles that leave node o between the start of the observation period and the end of the first sampling time interval, that is, during the first sampling time interval (notice there is an implicit scaling by t_s).

Here, the first problem is encountered, and it concerns the distribution of departures from a node during a sampling time interval. Consider a link (o, i) with $c_{oi} < 1$ and a flow x_o^1: vehicles leaving o near the end of the interval (after $1-c_{oi}$) will still be in link (o, i) during the successive interval, while those that have left in the beginning of the time interval (before $1 - c_{oi}$) will have left it by then. The example is graphically represented as a time diagram in figure 3.5: the two boxes represent the time spent in link c_{oi} for vehicles A and B. Vehicle A will have left the link by the point where the time interval ends, while vehicle B will still be in it, even though they are part of the same flow x_o^1. As such, without making assumptions on the departure of flows, a fixed matrix P^τ cannot be derived. The simplest case would be to assume that all flows x_o^t leave at the beginning of the t^{th} time interval. A more realistic but still simplified assumption would be to have vehicles leave in deterministic fashion such that they are equally spaced throughout their journey. During the subsequent discussion, we will assume the former case for simplicity: in the example above, we would know that all of x_o^1 has left link (o, i) at time interval 2. The first problem is therefore solved.

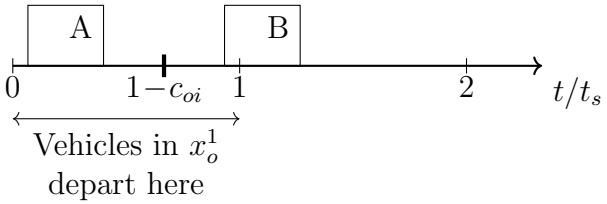


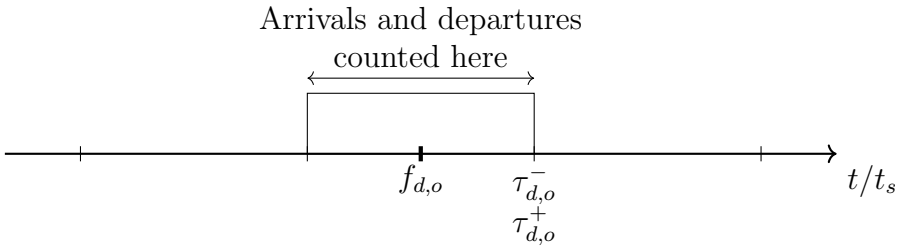
Figure 3.5: Time diagram of flow x_o^1 through link (o, i). The boxes represent the time spent by vehicles A and B in link c_{oi}.

Now that the departure distribution has been fixed, we notice that a key point when working with shortest path traffic assignment is the fact that all flows from a specific origin o leaving at a time step t_0 will reach any node d in the network $f_{d,o}$ time steps after they have departed, irrespective of the exact sequence of nodes they have visited to get there. Given this, let us continue our analysis by introducing two matrices that will provide us with information on the time steps during which flows reach nodes in the network:

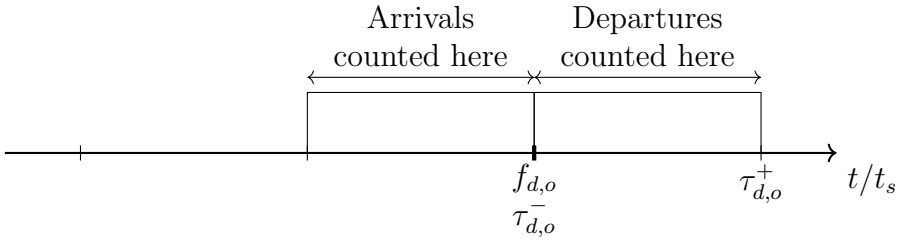
- *Node arrival matrix* T^-, where each element $\tau_{i,o}^- = \lceil f_{i,o} \rceil$ denotes the time step during which a flow originated in o reaches node i

- *Node departure matrix* T^+, where each element $\tau^+_{i,o} = \lfloor f_{i,o} + 1 \rfloor$ denotes the time step during which a flow originated in o leaves node i

One can see that $\tau^-_{i,o} = \tau^+_{i,o}$ in most cases, except when $f_{i,o}$ is an integer, that is, when a flow reaches a node exactly between two sampling time intervals. Figure 3.6 provides a graphical representation of this: in sub-figure 3.6a, flow from o arrives in d during a sampling time interval, which means that vehicles entering and exiting the node will be counted in the same step of the assignment matrix. On the other hand, in sub-figure 3.6b, flow from o reaches node d exactly between two intervals, which means that arrivals will be part of $P^{\tau^-_{d,o}}$ and departures will be part of $P^{\tau^+_{d,o}}$.



(a) $f_{d,o}$ in sampling interval



(b) $f_{d,o}$ between sampling intervals

Figure 3.6: Counting arrivals in and departures from a node

With these elements defined, the new mapping from O flows to OD flows is given by the following equation:

$$s^t_{od} = x^t_o \left(\sum_i p^{\tau^-_{d,o}}_{id} - \sum_j p^{\tau^+_{d,o}}_{dj} \right) \tag{3.12}$$

The observability, flow and shortest path constraints are modified, while a new constraint, that we will refer to as *repeated link counts* constraint, is introduced. The finalised set of constraints is shown below. Unchanged constraints from the original framework are not shown.

$\text{C3}_{\text{sp,ms}}$: We only consider the first time step, since whenever $c_{oi} > 1$, that specific link would end up contributing more than once to the summation. This gives:

$$\sum_{i \neq o} p^1_{oi,o} = 1, \ \forall o$$

$\text{C5}_{\text{sp,ms}}$: This is derived directly from the above discussion about arrivals and departures in

nodes from a given origin.

$$\sum_j p_{ji,o}^{\tau_{i,o}^-} - \sum_k p_{ik,o}^{\tau_{k,o}^+} \geq 0, \ \forall o \text{ and } \forall i \neq o$$

$\text{C6}_{\text{sp,ms}}$:

$$p_{ij,o}^t = 0, \forall i, j, o, t \text{ such that } f_{i,o} + c_{ij} > f_{j,o}$$

This can actually be reformulated in terms of time steps since link costs are expressed in units time:

$$p_{ij,o}^t = 0, \forall t \text{ such that } t > \tau_{j,o}^- \text{ or } t < \tau_{i,o}^+$$

$\text{C7}_{\text{sp,ms}}$: This new constraint translates the fact that repeated counts of the same vehicles over a link should remain constant, or equivalently that vehicles cannot stop or originate in the middle of a link.

$$p_{ij,o}^t = p_{ij,o}^{\tau_{i,o}^+} \quad \text{for } t = \tau_{i,o}^+ + 1, \ldots, \tau_{j,o}^-$$

Single-step model

The only problem we need to address in the multi-step case is repeated counts of the same vehicles across individual links: a way to determine the number of counts needs to be devised. For this, we define the *link count matrix* Γ, where each element $\gamma_{ij,o} = \tau_{j,o}^- - \tau_{i,o}^+ + 1$ denotes the number of times a flow originating in o will be counted through link (i, j). For computational reasons, this must be set to:

$$\gamma_{ij,o} = \begin{cases} 1 & \text{if } \tau_{j,o}^- = \infty \text{ or } \tau_{i,o}^+ = \infty \\ \max\left(\tau_{j,o}^- - \tau_{i,o}^+ + 1, 1\right) & \text{otherwise} \end{cases}$$

since the result is meaningless when link (i, j) does not appear in any shortest path.

The mapping from O flows to OD flows is modified as follows:

$$s_{od}^t = x_o^t \left(\sum_i \frac{p_{id}}{\gamma_{id,o}} - \sum_j \frac{p_{dj,o}}{\gamma_{dj,o}} \right) \tag{3.13}$$

Regarding the constraints in the optimisation problem, the repeated link counts constraint cannot be enforced in the single-step model.

$\text{C3}_{\text{sp,ss}}$:

$$\sum_{i \neq o} \frac{p_{oi,o}}{\lceil c_{oi} \rceil} = 1, \ \forall o$$

Notice that one could use $\gamma_{oi,o}$ instead of $\lceil c_{oi} \rceil$.

$C5_{sp,ss}$:

$$\sum_j \frac{p_{ji,o}}{\gamma_{ji,o}} - \sum_k \frac{p_{ki,o}}{\gamma_{ki,o}} \geq 0, \ \forall o \text{ and } \forall i \neq o$$

$C6_{sp,ss}$:

$$p_{ij,o} = 0, \forall i, j, o \text{ such that } f_{i,o} + c_{ij} > f_{j,o}$$

This concludes the modification of the framework to accommodate for shortest path assignment of flows, and its extension to non-unitary link costs that translate the time spent by vehicles to traverse links. Let us now briefly study and discuss the relevance of the O flow based single-step and multi-step models, when applied to practical situations.

3.8 Applicability of O flow based frameworks

In this section, nodes in a simple network are associated with basic urban elements: the purpose of this is to express usual behaviour of users, the flows and assignments that derive from it, and qualitatively study the best procedure to accurately estimate OD flows in a simplified real-world scenario using an O flow based framework.

Let us define a simple urban area composed of two residential zones R_1 and R_2, a city centre C, a school S and a working zone W, where offices might be located, for instance. Congestion is not taken into account, and vehicles are assumed to take the same amount of time to traverse the same link. The resulting network is depicted in figure 3.7.

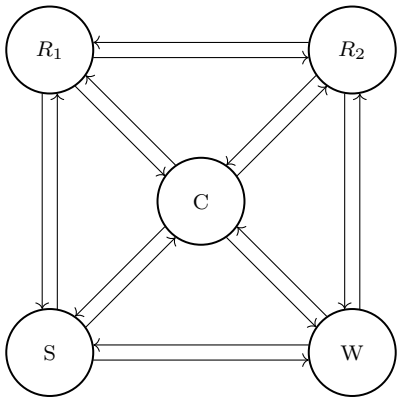


Figure 3.7: Simple urban network

At this point, the remarks on fixed O flow proportions conducted in subsection 2.2 are particularly relevant: it was stated that vehicles departing from an origin over the observation time horizon necessarily choose destinations in fixed proportions, by definition of the assignment matrix P or $\{P^\tau\}$. This remark can actually be extended by saying that because proportions are defined for all links, then origin flows also choose paths in equal proportions.

Let us first consider the multi-step model: it is obvious that depending on the time of the day, vehicles will be directed to destinations in different proportions. For example, from 5am to 6am

on a weekday, we might expect more users to be directed to W than to any other node, while from 7am to 8am we could have many vehicles travelling to node S as well. A possible scenario here is the following: assume vehicles take one time step to traverse one link, for simplicity, then

- At 6am: $p^2_{CW,R_1} = 0.9$ and $p^1_{R_1S,R_1} = 0.1$ since the majority of people leaving residential areas at this time are expected to be headed to their workplace
- At 7am: $p^2_{CW,R_1} = 0.4$, $p^1_{R_1S,R_1} = 0.6$ since people leaving residential areas at this time are expected to be headed to work or school with approximately equal proportions

The same reasoning may be applied to evening flows, and to all flows in general. Clearly, a fixed assignment matrix over the whole day does not exist: the departure time inherently shapes the destinations users are going to, and more information is needed to represent the situation. By defining $P^{t,\tau}$ as the assignment matrix equivalent to P^τ, but for flows that departed during time interval t, we end up with more assignment matrices and unknowns, but with the possibility of correctly representing the users' behaviour.

A possible solution may lie in a level set partitioning of the observation time horizon, similar to the one defined in [7] and described in section 2.4.3 of this report. Recall that level sets $M_t = \{t' : \mu(t') = \mu(t)\}$ of the map $\mu(t)$ were defined such that O flows leaving at a time in M_t would choose paths in equal proportions, or expressed mathematically: $r^t_{\boldsymbol{p}|o} = r^{\mu(t)}_{\boldsymbol{p}|o}$. We can modify it and define level sets such that O flows leaving at a time in M_t would flow through links in equal proportions: $p^{t,\tau}_{ij,o} = p^{\mu(t),\tau}_{ij,o}$ for all time steps $\tau \leq \tau_{\max}$. Notice this is a modification in formulation, but the idea and fixed-proportions assumption remains essentially the same. For instance, if O flows choose paths with equal proportions, it follows that they also choose links in equal proportions, since flowing through a path results in flowing through a pre-determined set of links.
The idea is to partition the whole set of time sampling intervals (obtained over several days) into a discontinued observation time horizon that relates to a particular time period: in that case, one may fix the assignment matrix for such flows without any loss of behavioural meaning. Say, for example, that we wish to know OD flows at 5am on weekdays: setting $t = 5$am on weekdays as the time category we wish to study, we can estimate $\{P^{t,\tau}\}$ and the corresponding O flows $\{\boldsymbol{x}^t_o\}$ by using all link counts $\boldsymbol{y}^{t'}$ such that $t' \in M_t$.

Regarding the single-step model, we now deal with link counts obtained over prolonged periods of time: as such, major differences in O flow proportions may be noticed between weekdays and weekends, or in the worst case scenario, between each day of the week. For example, during weekdays we would expect the majority of flows to be between R_1, R_2 and S, W while during weekends one is likely to observe smaller proportions of flows from R_1, R_2 to S, W and larger proportions between the two residential areas and to the city centre. Again, by considering the level sets M_{Monday}, M_{Tuesday} and so on, one would be able to work with fixed proportions P^{Monday}, P^{Tuesday} by using link counts from these specific categories. Note that the precision of the division carried out is variable and can accommodate the needs of the estimation in

question, for both models: for instance, one might wish to distinguish between school periods and holiday periods, obtaining a more fine-grained definition of level sets, or on the contrary may simply wish to consider weekdays and weekends, obtaining only two level sets in the single-step case.

This division of time intervals in level sets was proved to be working on a real dataset [7] (for weekdays only). However, in the paper just cited, a path flow based model was used: application of the O flow based framework, given this division in similar time periods is well studied and executed, is promising and might lead to a great reduction in complexity of inquired flows with negligible loss of performance. Considering shortest path assignment, this would bring the complexity down from $\mathcal{O}(n^2)$, which is the number of viable shortest paths in the network, to $\mathcal{O}(n)$.

4. Implementation

After finalising the modifications and extensions of the framework on a theoretical level, simulations were required to determine their effectiveness in OD estimation, and to study how parameters affect the results obtained. In this section, the implementation of the generator and solver is detailed.
The original simulations presented in [8] were developed in Matlab by the authors. The code consisted of three main scripts, one for bi-directional networks, and two for uni-directional networks with and without sparsity assumption. All models were multi-step. These main scripts called several helper functions and sub-scripts for generation of assignment matrices and flows, mapping from O flows to OD flows, optimisation of $\{P^t\}$ and $\{\boldsymbol{x}_o^t\}$, and other subtasks such as obtaining the constraints on $\{P^t\}$ as Boolean matrices. The optimisation tool employed was CVX [29, 30], a Matlab package developed to formulate and solve convex programs. In the later subsections, it will be mentioned whenever the original code was used without considerable modifications, or when the implementation of a certain component was directly inspired by an existing piece of code.

4.1 Generator

It was decided to rewrite the generator from scratch due to a lack of composability and extendibility inherent to Matlab scripting, when compared to other languages supporting more advanced features in object-oriented programming. Python was chosen for several reasons: first, it is a readable and easy-to-pick-up language, particularly suited to the quick development of research projects such as this one. Moreover, the Python community has developed over the years fully functional libraries that automate and facilitate many tasks one would otherwise have to go manually through in languages such as Matlab. A concrete example is the `NetworkX` package [31], developed for the creation and manipulation of networks.

The initial idea was to port the whole framework to Python, including the solver. However, `scipy.optimize`, the optimisation library that was tried, was slower than the existing Matlab/CVX routine by several orders of magnitude, even for relatively small inputs. Other popular options such as `PyTorch` or `TensorFlow` could have been looked into, but it was chosen to do so at a later stage, if possible, and to focus instead on experimentally proving that the framework was functional. A high-level schematic overview of the resulting code can be seen in figure 4.1. (Appendix B contains all the main Python code and a link to an online repository containing all the code used in the project). Note that the `Solver` class, which will be presented in detail in the next subsection, only deals with the creation of the constraint matrices and would have hosted the optimisation routines if these had been coded in Python, as the original design had envisioned.

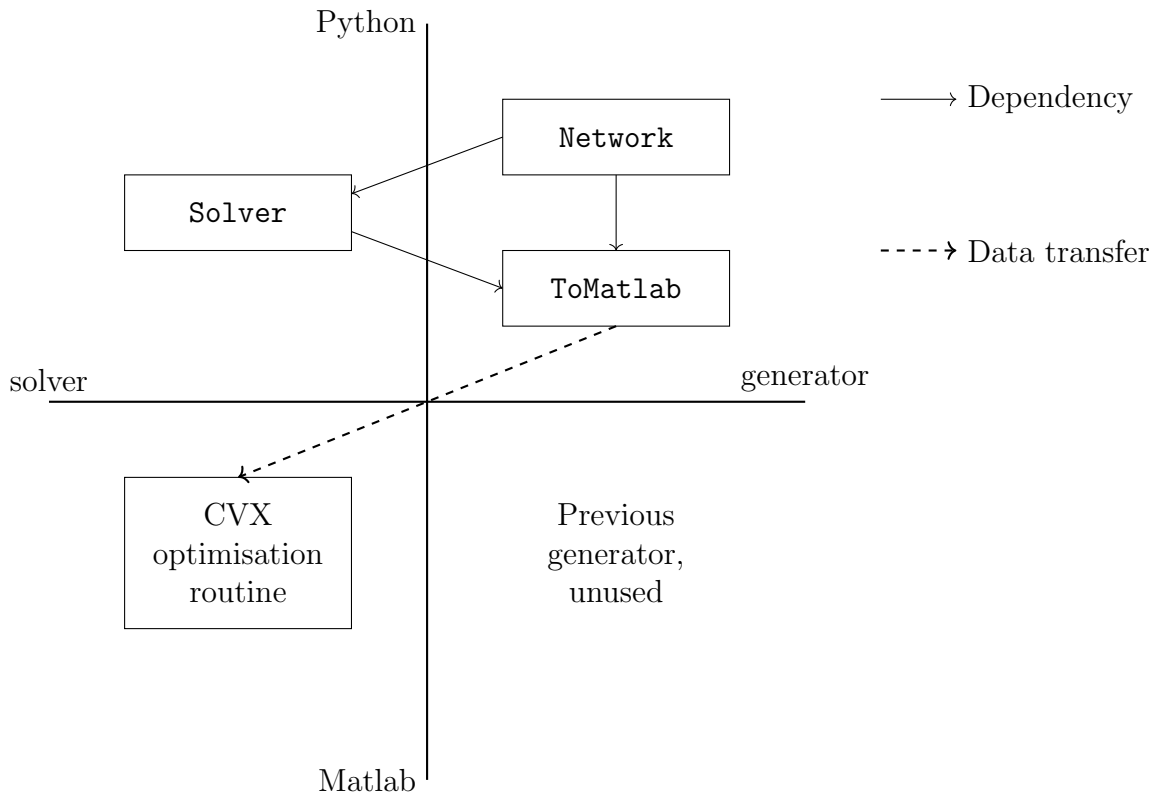


Figure 4.1: Schematic overview of implemented system

As can be seen in figure 4.1, the `Network` class is the basic data structure in the application. `Network` instances will contain all information about the network topology and expected user's behaviour that is needed to derive flows and assignment matrices. These are generated and stored as members of the network's instance. While developing code, special emphasis was placed on extendibility, so that new types of assignment, flow generation, or link costs could easily be added. Listing 1 shows how a network and its respective assignment matrix can be generated: in this case, a simple 3 by 3 bidirectional network with real link costs is constructed, and an assignment is computed.

```
1 from network import Network
2 net = Network(uni_bi='bi', h=3, w=3)
3 net.assign_link_costs(costs='real')
4 net.find_all_paths(tau_max=4, assignment='shortest_path')
5 net.generate_od_pairs()
6 net.compute_path_assignment_matrix()
7 net.generate_random_proportions()
8 net.compute_assignment_matrix()
```

Listing 1: Setting up a network with a specific assignment

The class interface allows for additional types of assignments to be added and more specific subsets of paths to be determined whenever this is needed by the user. A few class methods are worth explaining in more detail:

- `find_all_paths` currently allows the user to set a maximum path length and an assignment type. All paths computed are loop-free, although a version containing loops could easily be introduced. The two assignments supported are `shortest_path` and `random`: the latter corresponds to the loop-free assignment used in the paper (loop-free paths whose length is less than `tau_max` are chosen). Link costs must be assigned prior to the path search, as cumulative path cost is used to determine a path's viability.

- Random proportions are generated using the same strategy as the one set out in [8]: each path is assigned a certain probability to be chosen by vehicles travelling in the network. Then, one can find the total proportions of OD flows by summing the probabilities of the paths that go from a certain origin to a certain destination. Finally, one can find the total proportions of O flows by summing the OD flow probabilities. This information is stored as instance attributes and used in `compute_assignment_matrix` to eventually obtain the assignment matrices.

An important modification that was made here concerns the computation of the O flow assignment matrix from the proportions: this requires knowledge of the OD flow assignment matrix, which itself is built using the path assignment matrix. In the rigid setting, the latter is easy to compute, since the n-th link in a path will be traversed during the n-th step. This is not necessarily the case when the link costs are real. The elements of the multi-step path assignment matrix Δ^τ are therefore computed as follows:

$$\delta^\tau_{ij,\boldsymbol{p}} = \begin{cases} 1, & \text{if } \boldsymbol{p} \text{ contains link } (i,j) \text{ and } \tau^+_{i,o} \leq \tau \leq \tau^-_{j,o} \\ 0, & \text{otherwise} \end{cases} \tag{4.1}$$

where o is the first node of path $\boldsymbol{p}$.

Another class written in Python is `ToMatlab`, which handles the conversion of `numpy` structures to Matlab readable data in `.mat` format. The purpose of this class is to prepare data that will be read by a Matlab script for OD estimation simulation. As such, apart from a `Network`'s instance that needs to be passed to the constructor, the user can set whether the simulation will be over a single or multi-step model, how many trials should be performed, and other technical details such as the directory where to save the data. Once conversion is over, the solver is ready to estimate the OD flows based on link counts.

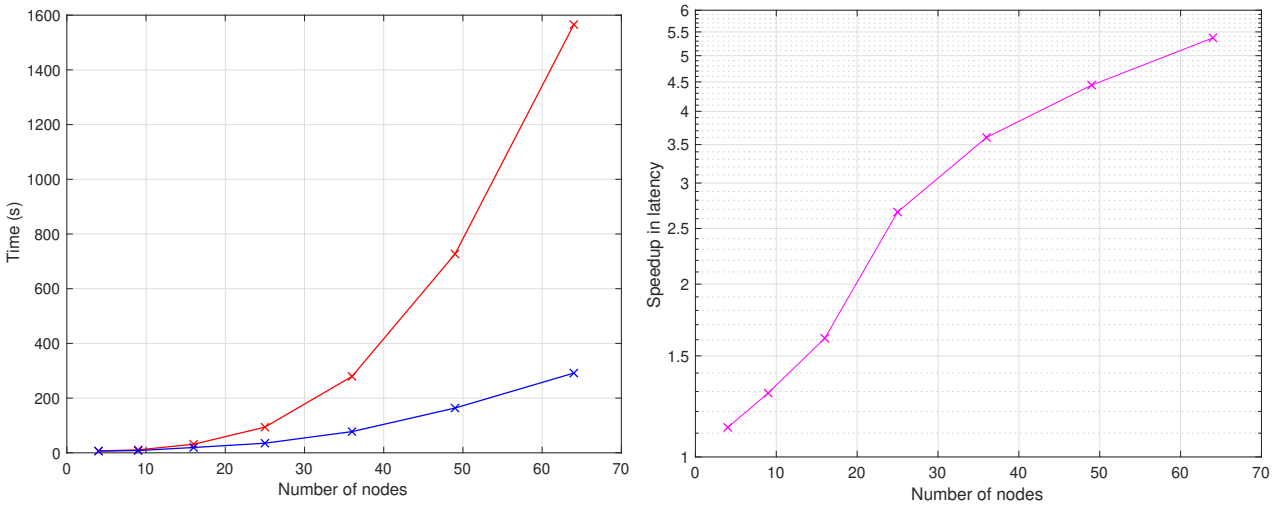
4.2 Solver

Our OD estimation problem was implemented as a CVX constrained problem, which consists of three elements: a variable matrix, an objective function to optimise, expressed in terms of the variable matrix, and a set of equality and inequality contraints again expressed in terms of the variables. The first two elements just mentioned are defined by the problem, and apart from the variable matrix dimensions, they remain unchanged throughout all the simulations of the framework. On the other hand, the constraints, defined for our problem in section 3, vary

with the network's topology and traffic assignment used. The next subsections provide insight into the optimisation algorithm and the structures used to build and store the constraints.

4.2.1 Optimisation algorithm

Even if the framework was modified to accommodate other assumptions on traffic assignment and extended to more realistic network topologies, the objective function remains (2.25) for the single-step model and (2.26) for the multi-step one. The problem is also intrinsically the same, that is, a bi-convex problem solved iteratively by optimising for $\boldsymbol{x}_o$ with P fixed, and vice-versa, until a threshold in NMSE_y is reached. As such, implementing the algorithm in the single-step case simply consisted in re-adpating the already existing one for the multi-step models. However, a considerable speed-up of the simulations was achieved thanks to the removal of unnecessary matrix multiplications with large dimensionality (tens of thousands of elements) throughout the optimisation routine. The results obtained when comparing the two systems are presented in 4.2: 10 iterations of the algorithm were run for different n by n bi-directional networks under loop-free rigid traffic assignment, with $\tau_{\max} = 4$, and over an observation period $n_T = 120$. It can be seen in 4.2a that the time taken to run the algorithm is superlinear in both cases. However, the latency speed-up plotted in 4.2b shows that the performance improvement achieved also increases with the input size.



(a) Time spent on 10 iterations of the algorithm versus number of nodes in the network

(b) Speedup in latency versus number of nodes in the network

Figure 4.2: Performance comparison for original and optimised code

4.2.2 Constraints representation

As was mentioned at the beginning of this section, the constraints depend on the network's topology and the type of traffic assignment. The non-negativity constraint, the only constraint enforced on $\boldsymbol{x}_o^t$, is trivial to implement. On the other hand, constraints on P^t require more thought, both in the process of obtaining them, but also in choosing the data structures that will store them for usage in CVX. This latter design task must actually be performed first, since

obtaining the constraints is tedious but reduces itself to carefully following the formulations and translating these into code.
In the original code, constraints 2 to 5 were created as Boolean matrices that would index those elements of $\{P^t\}$ that figure in each condition: this logical indexing is adapted to conditions on individual elements as well as conditions involving sums over nodes. Whenever the sum involves other requirements regarding nodes and time steps, such as in the flow and repeated link counts constraints, the implementation is more complex.
Regarding flows, the condition is expressed as a sum over nodes, for all origins. As such, a single Boolean matrix would not suffice to collect this information. Four components were employed in the original code to represent different aspects:

- `in_edges` and `out_edges`: this matrix tells, for each node in the network, the links that flow into it and out of it, respectively
- `out_check` and `out_edges`: since not all O flows participate in the inflows and outflows of every node (for example, origin o does not participate in the inflow in node o), these matrices tell which origins should be counted when summing elements of P to calculate inflows and outflows, respectively

This structure was satisfactory and was thus retained in the new code, the construction simply being modified to take shortest path elements into account.
Finally, the repeated link counts constraint had to be implemented from scratch: the condition expressed an equality involving all elements of P, for all time steps. It was decided not to use a Boolean matrix in this case, but to store information about arrival and departures of origin flows for all nodes, giving for every link (i, j):

- `enter_link` contains the time step at which each O flow enters link (i, j), that is, $\tau^+_{i,o}$
- `end_link` contains the time step at which each O flow leaves link (i, j), that is, $\tau^-_{j,o}$

When $\tau^-_{j,o}$ is greater or equal than $\tau_{\max}$ or if $\tau^+_{i,o}$ is less or equal than $\tau^-_{j,o}$, then no constraints are retained for link (i, j). The equality constraints forming C7 can now be implemented in CVX as a loop on all links, on all origins, and all time steps between the corresponding `enter_link` and `end_link` time steps.

4.3 Testing

Let us now decribe the strategy that was set out for testing the framework and ensuring its correct functionality. The newly written Python code had to generate correct assignment matrices and obtain relevant constraints: this setting was adapted to test-driven development, due to the mutual dependence of the two aforementioned elements. As such, a series of tests involving the satisfaction of constraints were written before programming the actual system (the full testing code can be found by following the link in appendix B). As the code was gradually verified, simple constraints that were not satisfied gave information about corner cases in the

generation that needed handling, while simple assignments that were proven to be correct and that did not pass the more complex tests showed that issues were present in the implementation of the corresponding constraints. Other lower level tests included checking that all paths were shortest paths, when this type of assignment was selected by the user, and that all paths were shorter than $\tau_{\max}$, by looking directly at the paths computed when generating the assignment matrices, but not at the assignment itself. Figure 4.3 shows a schematic representation of this process.

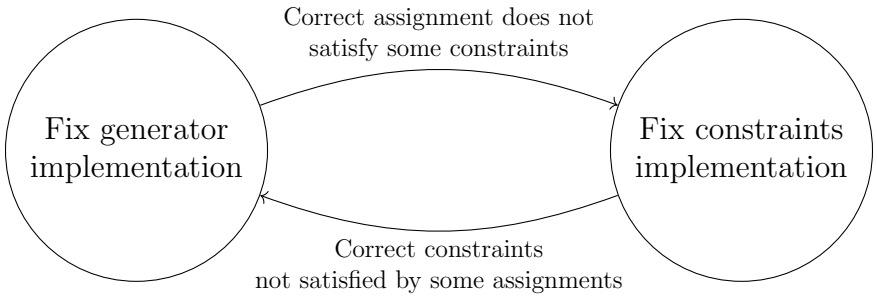


Figure 4.3: Testing process followed for developing generator and constraints

This process was applied to all combinations of network topologies and traffic assignment that were relevant to this project, for different values of $\tau_{\max}$, different network sizes and different link costs. The final tests that resulted from this were run on 1000 randomly generated assignments in each scenario to verify the framework's correctness.

The framework was also tested empirically, by comparing the results obtained for multi-step models of uni-directional and bi-directional networks with the original findings in the paper and the output of the original code. Because a comparison with the initial framework was desirable, it was important to ensure that for the same input parameters, the accuracy and precision of the estimation were matching. Random proportions of path flows were thus obtained from a normal random distribution shifted to ensure strict positivity, and P^t was initialised to a random, but viable assignment that satisfies all constraints, as was done when obtaining the results presented in [8].

5. Results

The generation of the synthetic data used in the following simulations was detailed in the previous section. We are now concerned with measuring the reliability of the extended framework: the OD estimates obtained through our model and objective function's optimisation need to be quantitatively analysed and compared to the results of the original framework developed in [8], that will be used as a benchmark.

5.1 Measuring performance

There exist different ways one can quantitatively assess a framework for OD estimation. [13] provides an overview of several measures of reliability. Define $\hat{\boldsymbol{s}}$ as the estimate of the true demand flows $\boldsymbol{s}$, and $\mathcal{N}_{OD}$ as the set of possible OD pairs, with cardinality $n_{OD} = |\mathcal{N}_{OD}|$. Provided one has access to the ground-truth OD flows, which is the case for simulated data and potentially in some datasets for traffic assignment, statistical measures that can be used include relative error, total demand deviation, mean absolute error and root mean square error. In the following subsections, it was decided to use relative error and total demand deviation, both defined below, as they bring complementary insight into the simulations' outcome.

- Relative error (RE):

$$\mathrm{RE} = \sqrt{\frac{1}{2} \sum_{od \in \mathcal{N}_{\mathcal{OD}}} \left(\frac{\hat{s}_{od} - s_{od}}{s_{od}} \right)^2} \tag{5.1}$$

  One can notice that whenever an OD flow s^t_{od} has a near-zero value, the relative error $\frac{\hat{s}_{od} - s_{od}}{s_{od}}$ can become large: therefore, achieving a low total RE implies that the framework accurately estimates flows of all magnitudes. RE is nevertheless undefined for OD flows whose true value is zero, which makes it less informative when the demand matrix is sparse, since only error for the non-zero flows will be taken into account. Because RE can be defined over individual flows as $\mathrm{RE}_{s^t_{od}} = \frac{\hat{s}_{od} - s_{od}}{s_{od}}$, this will be used to plot histograms of the results in order to analyse the errors' distributions.

- Total Demand Deviation (TDD):

$$\mathrm{TDD} = \frac{\left| \sum_{od \in \mathcal{N}_{\mathcal{OD}}} \hat{s}_{od} - \sum_{od \in \mathcal{N}_{\mathcal{OD}}} s_{od} \right|}{\sum_{od \in \mathcal{N}_{\mathcal{OD}}} s_{od}} \tag{5.2}$$

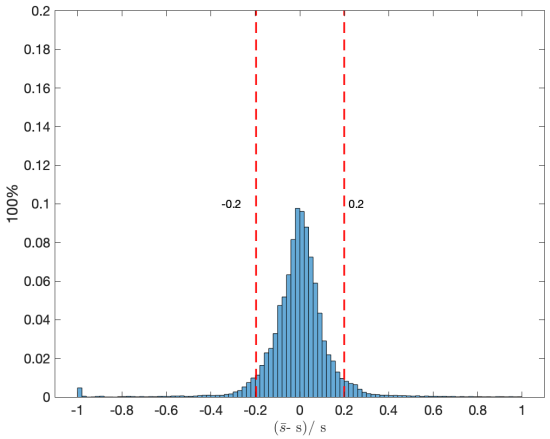
  TDD penalises the overall deviation of the estimated flows from the ground-truth data: small OD flows do not produce larger errors, but non-zero deviations from null ground-truth flows will increase it. It is usually considered to provide a good measure of the quality of the estimated OD matrix.

5.2 Simulations

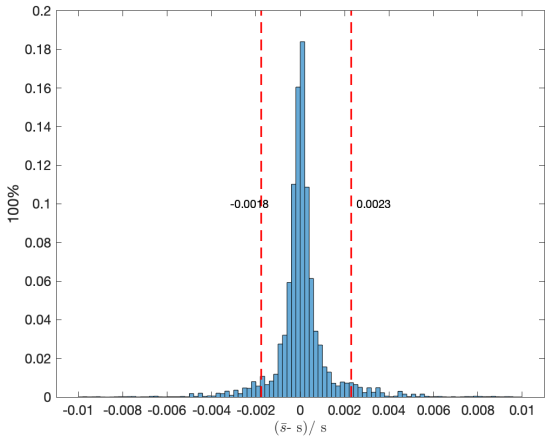
The scope of the first simulations that were carried out was to compare the performance of the modified framework to the original one's, and thus confirm experimentally that matching solver and generator by preventing impossible solutions effectively results in an improved estimation of OD flows. For this, assignments over a simple 3 by 3 bi-directional network (same nodes as the 3 by 3 uni-directional network in figure 2.1, but with directed edges in both directions between nodes) were simulated. The following parameters were not changed throughout these initial experiments:

- Observation time horizon n_T: 60. However, only link counts for $\tau_{\max} < t < n_T - \tau_{\max}$ are used: link flows observed outside this interval are explained by O flows outside the observation time horizon and are not used in the optimisation process. However, because all O flows observed explain the link counts between $\tau_{\max}$ and $n_T - \tau_{\max}$, we still obtain n_T sets of OD flows in the end
- Trials: 10
- NMSE_y: 10^{-5} for the single-step model and $5 \cdot 10^{-5}$ for the multi-step model
- Maximum number of iterations: 1500
- $\tau_{\max} = 4$, which corresponds to the shortest path between the most distant nodes in the network (i.e. pairs 1,9 and 3,7). This always results in the same 72 OD pairs for the rigid traffic model with unitary link costs, but is likely to rule out some pairs for real link costs
- O-flows are assigned a value from a normal random distribution in the open interval $(0, 1)$

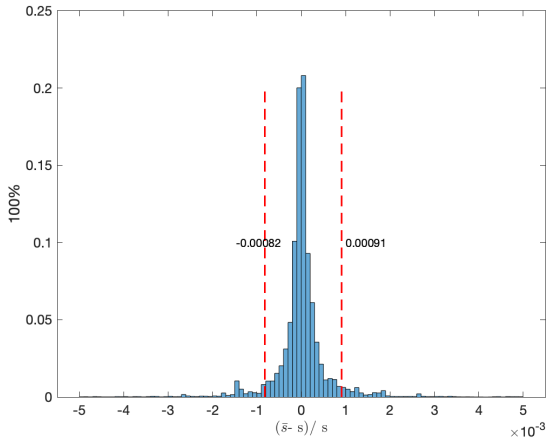
For the newly introduced framework, link costs were assigned a random value from a uniform distribution in the interval $(0, 4)$: as such, we ensure that all links will carry some flow, and that none of the O-flows will be zero. Some OD flows may be zero: this is not a problem, since fractions of O flows in links leading to such impossible destinations will be set to zero by the speed constraint (the shortest path constraint, as we have defined it).



(a) Original framework: loop-free assignment



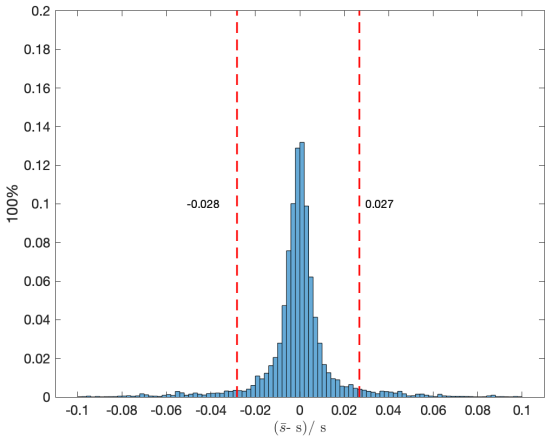
(b) Shortest path assignment, unitary link costs



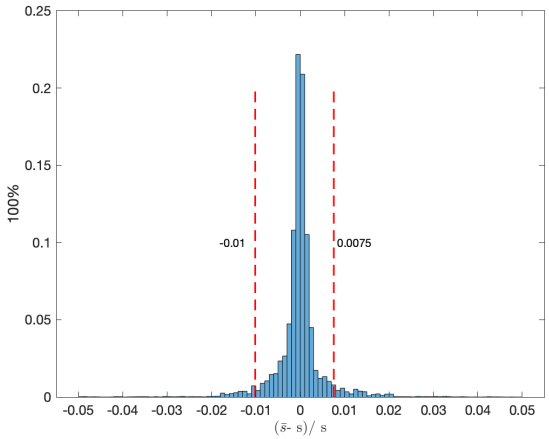
(c) Shortest path assignment, real link costs

Figure 5.1: Comparison of framework performance for the single-step model: histograms of relative errors for a 3-by-3 bidirectional network

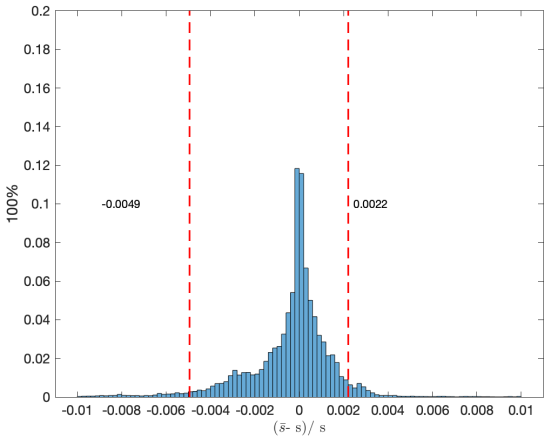
As can be seen in figure 5.1, the estimation's performance is largely improved when considering shortest path assignment of flows, proving experimentally that the performance of the solver is affected when impossible solutions are considered. 5% of estimated OD flows have relative errors lower, respectively higher, than the values indicated by the left, respectively right, red vertical lines plotted over the histograms. In the original framework, 90% of flows have relative erros between -0.2 and 0.2, while in the new framework, 90% of flows have relative errors between -0.0018 and 0.0023: blind estimation can therefore be achieved by considering shortest path assumption. When real link costs are introduced, the results plotted in subfigure 5.1c give an even larger improvement: 90% of flows have relative erros between -0.00082 and 0.00091.
In terms of total relative error, RE as defined in (5.1) drops from 22.5650 for the loop-free assignment to 0.9046 and 0.0857 for the shortest path assignment with rigid and real link costs respectively. TDD, on the other hand, also decreases from 9.5509e-04 for the original framework to 7.4948e-05 and 4.6841e-05 for rigid and real link costs in the modified framework: the improvement is still present, although less remarkable.
The shortest path assignment outperforms the loop-free assignment in the multi-step model as



(a) Original framework: loop-free assignment



(b) Shortest path assignment, unitary link costs



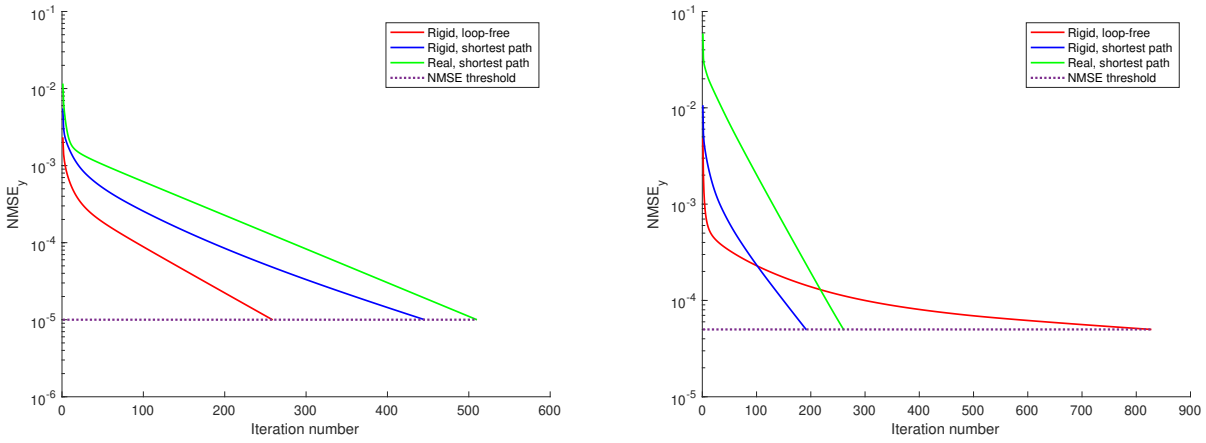
(c) Shortest path assignment, real link costs

Figure 5.2: Comparison of framework performance for the multi-step model: histograms of relative errors for a 3-by-3 bidirectional network

well. However, as can be seen in sub-figure 5.2a, the baseline estimation is already roughly 7.4 times more accurate than in the single-step case. We notice a 3-fold improvement in relative error distribution when flows follow a shortest path assignment with unitary link costs, and slightly less than a 6-fold improvement when real link costs are introduced: blind estimation is maintained.

Plotting NMSE_y at each iteration of the optimisation algorithm yields the results plotted in figure 5.3. In the single-step model, the algorithm in the new framework converges slower than in the original one. On the other hand, convergence is much faster for the new framework in the multi-step case. This difference might be explained by the fact that the constraints imposed on the variables are essentially the same in the single-step model, the only notable difference being that more elements of P^t are set to zero by the speed constraint, while in the multi-step case, a whole new equality constraint on repeated link counts is introduced, further reducing the feasible set.

Let us now proceed with a more insightful interpretation of the results obtained so far. A first



(a) Single-step model **(b)** Multi-step model

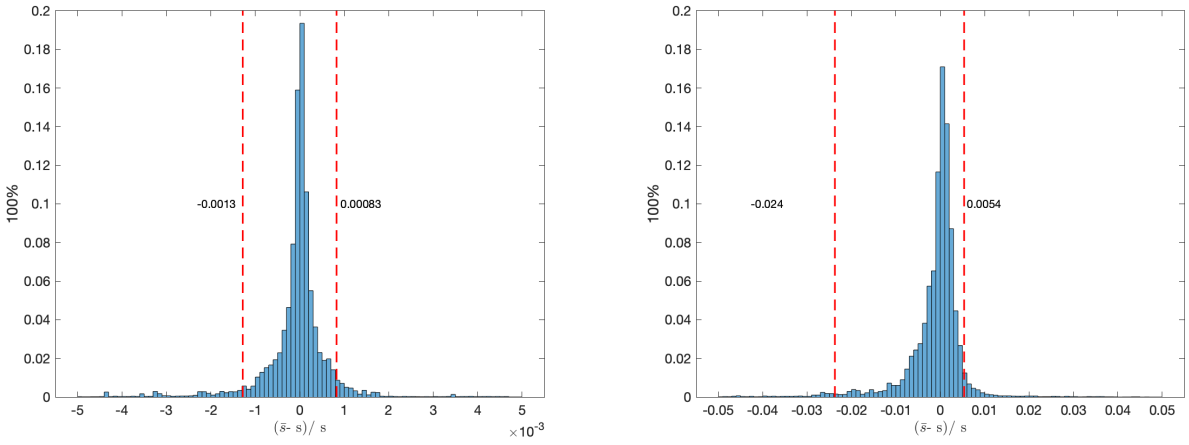
Figure 5.3: Comparison of convergence of solvers for the original and modified framework

remark is the small peak in distribution that appears in sub-figure 5.1a at $\mathrm{RE}_{s_{od}^t} = -1$: around 0.39% of estimated OD flows (168 out of 43200) produce such a relative error. $\mathrm{RE}_{s_{od}^t} = -1$ implies that $\hat{s}_{od}^t = 0$. This peak is also present in the other histograms, at varying proportions, but was not shown for clarity: we suppose that the solver in both frameworks has an overtendency to assign values of zero to OD flows, with respect to ground-truth data. This tendency is most certainly due to the numerical methods employed by CVX, and attenuating it by modifying the objective function is unlikely to noticeably improve the estimation results. A second tendency of the solver is present in the multi-step model: the distribution of relative errors is not centred around zero, the mean is negative in the 3 cases and this is particularly evident in sub-figure 5.2c: there are more underestimated OD flows than overestimated ones. This means that link counts tend to be explained by longer path flows. Again, this is not a cause for concern as the overall estimation is highly accurate.

Finally, an important remark lies in the fact that real link costs bring an improvement in performance over rigid link costs, which might be somewhat counter-intuitive at first. However, the average link cost is 2 and $\tau_{\max}$ was kept equal to 4 throughout the simulations. This means that some OD flows might be null. Indeed, out of 72 possible OD pairs, present in the unitary link costs' case, the simulation for the single-step model with real link costs contained only 55 OD pairs out of the possible 72, and the multi-step one only 49 out of 72. Also, as was briefly mentioned in the beginning of this susection, increasing $\tau_{\max}$ decreases the actual number of full time samples that can be used in the estimation. The next simulations involve changing the maximum path length so that all shortest paths actually carry traffic, in order to provide a fairer comparison with the original framework.

Unsuprisingly, the performance drops slightly in both models. The single-step model still greatly outperforms the original framework, while the tendency to underestimate becomes even clearer in the multi-step case, where the 5% lower bound relative error almost reaches the baseline value. This might be explained by the fact that the algorithm is still converging at a high rate when reaching the threshold in NMSE_y, as can be seen in sub-figure 5.3b.

The final experience conducted aims to find a relationship between quality of OD estimates



(a) Single-step model **(b)** Multi-step model

Figure 5.4: Histograms of relative errors for a 3-by-3 bidirectional network when all OD pairs are considered

and simulation parameters: to this end, we vary observation time horizon while keeping other parameters fixed. The simulation consisted in 1 trial for 100 iterations of the algorithm at every value of the varying parameter, in a 4 by 4 bidirectional network. The observation time horizon was set to 60 samples, and no limit was imposed on the NMSE as the purpose was to keep the number of iterations fixed. $\tau_{\max}$ was set automatically to consider all possible OD pairs between the 16 nodes.

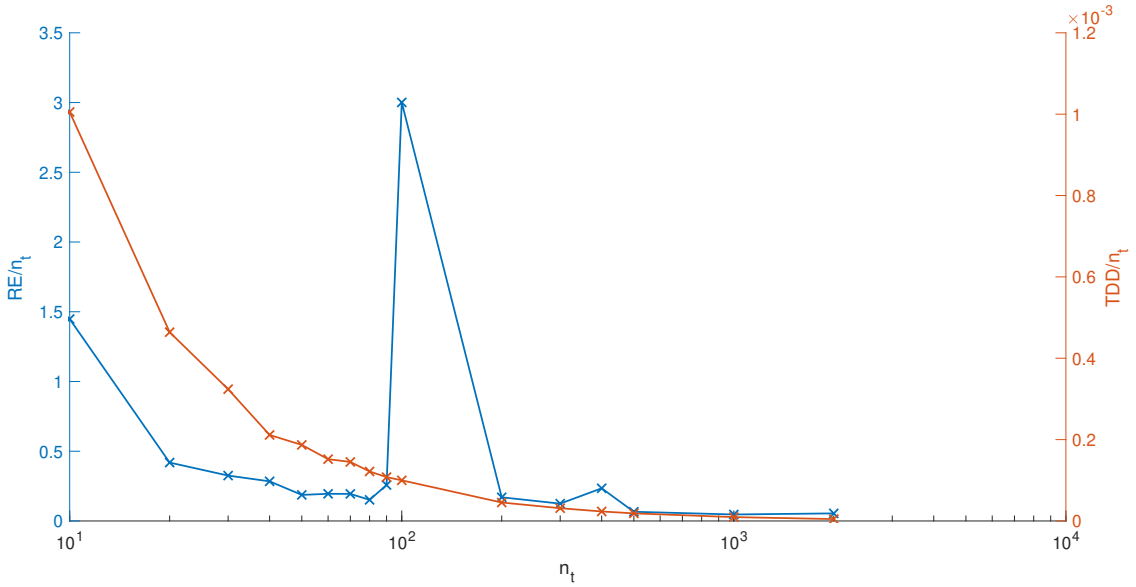


Figure 5.5: Average RE and TDD per time sample versus observation time horizon

Figure 5.5 shows how estimation quality varies as the observation time horizon n_T increases from 10 to 2000 samples. In this case, both error measures were averaged over time samples: because RE and TDD are calculated over all OD flows, increasing the observation time horizon means more elements are considered in the sum, and a higher error results from this. Looking at the average error per time sample is more insightful. As we can see: increasing the observation time horizon brings more independent equations and the problem is solved more and more

precisely, when all other parameters are fixed, which is coherent with theoretical expectations. The peak appearing at $n_T = 100$ shows one of the limitations of RE with respect to TDD when calculating overall error: the presence of outliers due to near-zero ground-truth OD flows may result in a RE that does not correctly represent the quality of estimation. TDD and relative error distributions are more robust ways of testing the framework, and both yield satisfactory improvements over the original framework.

6. Evaluation

The initial objective of this project was to alleviate the negative effects of loop flows on the performance of O flow based solvers for OD estimation. Extensions to make the framework more applicable to real-world networks were also sought after.
On the analytical level, it was shown that impossible solutions could be found in the single-step and multi-step model, which inevitably degraded quality of the estimated OD flows. A contribution of this project was to derive conditions on the true and estimated assignments in order to match generator and solver, which was not the case in the original framework. A shortest path assignment assumption was employed to achieve this desirable property. Based on this assumption, we were also able to extend the models to account for real link costs, which are much more versatile in representing the topology of general uncongested networks.
It must be said that the shortest path assumption is not directly relevant to physical transportation networks. However, it makes sense in an uncongested setting, and weaker formulations of the assumption have been employed to estimate real-world OD flows [7, 20].

From an experimental point of view, the results obtained by simulation are satisfactory and show noticeable improvement in estimation accuracy, especially in the single-step model. Observation periods are generally of the order of days in this type of models, which might lead one to argue that it is less relevant than its multi-step counterpart, more informative by nature. However, single-step traffic models still have their place in important studies conducted around the world. For example, daily link counts were used among other data in a 2008 California PATH (Partners for Advanced Transit and Highway) report *"to identify and to investigate locations within the state highway system where a relatively large number of collisions occur"* [28]. Furthermore, figure 6.1, taken directly from the report, mentions that the daily flows were obtained from accumulating 5-minute points, which means that knowing total traffic volume during selective days of the week was judged insightful, despite the fact that shorter time sampling intervals were available. This is just an individual example, but it corresponds to the exact scenario where improvements in evaluation and estimation methods can be highly beneficial to society.

While simulating the framework's performace, it was decided not to include real networks since the flows would still remain fully synthetic and the exact topology is unlikely to bring meaningful elements for analysis and discussion. A more insightful experiment that could not be conducted would have consisted in trying to estimate OD flows using the technique described in section 3.8. This would have implied obtaining link counts and OD flow data for a real transportation network, alongside the network's properties, such as free flow link travel times, or link length and speed limit, from which link costs for our framework could have been derived. However, existing open-source datasets providing access to this sort of information were not found. The closest match was Dr. Hillel Bar-Gera's Transportation Network Test Problem Repository (TNTP, available at: https://github.com/bstabler/TransportationNetworks) [32]. However,its purpose is to aid in the testing and development of algorithms for the *forward* assignment of

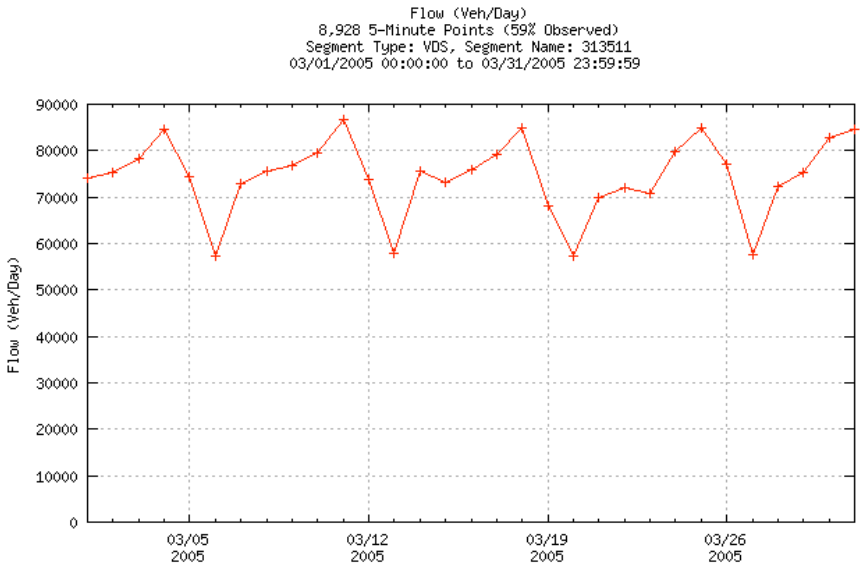


Figure 6.1: Day Traffic Flow on One Sample Data Station on I-80 [28]

link flows: that is, link flows, when given, are part of the best known flow solution to the assignment problem, and are not based on actual link counts. Data collected through cameras, sensors or loop counts (i.e. *real* data) is essential if one wants to study the applicability of a framework to practical real-world situations. Researchers usually partner with cities, private companies [7], governmental agencies [6] in order to use pre-existing data, instead of setting all the material needed to count traffic flows directly, which would be way too expensive for model validation: seeking such a partnership was judged outside the scope of this project. Also, almost all real networks in the TNTP dataset are considerably larger (only 1 network has less than 70 nodes) than what the current algorithm can handle in a reasonable amount of time. Finally, writing software to properly integrate this data format into the project would require a non-negligible amount of time.

7. Conclusion

Starting from a novel reduced-dimensionality framework for OD estimation, our analysis started with the study of loops in path flows and their prevention in the optimisation problem's solutions. A necessary and sufficient condition on the variables of the problem was found to guarantee that the estimated flows were generated via shortest path assignment, thus preventing the solver from converging to solutions containing loops, and most importantly allowing it to consider all and only viable assignments. The modified framework was then extended to allow for real link costs to be specified, which widens the potential applications to real world transportation networks. Indeed, assuming the network under study is uncongested, real link costs can translate aspects such as speed limits, road quality and link length. The framework was implemented in Python and Matlab to assess its usefulness in OD estimation: results were satisfactory and coincided with theoretical expectations, showing considerable improvements after shortest path assignment was assumed.

It is worth considering several directions for further work. First, the shortest path assumption is reasonable in uncongested networks, but relaxing it to include alternative paths within an arbitrary length range or to include elements other than free flow time would be of great benefit to the applicability of the framework, even when congestion is not taken into account. At this point, testing with physical data would be insightful in order to assess the performance of O flow based models on real-world networks. Regarding software, simulations currently require data transfer between two platforms. Fully implementing the framework in Python is essential for extendibility and seamless usability, and could result in an open-source tool that facilitates research in OD estimation. Finally, another sensible task would be to implement the solver using the ADMM (alternative direction method of multipliers) algorithm applied to bi-convex problems: this would likely result in a considerable speed-up in convergence, allowing for estimation to be performed on larger networks.

References

[1] L. G. Willumsen. Simplified transport models based on traffic counts. *Transportation*, 10(3):257–278, Sep 1981.

[2] William H. K. Lam and H. P. Lo. Estimation of origin-destination matrix from traffic counts: a comparison of entropy maximizing and information minimizing models. *Transportation Planning and Technology*, 16(2):85–104, 1991.

[3] H.P. Lo, N. Zhang, and W.H.K. Lam. Estimation of an origin-destination matrix with random link choice proportions: A statistical approach. *Transportation Research Part B: Methodological*, 30(4):309 – 324, 1996.

[4] Michel Bierlaire. The total demand scale: a new measure of quality for static and dynamic origin-destination trip tables. *Transportation Research Part B: Methodological*, 36(9):837 – 850, 2002.

[5] M. Bierlaire and F. Crittin. An efficient algorithm for real-time estimation and prediction of dynamic od tables. *Operations Research*, 52(1):116–127, 2004.

[6] Aditya Krishna Menon, Chen Cai, Weihong Wang, Tao Wen, and Fang Chen. Fine-grained od estimation with automated zoning and sparsity regularisation. *Transportation Research Part B: Methodological*, 80:150–172, 10 2015.

[7] D. Bauer, G. Richter, J. Asamer, B. Heilmann, G. Lenz, and R. Kölbl. Quasi-dynamic estimation of od flows from traffic counts without prior od matrix. *IEEE Transactions on Intelligent Transportation Systems*, 19(6):2025–2034, June 2018.

[8] J. Xia, W. Dai, J. Polak, and M. Bierlaire. Dimension reduction for origin-destination flow estimation: Blind estimation made possible. unpublished, 2018.

[9] Ennio Cascetta. *Transportation Systems Analysis: Models and Applications*, volume 29 of *Springer Optimization and Its Applications*. Springer US, Boston, MA, 2009.

[10] Yosef Sheffi. *Urban Transportation Networks*. Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1985.

[11] Ennio Cascetta. Estimation of trip matrices from traffic counts and survey data: A generalized least squares estimator. *Transportation Research Part B: Methodological*, 18(4):289 – 299, 1984.

[12] Chi Xie, Kara M. Kockelman, and S. Travis Waller. Maximum entropy method for subnetwork origin-destination trip matrix estimation. *Transportation Research Record*, 2196(1):111–119, 2010.

[13] Sharminda Bera and K. V. Krishna Rao. Estimation of origin-destination matrix from traffic counts: the state of the art. *European Transport Trasporti Europei*, 49(49):2–23, 2011.

[14] Rodric Frederix, Francesco Viti, and Chris M.J. TampÃšre. Dynamic origin-destination estimation in congested networks: theoretical findings and implications in practice. *Transportmetrica A: Transport Science*, 9(6):494–513, 2013.

[15] K. Ashok and M. E. Ben-Akiva. Alternative approaches for real-time estimation and prediction of time-dependent origin-destination flows. *Transportation Science*, 34(1):21–36, 2000.

[16] T. Toledo and T. Kolechkina. Estimation of dynamic origin–destination matrices using linear assignment matrix approximations. *IEEE Transactions on Intelligent Transportation Systems*, 14(2):618–626, June 2013.

[17] Hillel Bar-Gera. Traffic assignment by paired alternative segments. *Transportation Research Part B: Methodological*, 44(8):1022 – 1046, 2010.

[18] A. J. Hoffman and H. M. Markowitz. A note on shortest path, assignment, and transportation problems. *Naval Research Logistics Quarterly*, 10(1):375–379, 1963.

[19] Kodialam, Muralidharan S, James Orlin, and James B. The origin-destination shortest path problem. *Massachusetts Institute of Technology (MIT), Sloan School of Management, Working papers*, 01 1993.

[20] Refat Barbour and Jon D. Fricker. Estimating an origin-destination table using a method based on shortest augmenting paths. *Transportation Research Part B: Methodological*, 28(2):77 – 89, 1994.

[21] Yong Wang, Xiaolei Ma, Yong Liu, Ke Gong, Kristian C. Henricakson, Maozeng Xu, and Yinhai Wang. A two-stage algorithm for origin-destination matrices estimation considering dynamic dispersion parameter for route choice. *PLOS ONE*, 11(1):1–24, 01 2016.

[22] Ennio Cascetta and Maria Nadia Postorino. Fixed Point Approaches to the Estimation of O/D Matrices Using Traffic Counts on Congested Networks. *Transportation Science*, 35(2):134–147, May 2001.

[23] Hai Yang and Jing Zhou. Optimal traffic counting locations for origin-destination matrix estimation. *Transportation Research Part B: Methodological*, 32(2):109 – 126, 1998.

[24] Pierre Robillard. Estimating the o-d matrix from observed link volumes. *Transportation Research*, 9(2):123 – 128, 1975.

[25] Per Hogberg. Estimation of parameters in models for traffic prediction: A non-linear regression approach. *Transportation Research*, 10(4):263 – 265, 1976.

[26] T. V. Mathew and K. V. Krishna Rao. Lecture notes in introduction to transportation engineering, May 2007.

[27] Ennio Cascetta and Sang Nguyen. A unified framework for estimating or updating origin/destination matrices from traffic counts. *Transportation Research Part B: Methodological*, 22(6):437 – 455, 1988.

[28] Judy Geyer, Elena Lankina, Ching-Yao Chan, David Ragland, Trinh Pham, and Ashkan Sharafsaleh. Methods for identifying high collision concentration locations for potential safety improvements. 06 2008.

[29] Michael Grant and Stephen Boyd. CVX: Matlab software for disciplined convex programming, version 2.1. http://cvxr.com/cvx, March 2014.

[30] Michael Grant and Stephen Boyd. Graph implementations for nonsmooth convex programs. In V. Blondel, S. Boyd, and H. Kimura, editors, *Recent Advances in Learning and Control*, Lecture Notes in Control and Information Sciences, pages 95–110. Springer-Verlag Limited, 2008. `http://stanford.edu/~boyd/graph_dcp.html`.

[31] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using NetworkX. In *Proceedings of the 7th Python in Science Conference (SciPy2008)*, pages 11–15, Pasadena, CA USA, August 2008.

[32] Transportation Networks for Research Core Team. Transportation networks, 2018.

Appendices

A Integer link costs: mapping and constraints

Single-step

Mapping :

$$s_{od}^t = x_o^t \left(\sum_i \frac{p_{id}}{c_{id}} - \sum_j \frac{p_{dj,o}}{c_{dj}} \right)$$

C3 :

$$\sum_{i \neq o} \frac{p_{oi,o}}{\lceil c_{oi} \rceil} = 1, \ \forall o$$

C5 :

$$\sum_j \frac{p_{ji,o}}{c_{ji}} - \sum_k \frac{p_{ki,o}}{c_{ki}} \geq 0, \ \forall o \text{ and } \forall i \neq o$$

C6 :

$$p_{ij,o} = 0, \forall i, j, o \text{ such that } f_{i,o} + c_{ij} > f_{j,o}$$

Multi-step

Mapping :

$$s_{od}^t = x_o^t \left(\sum_i p_{id,o}^{f_{d,o}} - \sum_j p_{dj,o}^{f_{d,o}+1} \right)$$

C3 :

$$\sum_{i \neq o} p_{oi,o}^1 = 1, \ \forall o$$

C5 :

$$\sum_j p_{ji,o}^{f_{i,o}} - \sum_k p_{ik,o}^{f_{i,o}+1} \geq 0, \ \forall o \text{ and } \forall i \neq o$$

C6 :

$$p_{ij,o}^t = 0, \forall i, j, o, t \text{ such that } f_{i,o} + c_{ij} > f_{j,o}$$

C7 :

$$p_{ij,o}^t = p_{ij,o}^{f_{i,o}+1} \quad \text{for } t = f_{i,o} + 2, \ldots, f_{j,o}$$

B Python code

The following code is also available at `https://github.com/JBacchelli/FYP2019`, alongside testing code, visualisation code, Matlab code and simulations data, not included here.

Listing 2: network.py

```
"""
Network class is declared in this file. All meaningful elements of simulations
are effectively derived from an instance of a network.
"""

VISUALISE = False   # Set to True if working with Jupyter notebooks
                    # in order to visualise networks and assignment matrices
import numpy as np
import networkx as nx
from copy import deepcopy
if VISUALISE:
    from visualiser import Vis
np.set_printoptions(linewidth=150)

def mat_conv(M, v):
    """Performs convolution of assignment matrix with flow vector and returns
    result. len(M) zeros are prepended to the flow vector to obtain a result
    with len(v) elements.

    Arguments:
        M {list} -- Multi-step assignemnt matrix, from 0 to tau_max-1.
            List of 2-dimensional arrays. Each matrix in the list
            corresponds to one time step.
        v {list} -- Flow vector, from 0 to n_t-1 for full convolution.
            List of 1-dimensional arrays. Each flow vector in the list
            corresponds to one time sample.

    Returns:
        {list} -- List of flow vectors, from 0 to n_t-1. Each element in the
            list corresponds to one time sample.
    """
    n_t = len(v)        # the observation time horizon
    tau_max = len(M)    # the maximum path length
    full_flows = [np.zeros(len(v[0])) for _ in range(tau_max-1)] + v
    res = [np.zeros(M[0].shape[0])] * n_t
    for t in range(n_t):                    # 0 to n_T-1
        for tau in range(tau_max):          # 0 to tau_max-1
            res[t] = res[t] + M[tau] @ full_flows[t-tau+tau_max-1]
    return res

class Network():
    """
    Representation of a network as a set of nodes and links.
```

```
    A NetworkX graph object is used to leverage all the built-in functionalities
    and algorithms. https://networkx.github.io/
    """

    def __init__(self, uni_bi='bi', h=3, w=3, nodes=None, links=None, seed=None,
    ↪  verbose=False):
        """Builds network and checks validity of links with respect to nodes

        Arguments:
            uni_bi {str} -- Type of network to generate automatically, either
                'uni' for unidirectional networks or 'bi' for bidirectional
                networks. Set to None for passing nodes and links
                manually (default: {'bi'})
            h {int} -- Height of network (default: {3})
            w {int} -- Width of network (default: {3})
            nodes {int list} -- Pass list of nodes in the network manually
                (default: {None})
            links {(int*int) list} -- Pass list of links in the network manually
                (default: {None})
            seed {int} -- Random seed used throughout random generations
                (default: {None})
        """
        self.G = nx.DiGraph()
        if uni_bi:
            grid = nx.grid_2d_graph(h, w)
            self.G.update(grid)
            self.G = nx.convert_node_labels_to_integers(self.G)
            if uni_bi == 'bi':
                rev_edges = []
                for (n1,n2) in self.G.edges:
                    rev_edges.append((n2, n1))
                self.G.add_edges_from(rev_edges)
        else:
            self.G.add_nodes_from(nodes)
            self.G.add_edges_from(links)

        self.nodes = list(self.G.nodes)
        self.links = list(self.G.edges)
        if seed:
            np.random.seed(seed)
        else:
            np.random.seed()

        # Verify links contain valid nodes  only
        for link in self.links:
            if link[0] not in self.nodes:
                raise ValueError('Node {n} was not declared in the list of
                ↪  nodes'.format(n=link[0]))
            elif link[1] not in self.nodes:
                raise ValueError('Node {n} was not declared in the list of
                ↪  nodes'.format(n=link[1]))
```

```
92
93          self.verbose = verbose
94
95      def assign_link_costs(self, costs='rigid'):
96          """Assigns costs to all links, generating them if necessary
97
98          Keyword Arguments:
99              costs {str or list} -- Either string to say which costs to use, or
100             list of costs for all links.
101                 'rigid' assigns a cost of 1 to each node
102                 'mult_int' assigns random non-zero positive integers
103                 'real' assigns random non-zero real valued numbers
104                 (default: {'rigid'})
105         """
106         self.costs = costs
107         if costs == 'rigid':
108             nx.set_edge_attributes(self.G, 1, 'cost')
109         elif costs == 'mult_int':
110             rand_costs = np.random.randint(1,5,len(self.G.edges))
111             c_dict = {e:rand_costs[i]
112                 for (i,e) in enumerate(self.G.edges)
113             }
114             nx.set_edge_attributes(self.G, c_dict, 'cost')
115         elif costs == 'real':
116             rand_costs = np.random.rand(len(self.G.edges))*4
117             # Do not want any link cost to be zero
118             while np.any(np.isclose(rand_costs, 0)):
119                 rand_costs = np.random.rand(len(self.G.edges))*4
120             c_dict = {e:rand_costs[i]
121                 for (i,e) in enumerate(self.G.edges)
122             }
123             nx.set_edge_attributes(self.G, c_dict, 'cost')
124         elif type(costs) == list:
125             c_dict = {e:costs[i]
126                 for (i,e) in enumerate(self.G.edges)
127             }
128             nx.set_edge_attributes(self.G, c_dict, 'cost')
129         else:
130             raise ValueError('argument to `costs` not recognised')
131         self.c = [cost for (_,_,cost) in self.G.edges.data('cost')]
132
133     def compute_spl_matrix(self):
134         """Computes shortest path length matrix F, which gives minimal cost to
135         reach any node i from any origin o.
136         """
137         # Initialise to infinity for all nodes, if a node is not reachable
138         # f will remain infinity
139         self.F = np.ones((len(self.nodes), len(self.nodes))) * np.inf
140         for spl in nx.shortest_path_length(self.G, weight='cost'):
141             for node in spl[1].keys():
142                 self.F[node, spl[0]] = spl[1][node]
```

```
143
144     def find_all_paths(self, tau_max=4, assignment='random'):
145         """Finds all feasible loop-free paths in given network, which are
146         shorter than tau_max. For this, shortest path matrix F is also computed.
147
148         Keyword Arguments:
149             tau_max {int or str} -- Maximum path length. If set to
150                 'mpl', tau_max is set to the length of the
151                 longest shortest path (default: {4})
152             assignment {str} -- Assignment, either 'random' or 'shortest_path'.
153                 Both assignments are loop-free. (default: {'random'})
154         """
155         self.compute_spl_matrix()
156         self.tau_max = tau_max
157         if self.tau_max == 'mpl':
158             self.tau_max = int(np.ceil((np.max(self.F[self.F < np.inf]))))
159         if min(self.c) >= self.tau_max:
160             print('No paths given current link costs, reassigning costs')
161             self.assign_link_costs(costs=self.costs)
162         self.assignment = assignment
163         self.paths = []
164         if self.assignment == 'shortest_path':
165             for n1 in self.G.nodes:
166                 for n2 in self.G.nodes:
167                     if n1 != n2:
168                         try:
169                             s_paths =
                                ↪ nx.algorithms.shortest_paths.generic.all_shortest_paths(
170                                 self.G, n1, n2, weight='cost'
171                             )
172                             for p in s_paths:
173                                 self.paths.append(p)
174                         except nx.exception.NetworkXNoPath:
175                             continue
176         elif self.assignment == 'random':
177             for n1 in self.G.nodes:
178                 for n2 in self.G.nodes:
179                     if n1 != n2:
180                         try:
181                             s_paths = nx.algorithms.simple_paths.all_simple_paths(
182                                 self.G, n1, n2, cutoff=self.tau_max
183                             )
184                             for p in s_paths:
185                                 self.paths.append(p)
186                         except nx.exception.NetworkXNoPath:
187                             continue
188         else:
189             raise ValueError('Assignment \'{a}\' is not recognised'.format(a=assignment))
190         # Store paths as sequence of link indexes as well
191         self.paths_links = []
192         for path in self.paths:
```

```
193             tmp_plink = []
194             for n_i in range(len(path)-1):
195                 tmp_plink.append(self.links.index((path[n_i],path[n_i+1])))
196             self.paths_links.append(tmp_plink)
197 
198         # Remove paths that exceed the maximum path length
199         if self.assignment == 'shortest_path':
200             # Maximum path length allowed
201             mpl = min(np.amax(self.F), self.tau_max)
202         else:
203             mpl = self.tau_max
204         pl_ps = sorted(list(filter(
205             lambda pl_p: sum(self.c[l] for l in pl_p[0]) <= mpl,
206             zip(self.paths_links, self.paths)
207         )))
208         self.paths_links, self.paths = [list(t) for t in zip(*pl_ps)]
209 
210         # Reduce tau_max if it is greater than longest path
211         if self.assignment == 'random':
212             mpl = max(map(
213                 lambda pl: sum(self.c[l] for l in pl),
214                 self.paths_links
215             ))
216         if self.tau_max > int(np.ceil(mpl)):
217             self.tau_max = int(np.ceil(mpl))
218             if self.verbose:
219                 print('Maximum path length tau_max changed to {tm}'.format(
220                     tm=self.tau_max)
221                 )
222 
223     def generate_od_pairs(self):
224         """
225         Generate set of Origin-Destination pairs for the given network.
226         """
227         assert(self.paths), 'Paths were not determined, or no paths exist.'
228 
229         self.od_pairs = sorted(list(
230             set(map(lambda p: (p[0],p[-1]), self.paths))
231         ))
232         self.origins = sorted(list(
233             set(map(lambda odp: odp[0], self.od_pairs))
234         ))
235         self.destinations = sorted(list(
236             set(map(lambda odp: odp[-1], self.od_pairs))
237         ))
238 
239     def compute_path_assignment_matrix(self):
240         """
241         Computes deterministic path assignment matrix, which depends on
242         topology of the network, assignment strategy and type of link costs.
243         NB: Random assignment is currently only supported for rigid costs.
```

```
        """
        assert(self.paths), 'Paths were not determined, or no paths exist.'
        # Initialise path assignment matrix and arrival/departure matrices
        self.Delta = np.zeros((len(self.links), len(self.paths)))
        self.Delta_ms = [
            np.zeros((len(self.links), len(self.paths)), np.int8)
            for _ in range(self.tau_max)
        ]
        self.T_minus = np.zeros((len(self.nodes), len(self.nodes)))
        self.T_plus = np.zeros((len(self.nodes), len(self.nodes)))
        # Compute
        if self.assignment == 'random':
            assert(all(cst == 1 for cst in self.c)), 'Random assignment only supported for
            ↪  rigid models'
            self.lc = np.ones((len(self.links), len(self.origins)))
            for l_idx in range(len(self.links)):
                for p_idx in range(len(self.paths_links)):
                    for tau in range(self.tau_max):
                        if tau < len(self.paths_links[p_idx]):
                            self.Delta_ms[tau][l_idx,p_idx] = (
                                l_idx == self.paths_links[p_idx][tau]
                            )
        elif self.assignment == 'shortest_path':
            self.T_minus = np.ceil(self.F)
            #self.T_minus = self.T_minus.astype(int)
            self.T_plus = np.floor(self.F + 1)
            #self.T_plus = self.T_plus.astype(int)
            self.lc = np.zeros((len(self.links), len(self.origins)))
            for (l_i,l) in enumerate(self.links):
                for (o_i,o) in enumerate(self.origins):
                    # Difference may be negative, or zero whenever the link is
                    # not used by the origin
                    if (
                        self.T_plus[l[0],o] != np.inf and
                        self.T_plus[l[1],o] != np.inf
                    ):
                        self.lc[l_i,o_i] = max(
                            self.T_minus[l[1],o] - self.T_plus[l[0],o] + 1,
                            1
                        )
                    else:
                        self.lc[l_i,o_i] = 1


            for (p_idx,pl) in enumerate(self.paths_links):
                o = self.links[pl[0]][0]
                for l_idx in pl:
                    i = self.links[l_idx][0]
                    j = self.links[l_idx][1]
                    for tau in range(self.tau_max):
                        if (
```

```
                            tau+1 >= self.T_plus[i,o] and
                            tau+1 <= self.T_minus[j,o] and
                            self.T_minus[j,o] != np.inf
                            ):
                            self.Delta_ms[tau][l_idx,p_idx] = 1
        # Obtain single-step path assignment matrix as sum of the steps
        self.Delta = sum(d for d in self.Delta_ms)

    def generate_random_proportions(
        self,
        like_paper=True,
        min_prop=0,
        max_prop=100
        ):
        """Generates random fixed proportions O to OD flows and OD to path flows.

        Keyword Arguments:
            like_paper {bool} -- Whether to generate proportions with same mean
                and variance as paper (default: {True})
            min_prop {int} -- Minimum flow proportion (default: {0})
            max_prop {int} -- Maximum flow proportion (default: {100})
        """

        assert(min_prop >=  0), 'Minimum proportion must be positive'
        assert(max_prop > 0), 'Maximum proportion must be strictly positive'
        assert(min_prop < max_prop+1), 'Maximum proportion must be greater than minimum
        ↪  proportion plus 1'

        self.o_od_proportions = np.zeros((len(self.origins), len(self.od_pairs)))
        self.od_path_proportions = np.zeros((len(self.od_pairs), len(self.paths)))
        if like_paper:
            # Generate path proportions
            path_proportions = 2 + np.random.rand(len(self.paths))
            path_proportions = path_proportions/np.sum(path_proportions)
            od_probs = np.zeros(len(self.od_pairs))
            for od_idx in range(len(self.od_pairs)):
                # Distribution of OD flow over path flows
                p_idxs = list(filter(
                    lambda p_i:
                        self.paths[p_i][0] == self.od_pairs[od_idx][0] and
                        self.paths[p_i][-1] == self.od_pairs[od_idx][1],
                    range(len(self.paths))
                ))
                od_fractions = [path_proportions[i] for i in p_idxs]
                od_probs[od_idx] = np.sum(od_fractions)
                od_fractions = od_fractions/od_probs[od_idx] # Normalise
                for (i,p_idx) in enumerate(p_idxs):
                    self.od_path_proportions[od_idx,p_idx] = od_fractions[i]
            for o_idx in range(len(self.origins)):
                # Distribution of O flow over OD flows
                od_idxs = list(filter(
```

```
                    lambda odp_i: self.od_pairs[odp_i][0] == self.origins[o_idx],
                    range(len(self.od_pairs))
                ))
                o_fractions = [od_probs[i] for i in od_idxs]
                o_fractions = o_fractions/np.sum(o_fractions) # Normalise
                for (i,od_idx) in enumerate(od_idxs):
                    self.o_od_proportions[o_idx,od_idx] = o_fractions[i]

        else:
            for o_idx in range(len(self.origins)):
                # Distribution of O flow over OD flows
                od_idxs = list(filter(
                    lambda odp_i: self.od_pairs[odp_i][0] == self.origins[o_idx],
                    range(len(self.od_pairs))
                ))
                o_fractions = np.zeros((1,len(od_idxs)))
                while np.sum(o_fractions) == 0:
                    o_fractions = np.random.randint(
                        min_prop, max_prop,
                        len(od_idxs)
                    )
                # Normalise distribution
                o_fractions = o_fractions/np.sum(o_fractions)
                for (i,od_idx) in enumerate(od_idxs):
                    self.o_od_proportions[o_idx,od_idx] = o_fractions[i]
            for od_idx in range(len(self.od_pairs)):
                # Distribution of OD flow over path flows
                p_idxs = list(filter(
                    lambda p_i:
                        self.paths[p_i][0] == self.od_pairs[od_idx][0] and
                        self.paths[p_i][-1] == self.od_pairs[od_idx][1],
                    range(len(self.paths))
                ))
                od_fractions = np.zeros((1,len(p_idxs)))
                while np.sum(od_fractions) == 0:
                    od_fractions = np.random.randint(
                        min_prop, max_prop,
                        len(p_idxs)
                    )
                # Normalise distribution
                od_fractions = od_fractions/np.sum(od_fractions)
                for (i,p_idx) in enumerate(p_idxs):
                    self.od_path_proportions[od_idx,p_idx] = od_fractions[i]

    def compute_assignment_matrix(self):
        """
        Computes assignment matrix A, and O-flow assignment matrix P based on
        traffic proportions of the network. Assignment matrix will be different
        based on assignment strategy selected when finding all paths, and on
        type of costs selected when generating link costs.
        """
```

```
395         assert(self.Delta.size is not None), 'Path assignment matrix was not computed.'
396         assert(self.o_od_proportions.size is not None), 'O to OD proportions have not been
            ↪  generated.'
397         assert(self.od_path_proportions.size is not None), 'OD to path proportions have not
            ↪  been generated.'
398         assert(self.assignment is not None), 'Assignment strategy has not been determined.'
399 
400         # Initialise traffic assignment matrices
401         self.A = np.zeros((len(self.links), len(self.od_pairs)))
402         self.P = np.zeros((len(self.links), len(self.origins)))
403         self.A_ms = [np.zeros((len(self.links), len(self.od_pairs))) for i in
            ↪  range(self.tau_max)]
404         self.P_ms = [np.zeros((len(self.links), len(self.origins))) for i in
            ↪  range(self.tau_max)]
405 
406         for (od_i, od) in enumerate(self.od_pairs):
407             p_idxs = list(filter(
408                 lambda p_i: self.paths[p_i][0] == od[0] and self.paths[p_i][-1] == od[1],
409                 range(len(self.paths))
410             ))
411             for p_i in p_idxs:
412                 for tau in range(self.tau_max):
413                     self.A_ms[tau][:,od_i] +=
                        ↪  self.Delta_ms[tau][:,p_i]*self.od_path_proportions[od_i,p_i]
414         self.A = sum(a for a in self.A_ms)
415 
416         for (o_i, o) in enumerate(self.origins):
417             od_idxs = list(filter(
418                 lambda od_i: self.od_pairs[od_i][0] == o,
419                 range(len(self.od_pairs))
420             ))
421             for od_i in od_idxs:
422                 for tau in range(self.tau_max):
423                     self.P_ms[tau][:,o_i] +=
                        ↪  self.A_ms[tau][:,od_i]*self.o_od_proportions[o_i,od_i]
424         self.P = sum(p for p in self.P_ms)
425 
426     def duplicate_network(self):
427         """
428         Returns a copy of the current network with the same topology and
429         assignment.
430         """
431         return deepcopy(self)
```

Listing 3: solver.py

```
1 """
2 Solver class is declared in this file. Notably, constraint matrices are
3 generated here for single-step, multi-step models, and for random, shortest path
4 assignment strategies.
```

```
5   """
6
7   import numpy as np
8
9   class Solver():
10      """
11      Class containing methods for obtaining constraint matrices to use in
12      Matlab CVX optimization routines. The constraints that can be obtained are:
13
14      for all assignment types:
15          (c2: probability constraint)
16              Fully included in speed constraint
17          c3: observability constraint
18              The sum of flows out of each origin during first step
19              should sum up to 1
20          c4: speed constraint
21              The speed constraint includes reachability and for
22              shortest path assignment only, allowed time steps
23              for each origin/link pair
24          c5: flow constraint, which is made of
25              - c5_in_edges: indicates whether link flows in a node
26              - c5_out_edges: indicates whether link flows out of node
27              - c5_in_check: indicates whether node is distinct from origin
28                  and for multi-step model whether time-step is valid
29              - c5_out_check: indicates whether node is distinct from origin
30                  and for multi-step model whether time-step is valid
31              The total O-flow that enters a node i (!=o) must be
32              greater or equal than the total O-flow that leaves
33              that same node. One constraint per node in the network
34      for multi-step shortest path assignment only:
35          (c6: shortest path constraint)
36              Fully included in speed constraint
37          c7: duplicate counts constraint, which is made of
38              - c7_enter_link: indicates O-flows that enter link at
39                  that time step
40              - c7_end_link: indicates O-flows that are still in link at
41                  a time step later than that in which they entered it
42      """
43
44      net = None
45
46      def __init__(self, net):
47          """Constructor of Solver class
48
49          Arguments:
50              net {Network} -- Network instance, where assignment matrices and
51                  link costs have already been set.
52          """
53          self.net = net
54
55      def get_single_step_constraints(self):
```

```
        """
        Generate constraint matrices for single-step model of network.
        """
        dims = self.net.P.shape

        #C4
        net2 = self.net.duplicate_network()
        # Generate positive flows through each link where flows are allowed
        net2.generate_random_proportions(1,2)
        net2.compute_assignment_matrix()
        if np.array_equal(net2.P, self.net.P):
            print('Assignment matrix P is the same')
        self.c4 = (net2.P > 0)
        #C3
        self.c3 = np.full(dims, False)
        for (l_i,l) in enumerate(self.net.links):
            for (o_i,o) in enumerate(self.net.origins):
                if l[0] == o:# and self.c4[l_i,o_i]:
                    self.c3[l_i,o_i] = True
        #C5
        if len(self.net.origins) <  len(self.net.nodes) and self.net.verbose:
            print('\n\nWARNING: Less origins than nodes in the network, check this is
            ↪  expected\n\n')

        self.c5_in_edges = np.full(
            (len(self.net.links), len(self.net.nodes)),
            False
        )
        self.c5_out_edges = np.full(
            (len(self.net.links), len(self.net.nodes)),
            False
        )
        self.c5_in_check = np.full(
            (len(self.net.origins), len(self.net.nodes)),
            False
        )
        self.c5_out_check = np.full(
            (len(self.net.origins), len(self.net.nodes)),
            False
        )
        for (l_i, l) in enumerate(self.net.links):
            for (n_i,n) in enumerate(self.net.nodes):
                if l[1] == n:
                    self.c5_in_edges[l_i, n_i] = True
                elif l[0] == n:
                    self.c5_out_edges[l_i, n_i] = True
        for (o_i,o) in enumerate(self.net.origins):
            for (n_i,n) in enumerate(self.net.nodes):
                if o != n:
                    self.c5_in_check[o_i, n_i] = True
                    self.c5_out_check[o_i, n_i] = True
```

```
106
107         #C6 is expressed by c4 in the single-step model
108         #C7 cannot be enforced in the single-step model
109
110     def get_multi_step_constraints(self):
111         """
112         Generates constraints for multi-step model of the network.
113         """
114         P = np.stack(self.net.P_ms)
115
116         #C3
117         self.c3 = np.full(P.shape, False)
118         for (l_i,l) in enumerate(self.net.links):
119             for (o_i,o) in enumerate(self.net.origins):
120                 if l[0] == o:
121                     # We only need to set this for P[0], constraints 4 and 6
122                     # will take care of later time steps
123                     self.c3[0,l_i,o_i] = True
124
125         #C4
126         net2 = self.net.duplicate_network()
127         # Generate positive flows through each link where flows are allowed
128         net2.generate_random_proportions(1,2)
129         net2.compute_assignment_matrix()
130         if np.array_equal(net2.P_ms, self.net.P_ms):
131             print('Assignment matrix P_ms is the same')
132         P2 = np.stack(net2.P_ms)
133         self.c4 = (P2 > 0)
134
135         # Initialising constraint 5 matrices
136         if len(self.net.origins) <  len(self.net.nodes) and self.net.verbose:
137             print('\n\nWARNING: Less origins than nodes in the network, check this is
                ↪  expected\n\n')
138         self.c5_in_edges = np.full(
139             (len(self.net.links), len(self.net.nodes)),
140             False
141         )
142         self.c5_out_edges = np.full(
143             (len(self.net.links), len(self.net.nodes)),
144             False
145         )
146         self.c5_in_check = np.full(
147             (self.net.tau_max, len(self.net.origins), len(self.net.nodes)),
148             False
149         )
150         self.c5_out_check = np.full(
151             (self.net.tau_max, len(self.net.origins), len(self.net.nodes)),
152             False
153         )
154
155         if self.net.assignment == 'random':# or self.net.assignment == 'shortest_path':
```

```
156             #C5
157             for (l_i, l) in enumerate(self.net.links):
158                 for (n_i,n) in enumerate(self.net.nodes):
159                     if l[1] == n:
160                         self.c5_in_edges[l_i, n_i] = True
161                     elif l[0] == n:
162                         self.c5_out_edges[l_i, n_i] = True
163             for (o_i, o) in enumerate(self.net.origins):
164                 for (n_i,n) in enumerate(self.net.nodes):
165                     if o != n:
166                         for tau in range(self.net.tau_max-1):
167                             self.c5_in_check[tau, o_i, n_i] = True
168                         for tau in range(1, self.net.tau_max):
169                             self.c5_out_check[tau, o_i, n_i] = True
170         elif self.net.assignment == 'shortest_path':
171             #C5
172             # For shortest path assignment, only need to take into account
173             # two time steps for flow contraint. The rest is set to 0 by C4.
174             for (l_i, l) in enumerate(self.net.links):
175                 for (n_i,n) in enumerate(self.net.nodes):
176                     if l[1] == n:
177                         self.c5_in_edges[l_i, n_i] = True
178                     elif l[0] == n:
179                         self.c5_out_edges[l_i, n_i] = True
180             for (o_i, o) in enumerate(self.net.origins):
181                 for (n_i,n) in enumerate(self.net.nodes):
182                     if o != n:
183                         arr_tau = self.net.T_minus[n_i,o] - 1
184                         dep_tau = self.net.T_plus[n_i,o] - 1
185                         if (
186                                 arr_tau < self.net.tau_max and
187                                 dep_tau < self.net.tau_max and
188                                 dep_tau >= 0 and
189                                 arr_tau >= 0 and
190                                 arr_tau <= dep_tau
191                             ):
192                             self.c5_in_check[int(arr_tau), o_i, n_i] = True
193                             self.c5_out_check[int(dep_tau), o_i, n_i] = True
194 
195             #C7
196             # time-step when 0-flow enters link
197             self.c7_enter_link = np.zeros(
198                 (len(self.net.links), len(self.net.origins)),
199                 dtype=int
200             )
201             # time-step when 0-flow reaches end of link
202             self.c7_end_link = np.zeros(
203                 (len(self.net.links), len(self.net.origins)),
204                 dtype=int
205             )
206             # Elements of P2 that are greater than 0 should remain constant
```

```
            # throughout their traversal of the link, by shortest path
            # assumption
            for (l_i,l) in enumerate(self.net.links):
                for (o_i,o) in enumerate(self.net.origins):
                    # Assume l is link ij
                    # enter_tau is time step when O flow leaves i (enters link ij)
                    enter_tau = self.net.T_plus[l[0],o] - 1
                    # leave_tau is time step when O flow reaches j (reaches end of link ij)
                    leave_tau = self.net.T_minus[l[1],o] - 1
                    if leave_tau < self.net.tau_max - 1 and leave_tau > enter_tau:
                        self.c7_enter_link[l_i,o_i] = enter_tau
                        self.c7_end_link[l_i,o_i] = leave_tau

        # Correctly reshaping constraint matrices for usage in Matlab routine
        # P is stacked along dimension 1 in order to obtain a
        # n_l by (n_o*tau_max) 2-dimensional matrix
        P = np.concatenate(P, axis=1)
        self.c3 = np.concatenate(self.c3, axis=1)
        self.c4 = np.concatenate(self.c4, axis=1)
        self.c5_in_check = np.concatenate(self.c5_in_check, axis=0)
        self.c5_out_check = np.concatenate(self.c5_out_check, axis=0)
```

Listing 4: to_matlab.py

```
"""
Script containing useful methods to export data produced in Python in
Matlab readable formats, so that it can safely be loaded and used with the CVX
optimisation routine.
"""

import os
import numpy as np
import scipy.io as io

from network import Network
from solver import Solver

class ToMatlab:
    """
    Class for converting network, assignment and flow structures generated in
    Python, to Matlab matrices so that they can be used for optimization problem
    """

    def __init__(
            self,
            net,
            step='multi',
            trials=5,
            m_dir='../../OflowEstimationFull/tests/'
        ):
```

```
27          self.net = net
28          self.step = step
29          self.trials = trials
30          self.m_dir = m_dir
31
32      def save_assignment(self):
33          sp = self.net.assignment == 'shortest_path'
34          io.savemat(self.m_dir+'shortest_path', {'shortest_path':sp})
35
36      def save_trials(self):
37          io.savemat(self.m_dir+'trials', {'trials':self.trials})
38
39      def save_tau_max(self):
40          io.savemat(self.m_dir+'tau_max', {'tau_max':self.net.tau_max})
41
42      def convert_P(self, ext):
43          if self.step == 'multi':
44              # 4-dimensional array
45              P_py = np.zeros((
46                  len(self.net.links),    # links
47                  len(self.net.origins),  # origins
48                  self.net.tau_max,       # time steps
49                  self.trials             # trials
50              ))
51              for tr in range(self.trials):
52                  self.net.generate_random_proportions()
53                  self.net.compute_assignment_matrix()
54                  P_py[:,:,:,tr] = np.dstack(self.net.P_ms)
55                  # print('P'+ext)
56                  # print(P_py[:,:,:,0])
57
58          if self.step == 'single':
59              # 3-dimensional array
60              P_py = np.zeros((
61                  len(self.net.links),    # links
62                  len(self.net.origins),  # origins
63                  self.trials             # trials
64              ))
65              for tr in range(self.trials):
66                  self.net.generate_random_proportions()
67                  self.net.compute_assignment_matrix()
68                  P_py[:,:,tr] = self.net.P
69
70          io.savemat(self.m_dir+'P_'+ext, {'P_'+ext:P_py})
71
72      def convert_o_list(self):
73          o_list_py = np.array(self.net.origins) + 1
74          io.savemat(self.m_dir+'o_list', {'o_list':o_list_py})
75
76      def convert_e_list(self):
77          e_list_py = np.array(self.net.links) + 1
```

```
78          io.savemat(self.m_dir+'e_list', {'e_list':e_list_py})
79
80      def convert_od_list(self):
81          od_list_py = np.array(self.net.od_pairs) + 1
82          io.savemat(self.m_dir+'od_list', {'od_list':od_list_py})
83
84      def convert_lc(self):
85          lc_py = self.net.lc
86          io.savemat(self.m_dir+'lc', {'lc':lc_py})
87
88      def convert_constraints(self):
89          solver = Solver(self.net)
90          if self.step == 'multi':
91              solver.get_multi_step_constraints()
92              if self.net.assignment == 'shortest_path':
93                  io.savemat(
94                      self.m_dir+'constraints',
95                      {
96                          'c3':solver.c3,
97                          'c4':solver.c4,
98                          'c5_in_edges':solver.c5_in_edges,
99                          'c5_in_check':solver.c5_in_check,
100                         'c5_out_edges':solver.c5_out_edges,
101                         'c5_out_check':solver.c5_out_check,
102                         'c7_enter_link':solver.c7_enter_link,
103                         'c7_end_link':solver.c7_end_link
104                     }
105                 )
106             elif self.net.assignment == 'random':
107                 io.savemat(
108                     self.m_dir+'constraints',
109                     {
110                         'c3':solver.c3,
111                         'c4':solver.c4,
112                         'c5_in_edges':solver.c5_in_edges,
113                         'c5_in_check':solver.c5_in_check,
114                         'c5_out_edges':solver.c5_out_edges,
115                         'c5_out_check':solver.c5_out_check
116                     }
117                 )
118         elif self.step == 'single':
119             solver.get_single_step_constraints()
120             io.savemat(
121                     self.m_dir+'constraints',
122                     {
123                         'c3':solver.c3,
124                         'c4':solver.c4,
125                         # 'c5_inflows':solver.c5_inflows,
126                         # 'c5_outflows':solver.c5_outflows,
127                         'c5_in_edges':solver.c5_in_edges,
128                         'c5_in_check':solver.c5_in_check,
```

```
                        'c5_out_edges':solver.c5_out_edges,
                        'c5_out_check':solver.c5_out_check
                    }
                )

    def convert_data(self):

        # One will be P_target, the second will be P_initialise
        self.save_assignment()
        self.save_trials()
        self.save_tau_max()
        self.convert_P('target')
        self.convert_P('initialise')
        self.convert_o_list()
        self.convert_e_list()
        self.convert_od_list()
        self.convert_lc()
        self.convert_constraints()

if __name__ == '__main__':
    # SETTING UP EXPERIMENT
    STEP = 'multi'
    COSTS = 'real'
    ASSMENT = 'shortest_path'
    H = 3
    W = 3
    TAU_MAX = 4
    TRIALS = 3
    DIRECTION = 'bi'

    DATA = 'histogram' # 'histogram' or 'plot'
    PARAM = 'n_T' # n_T or 'size'

    # NO NEED TO MODIFY AFTER THIS POINT
    if DATA == 'histogram':
        m_dir =
        ↪ '../../OflowEstimationFull/tests/{step}_{costs}_{assment}_{h}by{w}_{tau_max}taumax_{trials}
        # m_dir =
        ↪ '../../temp/tests/RANDCONSC4C5_{step}_{costs}_{assment}_{h}by{w}_{tau_max}taumax_{trials}tr
            step=STEP,
            costs=COSTS,
            assment=ASSMENT,
            h=H,
            w=W,
            tau_max=TAU_MAX,
            trials=TRIALS
        )
        if not os.path.exists(m_dir):
            os.mkdir(m_dir)
        else:
            print('WARNING: test data in {dir} will be overwritten'.format(dir=m_dir))
```

```
178             ans = input('Press any key to continue, or \'n\' to stop execution')
179             if ans == 'n':
180                 print('Stopping execution.')
181                 exit()
182             else:
183                 print('Overwriting files.')
184 
185         net = Network(DIRECTION, h=H, w=W)
186         net.assign_link_costs(COSTS)
187         net.find_all_paths(tau_max=TAU_MAX, assignment=ASSMENT)
188         net.generate_od_pairs()
189         net.compute_path_assignment_matrix()
190         conv = ToMatlab(
191             net, step=STEP, trials=TRIALS,
192             m_dir=m_dir
193         )
194         conv.convert_data()
195     elif DATA == 'plot':
196         m_dir =
        ↪ '../../OflowEstimationFull/tests/{param}_{step}_{costs}_{assment}_{h}by{w}_{tau_max}taumax_
197             param=PARAM,
198             step=STEP,
199             costs=COSTS,
200             assment=ASSMENT,
201             h=H,
202             w=W,
203             tau_max=TAU_MAX,
204             trials=1
205         )
206         if not os.path.exists(m_dir):
207             os.mkdir(m_dir)
208         else:
209             print('WARNING: test data in {dir} will be overwritten'.format(dir=m_dir))
210             ans = input('Press any key to continue, or \'n\' to stop execution')
211             if ans == 'n':
212                 print('Stopping execution.')
213                 exit()
214             else:
215                 print('Overwriting files.')
216         if PARAM == 'n_T':
217             TRIALS = 1
218             net = Network(DIRECTION, h=H, w=W)
219             net.assign_link_costs(COSTS)
220             net.find_all_paths(tau_max=TAU_MAX, assignment=ASSMENT)
221             net.generate_od_pairs()
222             net.compute_path_assignment_matrix()
223             conv = ToMatlab(
224                 net, step=STEP, trials=1,
225                 m_dir=m_dir
226             )
227             conv.convert_data()
```

```
228         elif PARAM == 'size':
229             TRIALS=1
230             #size: 4, 6, 9, 12, 16,
231             hw_list = [(2,2), (2,3), (3,3), (3,4), (4,4), (4,5), (5,5), (5,6), (6,6), (7,7),
                ↪  (8,8)]
232             for i in range(len(hw_list)):
233                 (H,W) = hw_list[i]
234                 net = Network(DIRECTION, h=H, w=W)
235                 net.assign_link_costs(COSTS)
236                 net.find_all_paths(tau_max=TAU_MAX, assignment=ASSMENT)
237                 net.generate_od_pairs()
238                 net.compute_path_assignment_matrix()
239                 m_dir =
                    ↪  '../../OflowEstimation/tests/{param}_{step}_{costs}_{assment}_{h}by{w}_{tau_max}tau
240                     param=PARAM,
241                     step=STEP,
242                     costs=COSTS,
243                     assment=ASSMENT,
244                     h=H,
245                     w=W,
246                     tau_max=TAU_MAX,
247                     trials=1
248                 )
249                 if not os.path.exists(m_dir):
250                     os.mkdir(m_dir)
251                 conv = ToMatlab(
252                     net, step=STEP, trials=1,
253                     m_dir=m_dir
254                 )
255                 conv.convert_data()
```